Dymola

Dynamic Modeling Laboratory

Full User Manual

(All Chapters from User Manual 1A – User Manual 2C Included)

March 2022 (Dymola 2023)

The information in this document is subject to change without notice.

© Copyright 1992-2022 by Dassault Systèmes AB. All rights reserved.
Dymola® is a registered trademark of Dassault Systèmes AB.
Modelica® is a registered trademark of the Modelica Association.

Other product or brand names are trademarks or registered trademarks of their respective holders.

Dassault Systèmes AB
Ideon Gateway
Scheelevägen 27 – Floor 9
SE-223 63 Lund
Sweden

E-mail: https://www.3ds.com/support
URL: https://www.dymola.com/
Phone: +46 46 270 67 00

Contents

1 What is Dymola? 17
1.1 Features of Dymola 17
1.2 Simulating an existing model 18
1.3 Building a model 22
1.4 Architecture of Dymola 24
1.5 Features of Modelica 25
1.6 Functional Mockup Interface 27
2 Getting started with Dymola 31
2.1 Introduction 31
2.2 Simulating a model — industrial robot 34
2.2.1 Investigating the robot before simulating it 34
2.2.2 Simulation 46
2.2.3 Other demo examples 51
2.3 Solving a non-linear differential equation 52
2.3.1 Simulation 57
2.3.2 Improving the model 62
2.4 Using the Modelica Standard Library 72
2.4.1 The Modelica Standard Library 73
2.4.2 Creating a library for components 83
2.4.3 Creating a model for an electric DC motor 84
2.4.4 Documenting the model 95
2.4.5 Testing the model 98
2.4.6 Handling the warnings 104
2.4.7 Creating a model for the motor drive 109
2.4.8 Parameter expressions 116

2.4.9 Documenting the simulation ... 122
2.4.10 Scripting ... 123
2.5 Building a mechanical model ... 126
2.6 Other libraries ... 132
2.6.1 Libraries available in the File menu by default ... 132
2.6.2 Libraries that can be added ... 135
2.7 Help and information ... 136
2.7.1 Reaching compressed information ... 136
2.7.2 Reaching more extensive information ... 136
2.8 Shortcuts ... 138
3 Introduction to Modelica ... 149
3.1 Modelica basics ... 149
3.1.1 Variables ... 151
3.1.2 Connectors and connections ... 151
3.1.3 Balanced models ... 152
3.1.4 Partial models and inheritance ... 153
3.2 Acausal modeling ... 154
3.2.1 Background ... 154
3.2.2 Differential-algebraic equations ... 155
3.2.3 Stream connector support ... 156
3.3 Advanced modeling features ... 157
3.3.1 Vectors, matrices and arrays ... 157
3.3.2 Class parameters ... 158
3.3.3 Algorithms and functions ... 159
3.4 Hybrid modeling in Modelica ... 159
3.4.1 Synchronous equations ... 160
3.4.2 Relation triggered events ... 162
3.4.3 Variable structure systems ... 164
3.5 Initialization of models ... 167
3.5.1 Basics ... 168
3.5.2 Continuous time problems ... 168
3.5.3 Parameter values ... 172
3.5.4 Discrete and hybrid problems ... 173
3.5.5 Example: Initialization of discrete controllers ... 175
3.6 Standard libraries ... 178
3.6.1 Modelica Standard Library ... 178
3.6.2 Modelica Reference ... 179
3.6.3 Other libraries ... 179
3.7 References ... 179
4 Developing a model ... 185
4.1 Windows, tabs and browsers available when developing a model ... 186
4.1.1 Dymola Main window ... 187
4.1.2 Edit window ... 189
4.1.3 Model tabs ... 200
4.1.4 Package and component browsers ... 202
4.1.5 Library window ... 213
4.1.6 Command window ... 214

4.1.7 Log window 215
4.1.8 Information browser 215
4.2 Basic model editing 217
4.2.1 Basic operations 217
4.2.2 Packages, models, and other classes 227
4.2.3 Components and connectors 235
4.2.4 Connections 247
4.2.5 Creating graphical objects 261
4.2.6 Changing graphical attributes 270
4.2.7 Programming in Modelica 271
4.2.8 Documentation 300
4.2.9 Selection of active simulation model 325
4.2.10 Support for national characters 327
4.3 Advanced model editing 329
4.3.1 Storing packages hierarchically 329
4.3.2 State Machines 331
4.3.3 Locking and unlocking classes 335
4.3.4 Creating a new model from a component with parameter values included 336
4.3.5 Splitting models 338
4.3.6 Components and connectors 345
4.3.7 Media propagation using components from Modelica.Fluid 366
4.3.8 Building templates using replaceable components 369
4.3.9 Building test-driven template models 372
4.3.10 Cleaning up a model developed step-by-step 376
4.3.11 Parameters, variables and constants 376
4.3.12 String parameters 393
4.3.13 Saving of parameter records 395
4.3.14 Searching records 397
4.3.15 Defining and editing enumeration types 401
4.3.16 Matching and variable selections 404
4.3.17 Using data from files 420
4.3.18 Display units 423
4.3.19 Working with annotations using the Annotations dialog 426
4.3.20 General GUI for displaying and setting flags 431
4.4 Checking the model 437
4.4.1 Commands 437
4.4.2 Unit checking and unit deduction in Dymola 437
4.5 Editing model reference 443
4.5.1 Window settings 444
4.5.2 Default view of classes 445
4.5.3 Package browser details 445
4.6 Editor command reference when developing a model 446
4.6.1 Main window: File menu 446
4.6.2 Main window: Graphics tab 467
4.6.3 Main window: Documentation tab 485
4.6.4 Main window: Text tab 488
4.6.5 Main window: Tools tab 495
4.6.6 Main window: Window menu 528

4.6.7 Main window: Toggle button for ribbon information level 529
4.6.8 Main window: Model layer toolbar 529
4.6.9 Context menu: Edit window (Icon layer) 529
4.6.10 Context menu: Edit window (Diagram layer) 530
4.6.11 Context menu: Edit window (Documentation layer) 531
4.6.12 Context menu: Edit window (Modelica Text layer) 533
4.6.13 Context menu: Edit window (Modelica Text layer – mathematical notation) 540
4.6.14 Context menu: Edit window (Used Classes layer) 541
4.6.15 Context menu: Model tabs 542
4.6.16 Context menu: Components 543
4.6.17 Context menu: Coordinate system boundary 553
4.6.18 Context menu: Connections 554
4.6.19 Context menu: Connection while connecting 555
4.6.20 Context menu: Connectors 555
4.6.21 Context menu: Graphical objects 555
4.6.22 Context menu: Variable declaration dialog; Value input field 559
4.6.23 Context menu: Parameter dialog; plot window 571
4.6.24 Context menu: Parameter dialog; parameter input field 572
5 Simulating a model 583
5.1 Windows available when working with simulation 584
5.1.1 Dymola Main window 585
5.1.2 Variable browser 588
5.1.3 Model View window 590
5.1.4 Plot window and Plot tab 591
5.1.5 Animation window and Animation tab 596
5.1.6 Visualizer window 598
5.1.7 Command window 599
5.1.8 Script Editor window and Script Editor tab 602
5.1.9 Log window 603
5.1.10 Information browser 613
5.2 Model simulation 614
5.2.1 Basic steps 614
5.2.2 Variable browser interaction 619
5.2.3 Using the Model View window 679
5.2.4 Plotting 690
5.2.5 Plot window interaction 692
5.2.6 Errors and warnings when translating 753
5.2.7 Animation 757
5.2.8 Animation window interaction 760
5.2.9 Using the command window 764
5.2.10 Working with documentation in the command window 771
5.2.11 Working with the command history 780
5.2.12 Simulation settings 782
5.3 Editor command reference when working with simulation 790
5.3.1 Main window: File menu 790
5.3.2 Main window: Simulation tab 790
5.3.3 Main window: Script Editor tab 832

5.3.4 Main window: Plot tab: Options subtab 839
5.3.5 Main window: Plot tab: Layout subtab 858
5.3.6 Main window: Animation tab 860
5.3.7 Main window: Tools tab 868
5.3.8 Main window: Window menu 868
5.3.9 Main window: Toggle button for ribbon information level 869
5.3.10 Context menu: Variable browser – nodes 869
5.3.11 Context menu: Variable browser – signals and arrays 873
5.3.12 Context menu: Plot window 874
5.3.13 Context menu: Plot window – curve and legend 876
5.3.14 Context menu: Plot window – signal operators 877
5.3.15 Context menu: Plot window – text objects 878
5.3.16 Context menu: Table window 879
5.3.17 Context menu: Table window - value 879
5.3.18 Context menu: Animation window 880
5.3.19 Context menu: Visualizer window 882
5.3.20 Context menu: Command window – Command log pane 882
5.3.21 Context menu: Command window – Command input line 883
5.3.22 Context menu: Command history window 885
5.3.23 Context menu: Script editor 885
5.3.24 Context menu: Log window 890
5.4 Dynamic Model Simulator 892
5.4.1 Overview 892
5.4.2 Running Dymosim 893
5.4.3 Selecting the integration algorithm 895
5.4.4 Dymosim reference 902
5.5 Scripting 904
5.5.1 Basic facts about scripting in Dymola 904
5.5.2 Scripting in Dymola – Functions/Script files 908
5.5.3 The possible content in a Modelica function or script file 909
5.5.4 What is a Modelica function? 912
5.5.5 Creating a Modelica function 914
5.5.6 Saving a function call as a command in the model 914
5.5.7 Editing a Modelica function 915
5.5.8 Executing a Modelica function call 916
5.5.9 Advanced use of functions 916
5.5.10 Pre-defined Modelica functions 917
5.5.11 What is a Modelica script file? 917
5.5.12 Creating a Modelica script file 918
5.5.13 Saving a script file as a command in the model 921
5.5.14 Editing a Modelica script file 922
5.5.15 Running a Modelica script file 925
5.5.16 The Dymola script editor 926
5.5.17 Some examples of Modelica script files 933
5.5.18 Predefined Modelica script files 936
5.5.19 Built-in functions in Dymola 937
5.6 Debugging models 974
5.6.1 Guidelines for model development 974

5.6.2 Error messages....976
5.6.3 Controlling the amount of diagnostics for nonlinear solvers in the simulation log....977
5.6.4 Direct link in the error log to variables in model window....982
5.6.5 Saving periodic snapshots during simulation....985
5.6.6 Event logging....985
5.6.7 Model instability....986
5.6.8 Output of manipulated equations in Modelica format....987
5.6.9 Initial guesses for nonlinear equations in the model during simulation....997
5.6.10 Checking for unconnected physical connectors....998
5.6.11 Checking for structural singularities....998
5.6.12 Bound checking for variables....1000
5.6.13 Diagnostic message for inline handling....1001
5.6.14 Online diagnostics for non-linear systems....1001
5.6.15 Diagnostics for stuck simulation....1002
5.7 Improving simulation efficiency....1005
5.7.1 Time of storing result....1005
5.7.2 Simulation speed-up....1006
5.7.3 Events and chattering....1012
5.7.4 Debug facilities when running a simulation....1013
5.7.5 Profiling....1017
5.7.6 Inline integration....1021
5.8 Initialization, steady state, and start values....1024
5.8.1 The continue command....1024
5.8.2 Over-specified initialization problems....1024
5.8.3 Discriminating start values....1024
5.8.4 Debugging failed initialization....1027
5.8.5 Steady state solver interface....1029
5.8.6 Steady state initialization....1037
6 Model Experimentation....1043
6.1 Introduction....1043
6.2 Sweeping parameters using new GUI....1044
6.2.1 General....1044
6.2.2 Sweeping one parameter....1045
6.2.3 Sweeping two parameters....1054
6.2.4 Sweeping more than two parameters....1055
6.2.5 Storing and retrieving a sweep setup in the model....1059
6.3 Varying parameters of a model....1064
6.3.1 Case Study: CoupledClutches model....1066
6.3.2 Response to parameter perturbations - perturbParameters....1067
6.3.3 Sweep one parameter – two variants....1073
6.3.4 Sweep two parameters - sweepTwoParameters....1080
6.3.5 Sweeping many parameters - sweepManyParameters....1082
6.3.6 Monte Carlo Analysis....1082
7 Model Calibration....1093
7.1 Introduction....1093
7.2 The basics of setting up and executing a calibration task....1095
7.2.1 Vehicle data....1096

7.2.2 Vehicle model ... 1098
7.2.3 Validation of the nominal model ... 1100
7.2.4 Measurement file formats ... 1108
7.2.5 Calibration ... 1111
7.2.6 Free start values ... 1114
7.2.7 Tune the parameters ... 1115
7.2.8 Validation using measurements from first gear ... 1115
7.2.9 The setup as Modelica code ... 1118
7.3 Saving the setup for reuse ... 1118
7.4 Reusing a setup for a similar operation ... 1120
7.5 Analysing parameter sensitivities and dependencies ... 1121
7.5.1 Sweep one parameter – sweepParameter ... 1121
7.5.2 Sweep two parameters – sweepTwoParameters ... 1129
7.5.3 Response to parameter perturbations - perturbParameters ... 1131
7.5.4 Check if tuners can be calibrated – checkCalibrationSensitivity ... 1133
7.6 Data Preprocessing ... 1135
7.6.1 Setting up for preprocessing ... 1135
7.6.2 Limiting and detrending signals ... 1139
7.6.3 Analyzing Signals: is there any noise? ... 1141
7.6.4 Filtering signals ... 1144
7.7 Static calibration ... 1146
7.7.1 The staticCalibrate function ... 1146
7.7.2 The calibrateSteadyState function ... 1158
8 Design Optimization ... 1163
9 Model Management ... 1167
9.1 Version management ... 1167
9.1.1 Short guide with new features included ... 1167
9.1.2 The context of version management ... 1167
9.1.3 Introduction to version management ... 1169
9.1.4 Scope of implementation ... 1171
9.1.5 Supported features ... 1171
9.1.6 Selecting version management system ... 1175
9.1.7 Version management using CVS ... 1176
9.1.8 An example of file management using CVS ... 1178
9.1.9 Version management using SVN ... 1184
9.1.10 An example of file management using SVN ... 1186
9.1.11 Version management using Git ... 1190
9.1.12 Short guide to version management with new features included ... 1191
9.1.13 References ... 1192
9.2 Model dependencies ... 1193
9.2.1 Cross-reference options ... 1194
9.3 Encryption in Dymola ... 1195
9.3.1 Introduction ... 1195
9.3.2 Visible and concealed classes ... 1195
9.3.3 Developing encrypted libraries ... 1196
9.3.4 Using encrypted components ... 1197
9.3.5 Examples ... 1198

9.3.6 Special annotations for concealment 1207
9.3.7 Licensing libraries 1210
9.3.8 Scrambling in Dymola 1212
9.4 Model and library checking 1215
9.4.1 Overview 1215
9.4.2 Regression testing 1216
9.4.3 Class coverage 1223
9.4.4 Condition coverage 1224
9.4.5 Style checking 1225
9.5 Model comparison 1229
9.5.1 Overview 1229
9.5.2 Getting started 1229
9.5.3 Comparison report 1231
9.6 Model structure 1238
9.6.1 Introduction 1238
9.6.2 Traversing models before translation 1238
9.6.3 Interface to semantics not only to syntax 1240
9.6.4 Extracting information and modifying structure before translation 1241
9.6.5 Traversing translated models 1246
10 FMI, eFMI, and SSP Support in Dymola 1249
10.1 FMI Support in Dymola 1250
10.1.1 Introduction 1250
10.1.2 Exporting FMUs from Dymola 1251
10.1.3 Importing FMUs in Dymola 1271
10.1.4 Validating FMUs from Dymola 1289
10.1.5 Profiling of FMUs 1292
10.1.6 FMI Kit for Simulink 1292
10.2 eFMI Support in Dymola 1293
10.3 SSP Support in Dymola 1294
10.3.1 Introduction 1294
10.3.2 SSP Import 1295
10.3.3 SSV Import 1297
10.3.4 SSP Export 1297
11 Simulation Environments 1301
11.1 Introduction 1301
11.2 Dymola – Matlab interface 1302
11.2.1 Using the Dymola-Simulink interface 1302
11.2.2 Other Matlab utilities 1312
11.3 Real-time Simulation 1313
11.3.1 dSPACE systems 1315
11.3.2 Simulink Real-Time 1321
11.3.3 Advanced Options for Real-Time Simulation 1324
11.4 DDE Communication 1328
11.4.1 DDE interface for Dymola 1328
11.4.2 Explorer file type associations 1330
11.4.3 DDE Server support in Dymosim simulator 1330
11.5 OPC Communication 1334

11.6 Code and Model Export 1334
11.6.1 Introduction 1334
11.6.2 Binary Model Export 1336
11.6.3 Source Code Generation 1339
11.6.4 The StandAloneDymosim project 1340
12 Scripting and Reporting 1349
12.1 Introduction 1349
12.2 Java Interface for Dymola 1350
12.3 Python Interface for Dymola 1365
12.4 JavaScript interface for Dymola 1375
12.5 Report generator 1376
12.5.1 Fundamentals 1376
12.5.2 JavaScript functions 1376
12.5.3 Example of HTML report sections 1379
12.5.4 Mouse and keyboard commands available for animation in reports 1382
13 Advanced Modelica Support 1385
13.1 Declaring functions 1385
13.2 User-defined derivatives 1385
13.2.1 Analytic Jacobians 1386
13.2.2 How to declare a derivative 1387
13.3 External functions in other languages 1391
13.3.1 C 1391
13.3.2 Java 1396
13.3.3 C++ 1401
13.3.4 FORTRAN 1401
13.4 Means to control the selection of states 1402
13.4.1 Motivation 1402
13.4.2 The state select attribute 1403
13.4.3 Avoiding dynamic state selection when having only states that can be integrated separately from other states 1404
13.5 Using noEvent 1405
13.5.1 Background: How events are generated 1405
13.5.2 Guarding expressions against evaluation 1405
13.5.3 How to use noEvent to improve performance 1406
13.5.4 Combined example for noEvent 1407
13.5.5 Constructing anti-symmetric expressions 1408
13.5.6 Mixing noEvent and events in one equation 1409
13.6 Equality comparison of real values 1411
13.6.1 Type of variables 1411
13.6.2 Trigger events for equality 1411
13.6.3 Locking when equal 1412
13.6.4 Guarding against division by zero 1412
13.7 Some supported features of the Modelica language 1413
13.7.1 Support for Modelica Language version 3.5 1413
13.7.2 GUI support for removing any existing modification of a parameter by using the keyword break.. 1413
13.7.3 Support for Modelica Language version 3.4 1418
13.7.4 Synchronous Modelica 1418

13.7.5 State Machines ... 1418
13.7.6 Operator overloading ... 1418
13.7.7 Homotopy operator ... 1419
13.7.8 Arrays ... 1420
13.7.9 Enumerations ... 1421
13.7.10 Support of String variables in models ... 1422
13.7.11 Support of inner/outer components ... 1422
13.7.12 Functions as formal input to functions ... 1423
13.7.13 Assert ... 1423
13.7.14 Identifiers starting with underscore and vendor-specific annotations ... 1423
13.7.15 Quoted identifiers containing dot supported ... 1423
13.7.16 Running a function before check/translation/simulation ... 1424
13.7.17 Forcing translation of functions ... 1424
13.7.18 Support for predefined plots ... 1424
13.7.19 Deprecation warnings ... 1424
13.7.20 Licensing ... 1425
13.8 Symbolic Processing of Modelica Models ... 1425
13.8.1 Sorting and algebraic loops ... 1425
13.8.2 Reduction of size and complexity ... 1426
13.8.3 Index reduction ... 1427
13.8.4 Example ... 1429
13.8.5 Alias elimination of parameters ... 1432
13.8.6 Efficient code generation for large tables ... 1432
13.8.7 References ... 1432
13.9 Symbolic solution of nonlinear equations in Dymola ... 1433
13.9.1 Introduction ... 1433
13.9.2 Solving a nonlinear equation with single appearance of the unknown by applying function inverses 1434
13.9.3 Solving a nonlinear equation with special patterns for the unknown ... 1437
13.9.4 Partitioning of a system of equations into a linear and nonlinear (one variable) part ... 1437
13.9.5 Using min and max values to evaluate if-conditions ... 1439
14 Visualize 3D ... 1443
14.1 Introduction ... 1443
14.2 Inserting and removing graphical objects ... 1446
14.3 Basic primitives ... 1456
14.4 Surface Plots ... 1458
14.4.1 Using the generateMeshGrid function to generate parametric surface matrices ... 1458
14.4.2 Three examples using the SurfaceDemo ... 1461
15 User-defined GUI ... 1479
15.1 Building user-defined dialogs ... 1479
15.1.1 Ways of working with annotations ... 1479
15.1.2 Records and dialogs ... 1480
15.2 Extendable user interface – menus, toolbars and favorites ... 1510
15.2.1 Defining content of menus and toolbars ... 1510
15.2.2 Displaying library-specific menus and toolbars in Dymola (commercial library developers) ... 1513
15.2.3 Defining packages with users own collection of favorite models ... 1513
16 Appendix — Migration ... 1517

16.1 Migrating to newer libraries 1517
16.1.1 How to migrate 1517
16.1.2 Basic commands to specify translation 1518
16.1.3 How to build a convert script 1524
16.2 Upgrading to new version of Modelica Standard Library 1526
16.2.1 Introduction 1526
16.2.2 Basics 1526
16.2.3 Upgrading to a new Modelica version 1528
16.2.4 Using old models after upgrading to the latest Modelica version 1532
16.2.5 Determining what libraries a model use 1532
16.2.6 Specifying the version of a package and checking of conversion scripts 1533
16.2.7 Upgrading models and libraries to a new library version 1534
16.3 Preparing libraries for migration 1536
16.4 Updating Modelica annotations 1537
17 Appendix — Installation 1541
17.1 Installation on Windows 1542
17.1.1 Dymola as 32-bit and 64-bit application 1542
17.1.2 Installing the Dymola software 1542
17.1.3 Installing a C compiler 1546
17.1.4 Installing the Dymola license file 1552
17.1.5 Additional setup 1556
17.1.6 Changing the setup of Dymola 1578
17.1.7 Removing Dymola 1579
17.1.8 Getting information about the latest Dymola release 1579
17.1.9 Installing updates 1580
17.2 Installation on Linux 1581
17.2.1 Installing Dymola 1582
17.2.2 Additional setup 1582
17.2.3 Removing Dymola 1584
17.3 Dymola License Servers on Windows 1585
17.3.1 FLEXnet Publisher license server 1585
17.3.2 Dassault Systèmes License Server (DSLS) 1596
17.4 Dymola License Server on Linux 1597
17.4.1 FLEXnet Publisher license server 1597
17.4.2 Dassault Systèmes License Server (DSLS) 1598
17.5 Utility programs 1600
17.5.1 Obtaining a host id 1600
17.6 System requirements 1601
17.6.1 Hardware requirements 1601
17.6.2 Hardware recommendations 1601
17.6.3 Software requirements 1601
17.6.4 Dymola license server 1604
17.7 License requirements 1606
17.7.1 General 1606
17.7.2 License for Dymola – Simulink interface 1606
17.7.3 License for real-time simulation 1606
17.7.4 Licenses for exporting code 1607

17.7.5 Licenses for executing/importing/using code 1608
17.7.6 Licenses for libraries in the Dymola library menu 1609
17.7.7 Licenses for libraries not in Dymola library menu 1611
17.8 Troubleshooting 1611
17.8.1 License file 1611
17.8.2 Compiler problems 1613
17.8.3 Simulink 1616
17.8.4 Change of language 1617
17.8.5 Other Windows-related problems 1617
18 Index 1619

1 WHAT IS DYMOLA?

1 What is Dymola?

1.1 Features of Dymola

Dymola – Dynamic Modeling Laboratory – is suitable for modeling of various kinds of physical systems. It supports hierarchical model composition, libraries of truly reusable components, connectors and composite acausal connections. Model libraries are available in many engineering domains.

Modelica is the modeling language; Dymola is the tool.

Employing the Modelica language, Dymola uses a modeling methodology based on object orientation and equations. The usual need for manual conversion of equations to a block diagram is removed by the use of automatic formula manipulation, leading to greater robustness and easier model building. Other highlights of Dymola are:

- Handles large, complex multi-engineering models. Uses symbolic methods to generate faster simulation code.
- Faster modeling by graphical model composition (drag-and-drop).
- Model libraries for several application domains, but also open for user defined model components.
- Open interface to other programs, including Python, Java and JavaScript.
- Code export using Functional Mockup Interface (FMI) or for real-time simulation.

Dymola supports the entire model development process from model creation to simulation and result analysis.

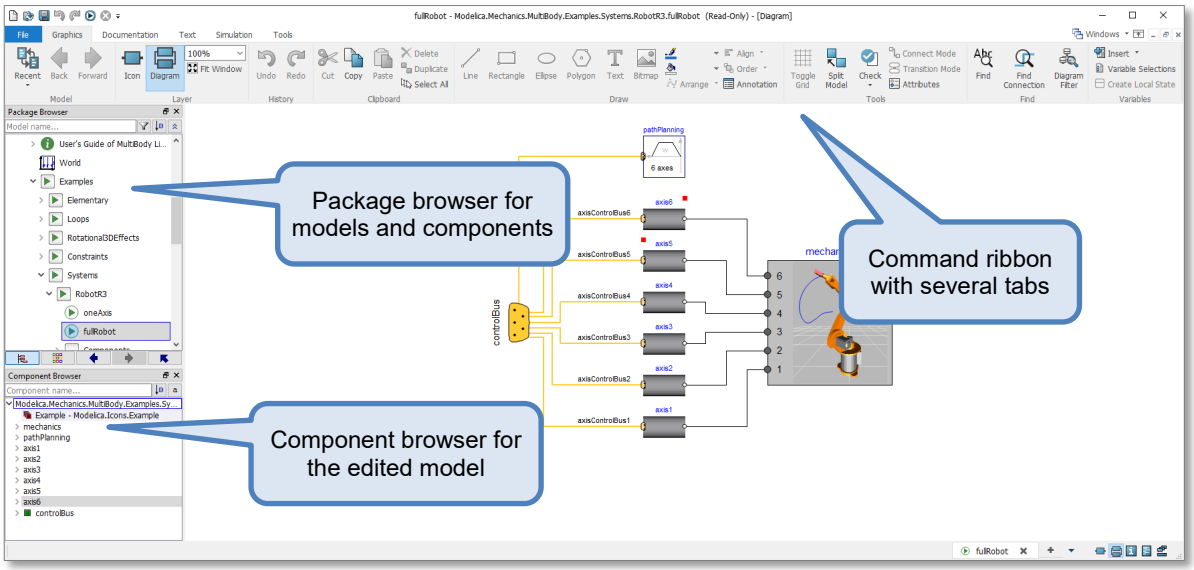
- Graphical and textual editors are used to create models and model components, typically by using component models from existing model libraries.
- The Simulation tab is used to make experiments on the model, plot results and animate the behavior. It also has a scripting window for automation of experimentation and performing calculations.

The sections below will give a short hands-on introduction to the basic features of Dymola.

1.2 Simulating an existing model

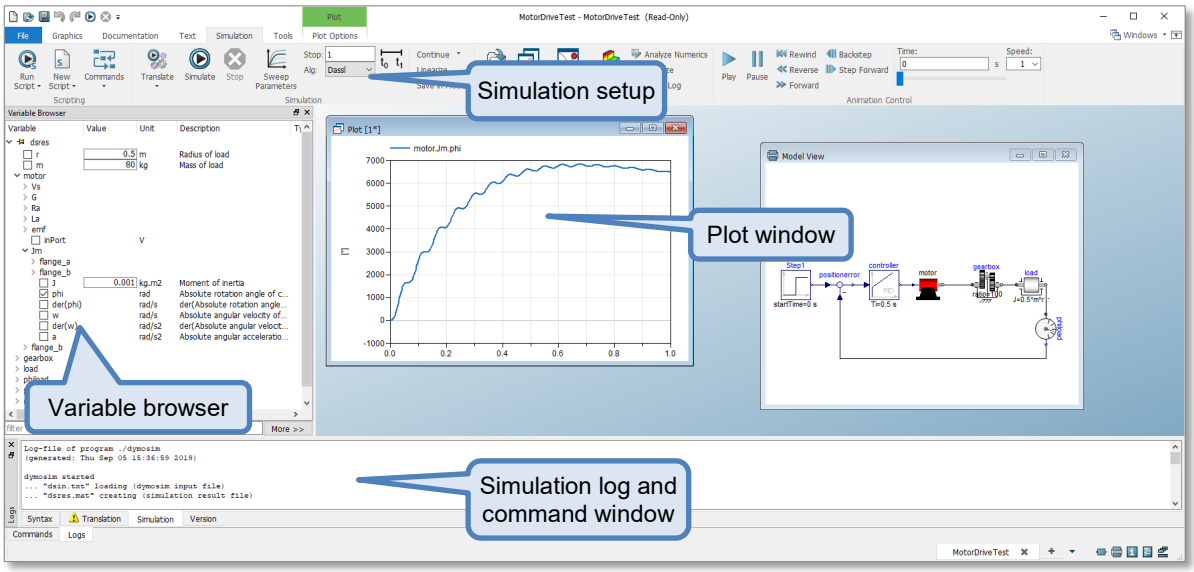
Finding the model

Dymola starts in the graphical editor. The model to simulate is found by using the **File > Open...** or **File > Demos** commands. Example models are also found in the Modelica Standard Library or in other libraries opened by the **File > Libraries** command.

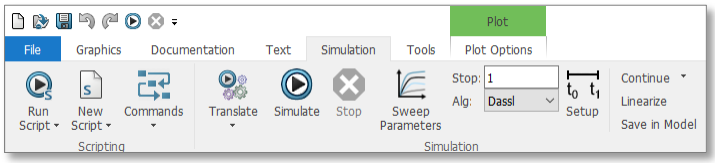


Simulation

The Simulation tab is used for experimentation. It has a simulation setup to define duration of simulation, etc., plot windows, animation windows and a variable browser. The model view can also be used for navigating the hierarch of the model.

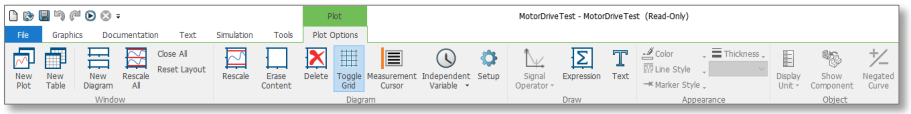


The command ribbon has controls to run scripts and pre-defined commands, as well as to setup and to run the simulation. The most common settings, the simulation stop time and the numerical integration algorithm are available in the ribbon; additional setting by clicking **Setup**.

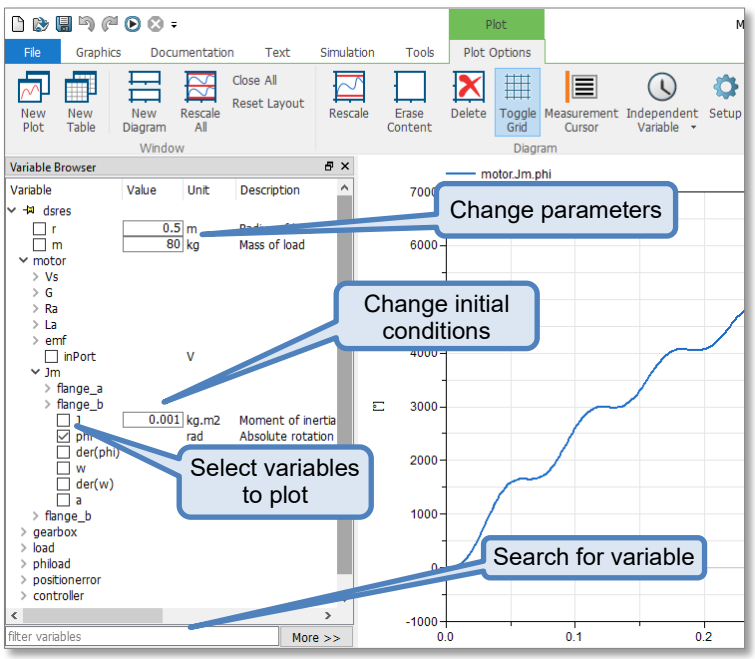


Experimentation and plotting

Additional controls are shown when the plot window is active. They are used to setup new plot windows, or the layout of what is plotted in the window.



The variable browser allows selecting plot variables, changing parameters and initial conditions.

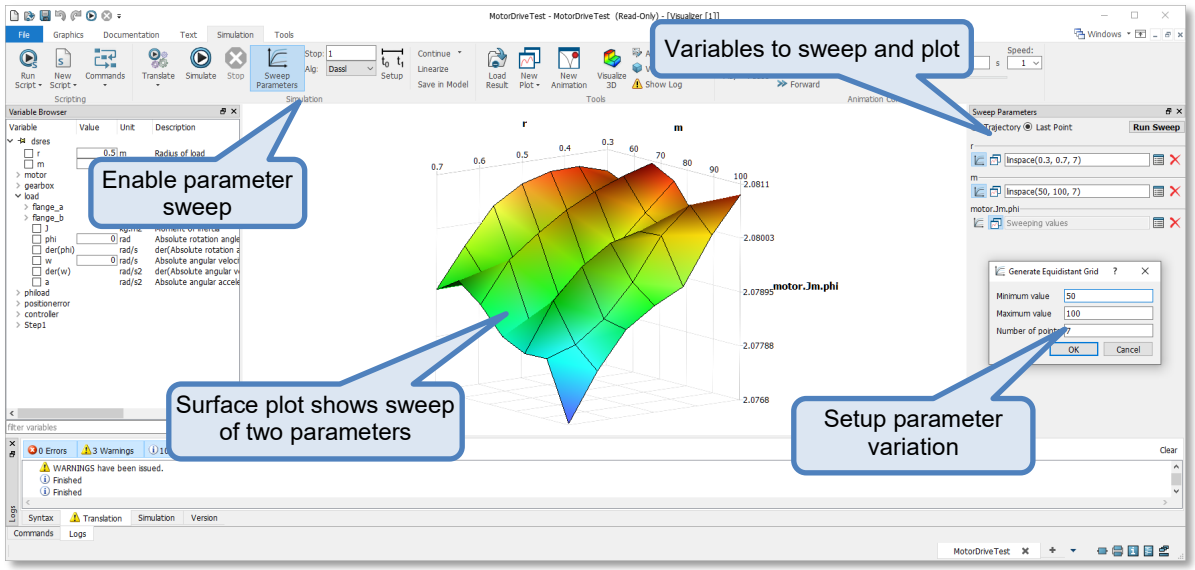


An animation window shows a 3D view of the simulated model. The animation can be run at different speeds, halted, single stepped and run backwards.

Sweeping parameters

Model experimentation involves running simulations for various combinations of parameters to determine the modeled system's properties. To increase performance, parameter sweeps are run on multiple CPU cores when possible.

A convenient user interface facilitates such experimentation. Sweeps over one or two parameters are visualized by plotting variables used as key performance indicators, or the surface formed by varying two parameters.

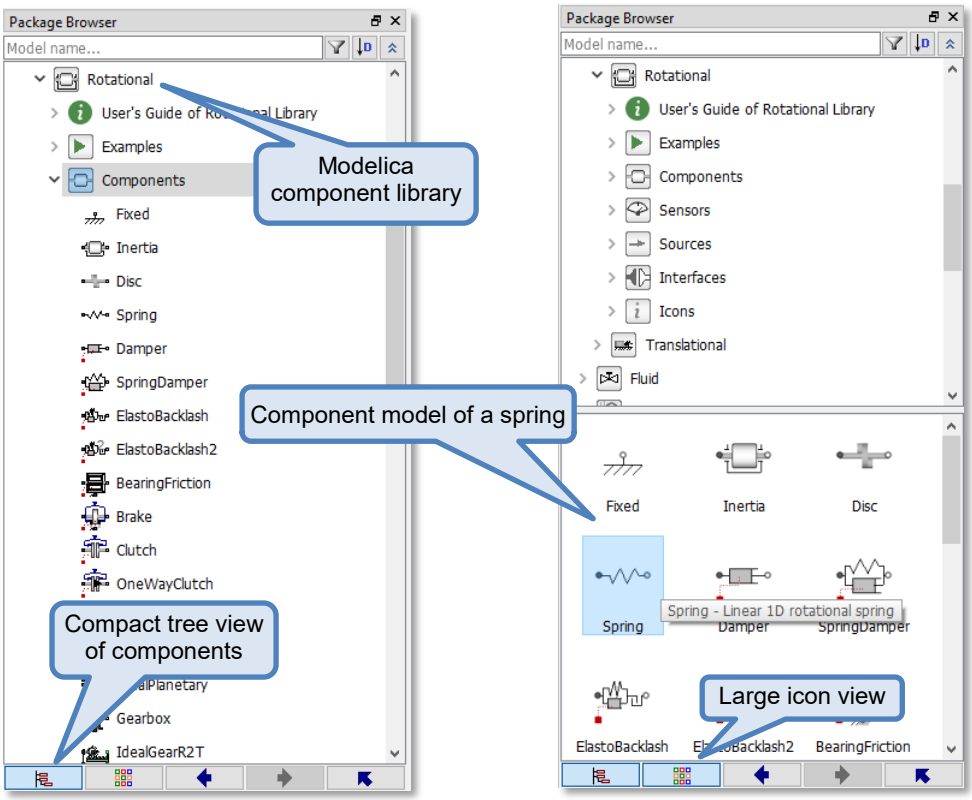


1.3 Building a model

The graphical model editor is used for creating and editing models in Dymola. Structural properties, such as components, connectors and connections are edited graphically, while equations and declarations are edited with a built-in text editor.

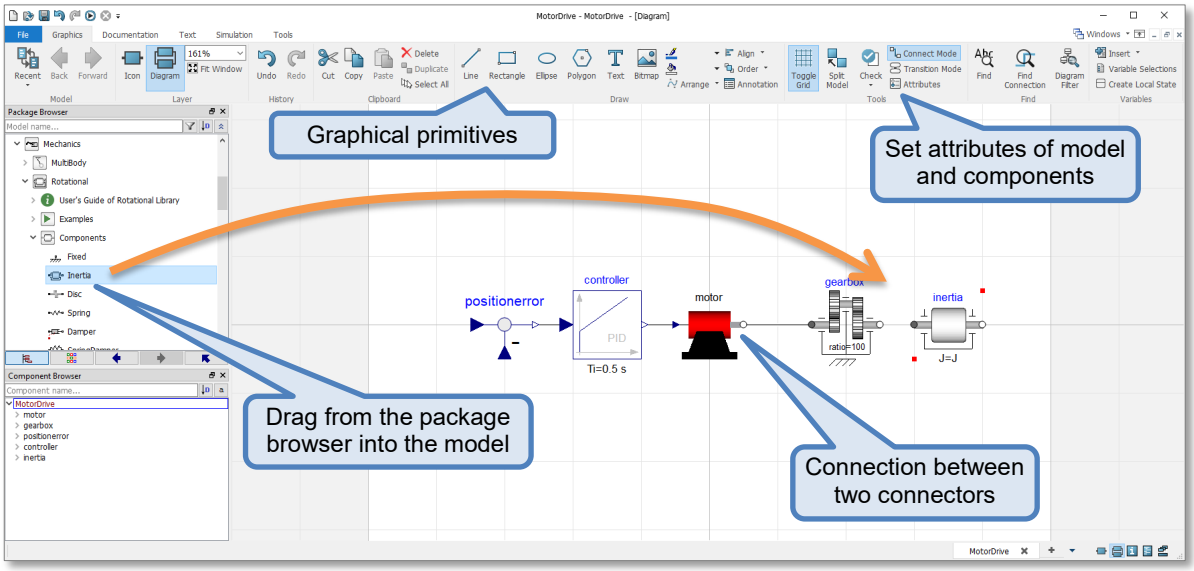
Finding a component model

The package browser allows viewing and selecting component models from a list of models with small icons. It is also possible to get a large icon view.



Composing a new model

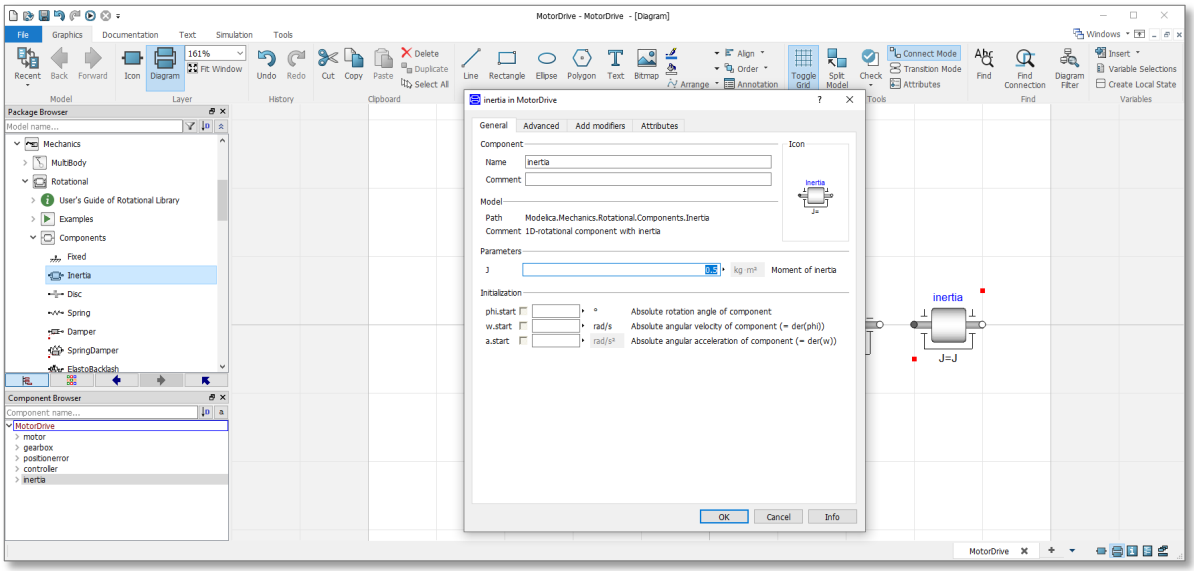
Components are dragged from the package browser to the diagram layer and connected. The component hierarchy is shown in the component browser.



A connection is made by clicking on one connector and drawing to the other connector.

Setting component parameters

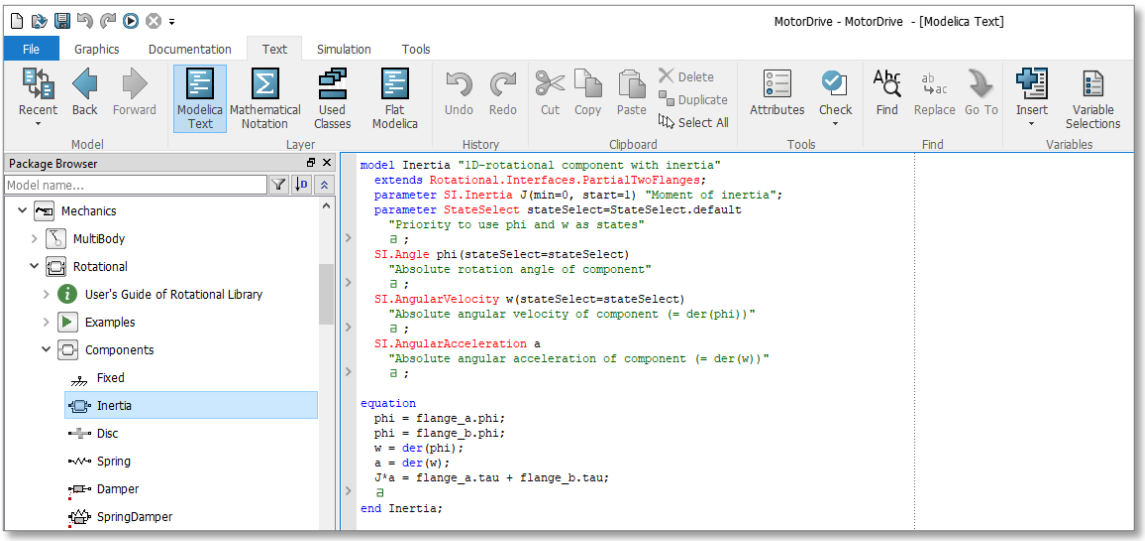
By double-clicking on a component, or right-clicking the component and selecting **Parameters**, a dialog for giving the name of the component and the values of its parameters is shown.



Editing Modelica text

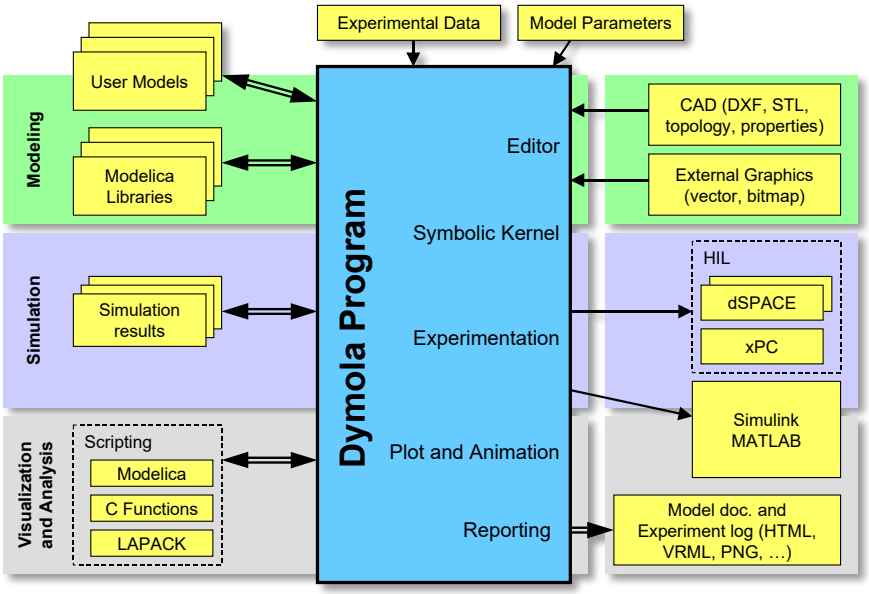
Clicking on the **Documentation** or **Text** tabs allows for inspection of the documentation or the model itself or the underlying Modelica code. This is also the editor for entering Modelica code, i.e. declarations and equations for low-level models.

In each tab there are different views, for example **Text** is used both to edit the Modelica text, and to view the equations in mathematical notation, etc.



1.4 Architecture of Dymola

The overall architecture of the Dymola program is shown below. Dymola has a powerful graphic editor for composing models, using Modelica models stored on files. Dymola can also import other data and graphics files. Dymola contains a translator for Modelica equations generating C code for simulation. The code can also be exported using the Functional Mockup Interface (FMI), or to Matlab/Simulink and hardware-in-the-loop platforms.



1.5 Features of Modelica

Modelica is an object-oriented language for modeling of large, complex and heterogeneous physical systems. It is suited for multi-domain modeling, for example for modeling of mechatronic systems within automotive, aerospace and robotics applications. Such systems are composed of mechanical, electrical and hydraulic subsystems, as well as control systems.

General equations are used for modeling of the physical phenomena. The language has been designed to allow tools to generate efficient code automatically. The modeling effort is thus reduced considerably since model components can be reused, and tedious and error-prone manual manipulations are not needed.

Background

Modeling and simulation are becoming more important since engineers need to analyze increasingly complex systems composed of components from different domains. Current tools are generally weak in treating multi-domain models because the general tools are block-oriented and thus demand a huge amount of manual rewriting to get the equations into explicit form. The domain-specific tools, such as circuit simulators or multibody programs, cannot handle components of other domains in a reasonable way.

There is traditionally too large a gap between the user's problem and the model description that the simulation program understands. Modeling should be much closer to the way an engineer builds a real system, first trying to find standard components like motors, pumps and valves from manufacturers' catalogues with appropriate specifications and interfaces.

Equations and reuse

Equations facilitate true model reuse.

Equations are used in Modelica for modeling of the physical phenomena. No particular variable needs to be solved for manually because Dymola has enough information to decide that automatically. This is an important property of Dymola to enable handling of large models having more than hundred thousand equations. Modelica supports several formalisms: ordinary differential equations (ODE), differential-algebraic equations (DAE), bond graphs, finite state automata, Petri nets etc.

The language has been designed to allow tools to generate very efficient code. Modelica models are used, for example, in Hardware-in-the-Loop (HIL) simulation of automatic gearboxes, which have variable structure models. Such models have so far usually been treated by hand, modeling each mode of operation separately. In Modelica, component models are used for shafts, clutches, brakes, gear wheels etc. and Dymola can find the different modes of operation automatically. The modeling effort is considerably reduced since model components can be reused and tedious and error-prone manual manipulations are not needed.

Interfaces and inheritance

Modelica has been designed to provide very strong support for expressing common properties of models and interfaces that can be shared throughout an entire engineering domain. The Modelica Standard Library defines many such interfaces that are shared by commercial and third party libraries.

Inheritance is a composition mechanism inspired by object-oriented programming languages. It gives an efficient and safe construct to share common properties, as shown in this definition of an electrical capacitor.

```
connector Pin
  Voltage v;
  flow Current i;   // Sums to zero
end Pin;
          partial model TwoPin
            Pin p, n;   Voltage v;
          equation
            v = p.v - n.v;
            p.i + n.i = 0;
          end TwoPin;
                    model Capacitor
                      extends TwoPin;
                      parameter Capacitance C;
                    equation
                      i = C*der(v);
                    end Capacitor;
```

Modelica history

Reuse is a key issue for handling complexity. There had been several attempts to define object-oriented languages for physical modeling. However, the ability to reuse and exchange models relies on a standardized format. It was thus important to bring this expertise together to unify concepts and notations.

A design group was formed in September 1996 and one year later, the first version of the Modelica language was available (http://www.Modelica.org). It has been designed by a group of more than 50 experts with previous knowledge of modeling languages and differential-algebraic equation models.

A paper by Hilding Elmqvist: "Modelica Evolution – From My Perspective" is available by the command **Help > Documentation**. The paper describes the history of Modelica and Dymola from the author´s perspective.

1.6 Functional Mockup Interface



The Functional Mockup Interface (FMI) is an industry standard for combining simulation code modules (FMUs) from multiple tools and vendors. Developed under the auspices of the Modelica Association, the specification provides a well-defined and vendor-independent exchange format for code (binary or source) as well as associated data and documentation.

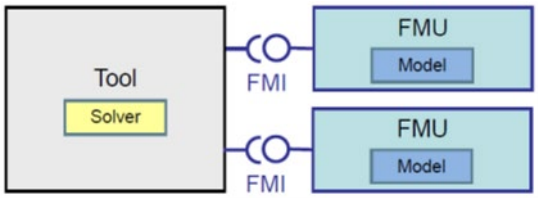
FMI is supported by a large number of authoring tools, including tools which are not Modelica based, making it the ideal foundation for a vendor independent simulation infrastructure.

The FMI 2.0 specification standardizes important extensions that improve simulation speed and accuracy.

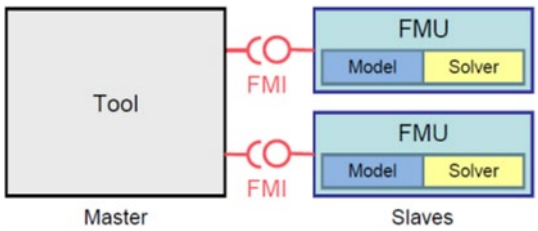
- Classification of interface variables
- Save and restore FMI module state
- Variable dependency information
- Partial derivatives
- Precise time event handling
- Improved unit definitions

Model exchange and Co-simulation

The FMI specification defines two exchange formats. FMI for model exchange defines the interface for simulation code modules that must be combined with a common, central, solver. This ensures a uniform numeric solution and centralized strict simulation error control.



FMI for co-simulation defines the interface for code modules with embedded numeric solvers, as used by the generating tool. This approach gives the opportunity to embed dedicated solvers for the modeled application, and facilitates compatibility with simulation in the authoring tool.



FMI tools for Simulink – FMI Kit

Free download available.

Dassault Systèmes provides tools with full support for FMU export and import with Simulink. The toolkit can be used free without any license key. Support is available through the download site on GitHub.

https://github.com/CATIA-Systems/FMIKit-Simulink/releases

2 GETTING STARTING WITH DYMOLA

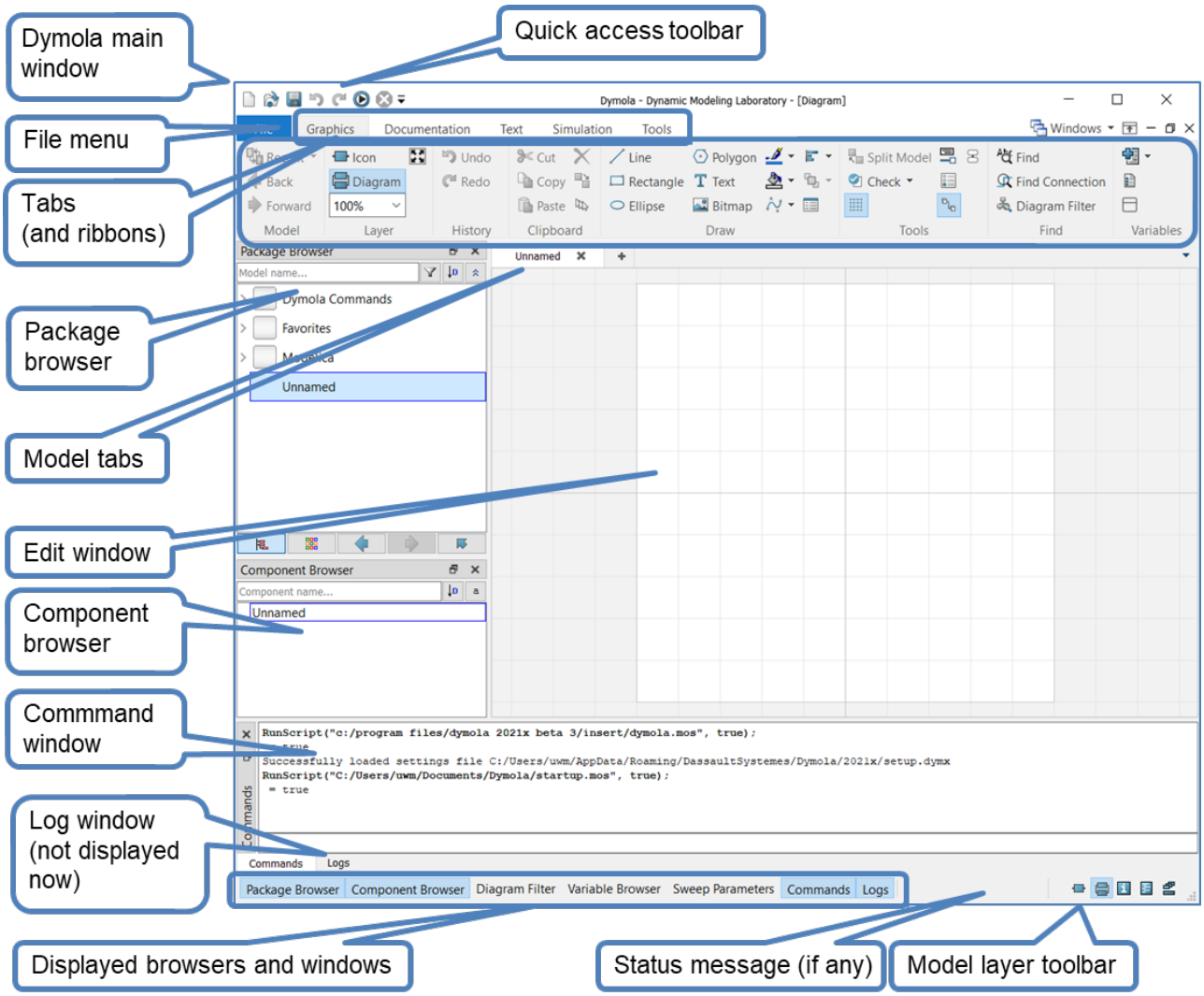
2 Getting started with Dymola

2.1 Introduction

This chapter will take you through some examples in order to get you started with Dymola. For detailed information about the program, you are referred to the on-line documentation and the user's manuals. The on-line documentation is available in the **Tools** tab, in the **Help** group, by clicking **Help Documents**. The tool tips and the **What's this**? features are fast and convenient ways to access information. Please see section "Help and information" on page 136 for more information.

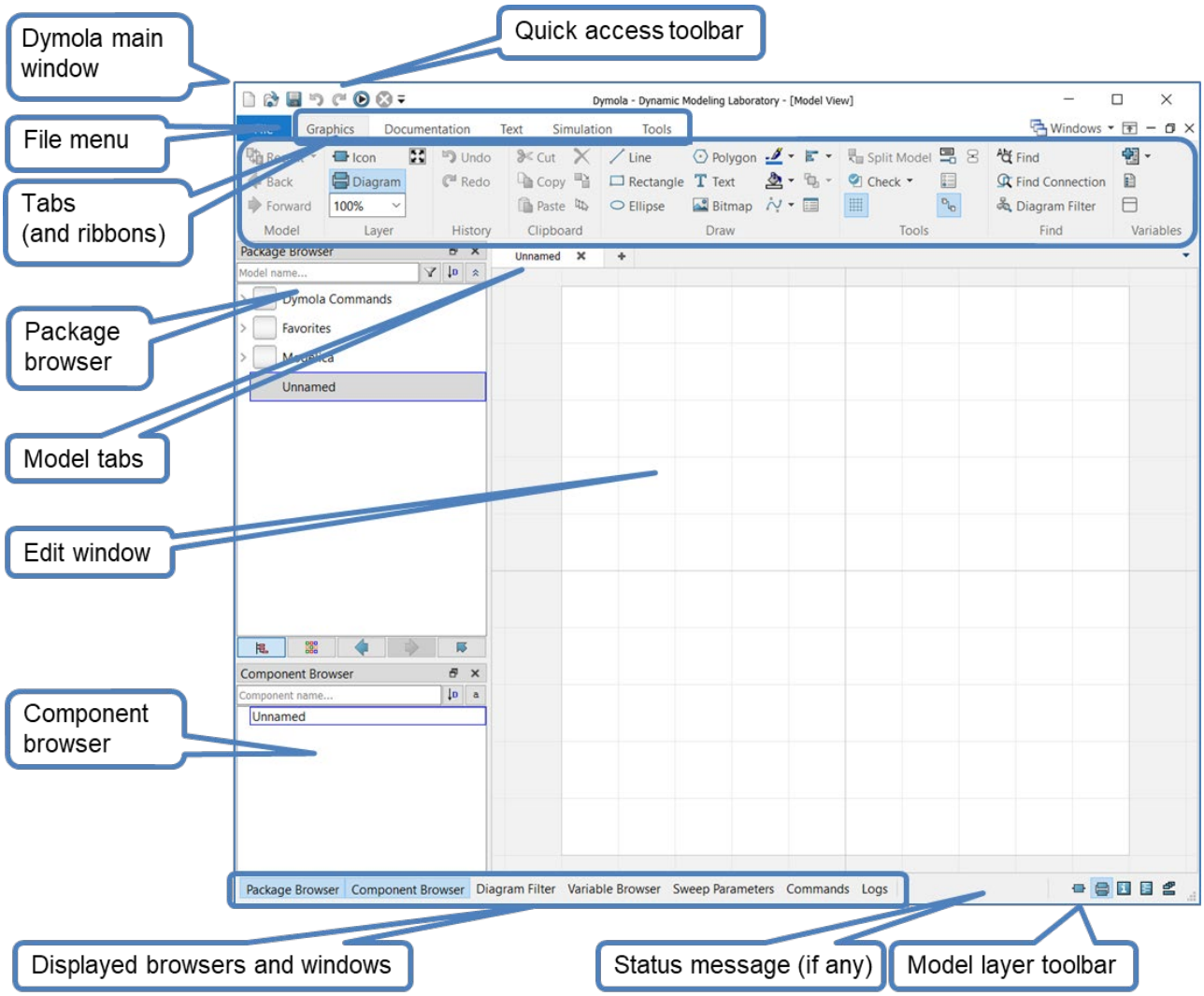
Start Dymola. The main Dymola window appears.

The operations, tool buttons available and types of sub-windows appearing depend on the mode and the user's choice. Dymola starts with a useful default configuration, but allows customizing:



One customizing we do immediately, to get more space for the model, is to click, in the last line in the window, **Commands** and **Logs**, to not display the command window and log window – we will come back to them later.

Doing this, we get:


Dymola main window
Quick access toolbar
File menu
Tabs (and ribbons)
Package browser
Model tabs
Edit window
Component browser
Displayed browsers and windows
Status message (if any)
Model layer toolbar
Dymola - Dynamic Modeling Laboratory - [Model View]
Graphics
Documentation
Text
Simulation
Tools
Windows
Icon
Diagram
Back
Forward
100%
Undo
Redo
Cut
Copy
Paste
Line
Rectangle
Ellipse
Polygon
Text
Bitmap
Split Model
Check
Find
Find Connection
Diagram Filter
Model
Layer
History
Clipboard
Draw
Tools
Find
Variables
Package Browser
Model name...
Dymola Commands
Favorites
Unnamed
Component Browser
Component name...
Unnamed
Package Browser
Component Browser
Diagram Filter
Variable Browser
Sweep Parameters
Commands
Logs

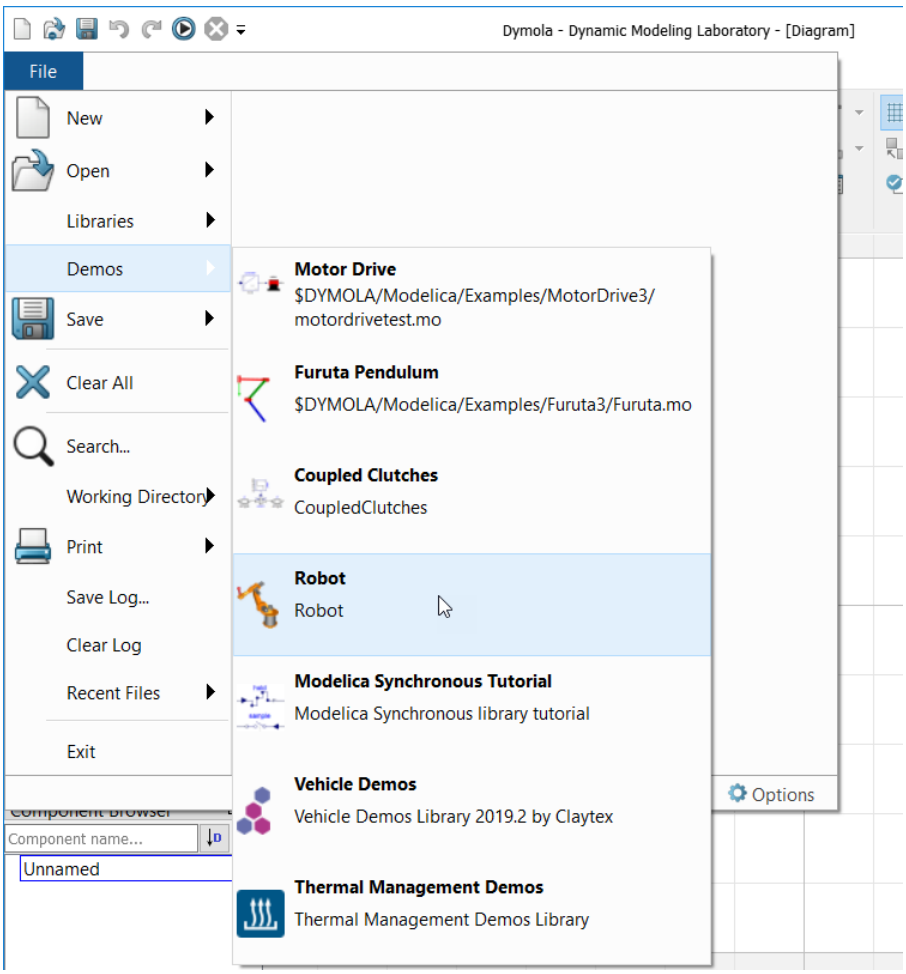
2.2 Simulating a model — industrial robot

This first example will show how to browse an existing model, simulate it, and look at the results. If you want to learn the basics first, you can skip to a smaller example in the next section 2.3 “Solving a non-linear differential equation” on page 52.

2.2.1 Investigating the robot before simulating it

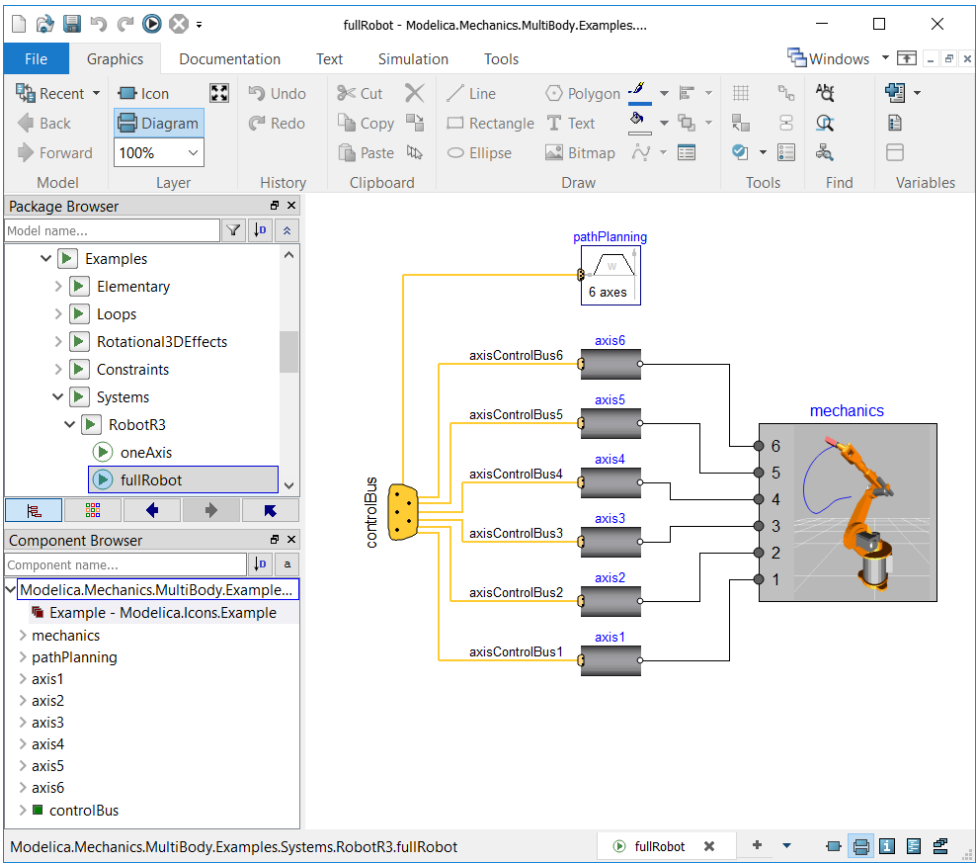
We will study a model of an industrial robot. To view the industrial robot model, use the menu **File > Demos** and select **Robot**.

Opening a demo example.



Dymola starts loading the model libraries needed for the robot model and displays it. The following is displayed:

The robot demo.



(To get more space for the model, you can, in the upper corner of the window, click on the button **Minimize Ribbons** until only the commands on top are shown.)

The package browser in the upper left sub-window displays the package hierarchy and it is now opened up with the robot model selected and highlighted. The model diagram in the edit window (the sub-window to the right) shows the top-level structure of the model. The model diagram has an icon for the model of the robot with connected drive lines for the joints. The reference angles to the drive lines are calculated by a path planning module giving the fastest kinematic movement under given constraints.

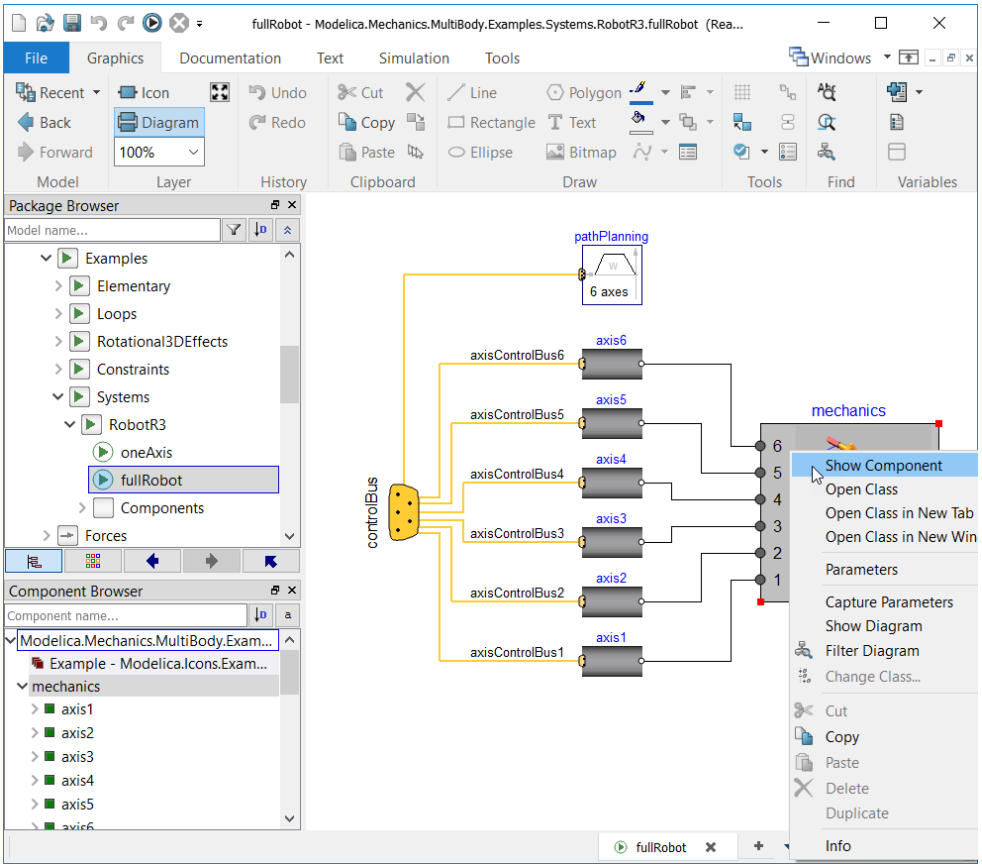
The edit window displays by default the model diagram ("diagram layer") but the user can select other information (other layers) to be displayed instead, e.g. documentation or Modelica text. For more information, please see the chapter "Developing a model".

The component browser in the lower left sub-window also shows the components of the robot experiment in a tree structured view.

To inspect the robot model, select the icon in the edit window (red handles appear, see below) and right-click (press the button to the right on the mouse). A menu pops that contains a

selection of actions that can be made for the selected object (a context menu). From the context menu, select **Show Component**.

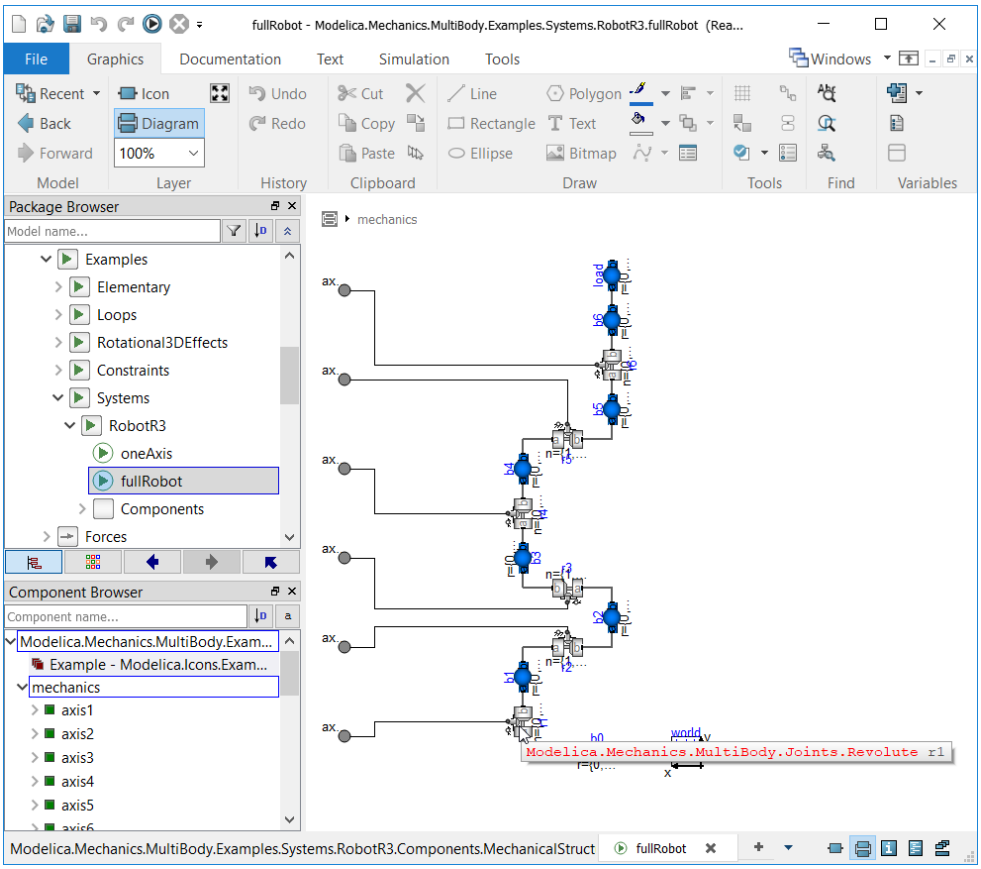
About to view the mechanical structure of the robot.



It is not necessary to select the robot component explicitly by first clicking with the left button on the mouse on it to access its menu. It is sufficient to just have the cursor on its icon in the edit window and right-click. The component browser also gives easy access to the robot component. Just position the cursor over “mechanics” and right-click to get the context menu for “mechanics”. The component browser provides a tree representation of the component structure. The edit window showing the diagram layer and the component browser are synchronized to give a consistent view. When you select a component in the edit window, it is also highlighted in the component browser and vice versa. The diagram layer of the edit window gives the component structure of one component, while the component browser gives a more global view; useful to avoid getting lost in the hierarchical component structure.

The edit window now displays the mechanical structure consisting of connected joints and masses. The component browser is opened up to also show the internals of the mechanics model component.

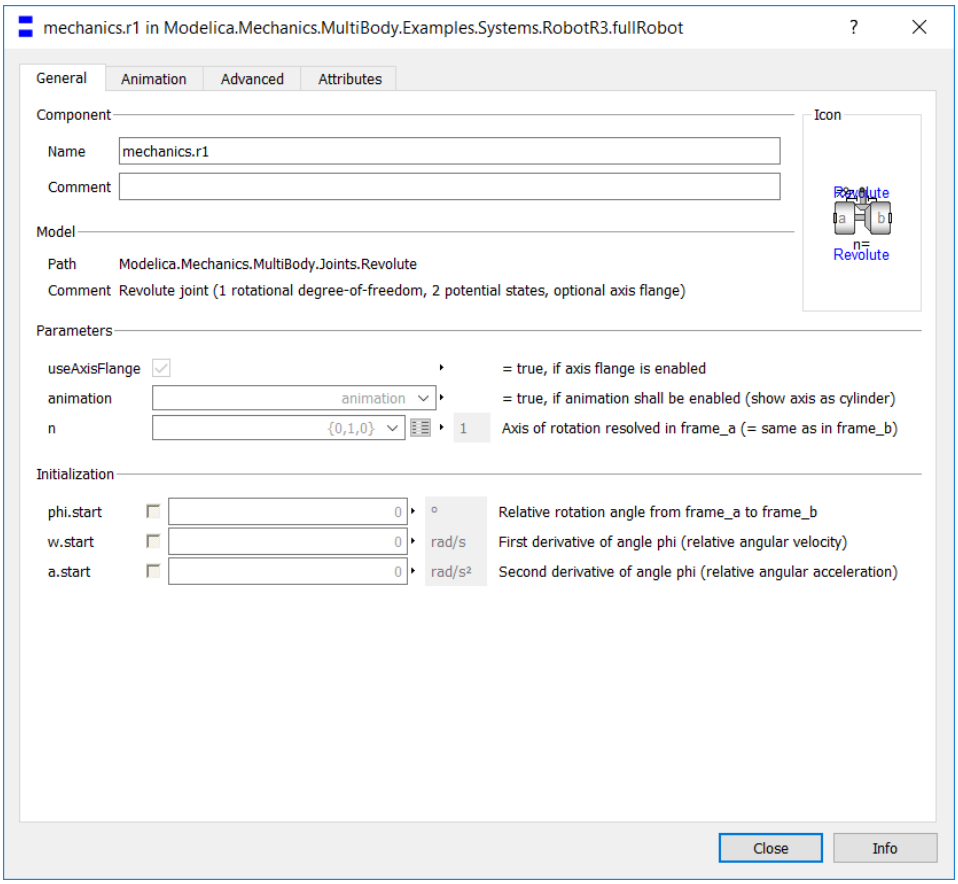
The mechanical structure of the robot.



Double-click on, for example, r1 at the bottom of the edit window (the cursor is resting over it in the figure above). This is a revolute joint. The parameter dialog of that component appears. The parameter dialog can also be accessed from the right button menu[1].

[1] You can define if double-clicking a component should display the parameter dialog (default) or the subcomponents inside it (other actions are also available). If any other action than opening the parameter dialog is defined on double-click, you can open the parameter dialog by **Shift+double-click** on the component.

Parameter dialog.



The parameter dialog allows you to inspect and change actual parameter values. In demos the parameter values are write-protected to avoid unintentional changes of the demo example – then the dialog just has a **Close** button (and an **Info** button). When the parameter values can be changed there is one **OK** button and one **Cancel** button to choose between. The values are dimmed to indicate they are not given at the top-level model, but somewhere down in the component hierarchy.

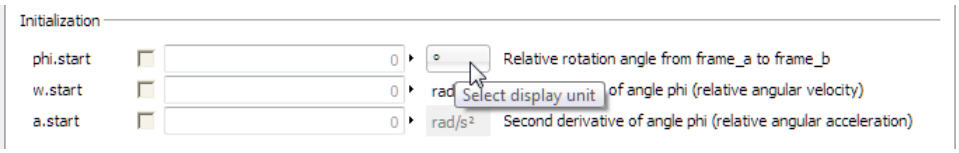
To be able to see how to do changes in this parameter dialog, you must create a demo that is not write-protected. To do so, right-click **fullRobot** in the package browser, select **New > Duplicate Class…**, change the name **fullRobot** to **MyfullRobot** and click **OK**. This will create a new model MyfullRobot that is not write-protected. You can now display the parameter dialog of r1 the same way as you did with the write-protected demo. You will get the image above; you recognize the MyfullRobot name.

A parameter dialog may have several tabs. This dialog has the tabs: **General**, **Animation**, **Advanced**, **Add modifiers**, and **Attributes**. In a tab the parameters can be further structured into groups as shown. It is easy for a model developer to customize the parameter dialogs. (More information about customization can be found in the chapter “Developing a model”,

section “Advanced model editing”, sub-section “Parameters, variables and constants”). Graphical illustrations can be included to show meaning of parameters.

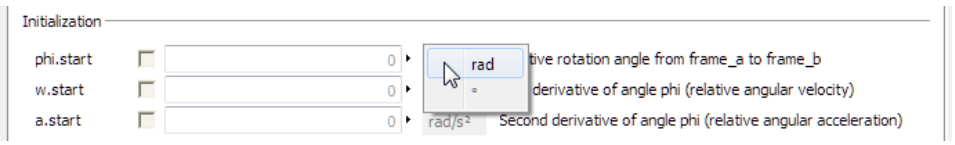
If prepared for, display units can be selected in the dialog. Units that have alternatives are marked by white background (here phi.start and w.start have selectable display units). By resting the cursor over such a unit a button is displayed,

Selectable display unit.



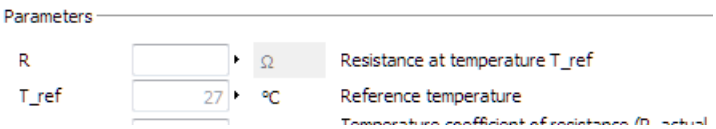
and by clicking on that the selection can be made:

Alternatives of selectable display unit.



It is useful to know that you can enter values with prefixed units, also if that prefix is currently not selectable from the display unit dialog.

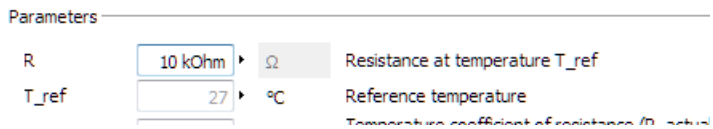
As an example, consider part of the parameter dialog of an ordinary resistor (this example is just a deviation to clarify, we will then continue with the present parameter dialog).



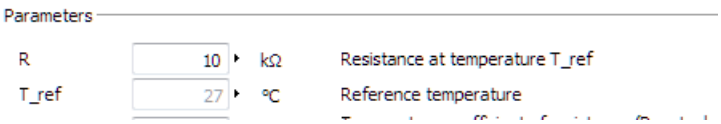
The gray background of Ω indicates no selection of display units can be made. However, if you type in any of the following in the input field to input 10 kiloOhm:

- 10 kOhm
- 10kOhm
- 10 k
- 10k

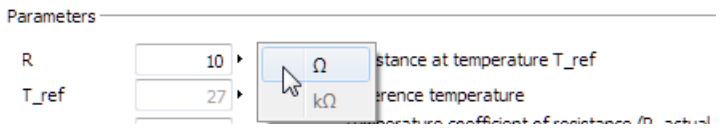
The below shows entering the first alternative:



And click **OK** and display the parameter dialog again, the result is:



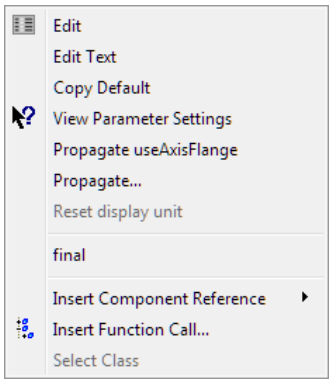
The display unit selection has also been updated:



The display unit selection for Ohm will be updated for all display unit selections of Ohm.

To define display units in general, see “displayUnit” in the index in the end of the manual.

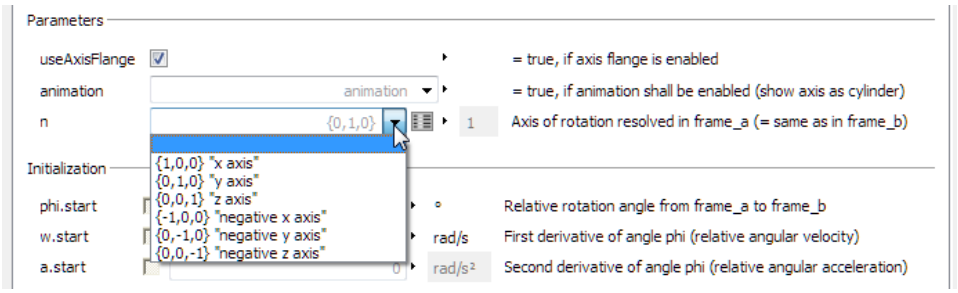
Going back to the present parameter dialog, next to each parameter field is a triangle, this gives you a set of choices for editing the parameters.



Edit gives a matrix editor/function call editor; **Edit Text** gives a larger input field, etc. An example is that for the first parameter useAxisFlange the command **Edit Text** can be used to enter an expression (that should be true in order for the axis flange to be enabled). If such an expression is entered, the checkbox will be displayed as .

Some parameters have a list of choices where you can select values instead of writing them. One example is the parameter n, which defines the axis of rotation. The value for this revolute joint is {0, 1, 0}, i.e. the axis of rotation is vertical.

Choices for n.

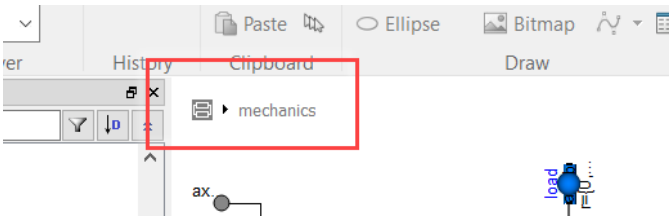




To learn more about the component, select **Info**. An information browser is opened to show the documentation of the revolute joint. Links in the document makes it easy to navigate to e.g. the description of the package containing the revolute joint. Now please close the browser and press **Cancel** in the parameter dialog to leave it.

(If you want to see the documentation without going via the parameter dialog, right-click on the component in the diagram and select **Info**.)

Let us now inspect the drive train model. There are several possible ways to get it displayed. One way is to go back the full robot model, right-click an axis and select **Show Component**. To first get back, you can use the navigation path to the opened component, displayed in top of the diagram:

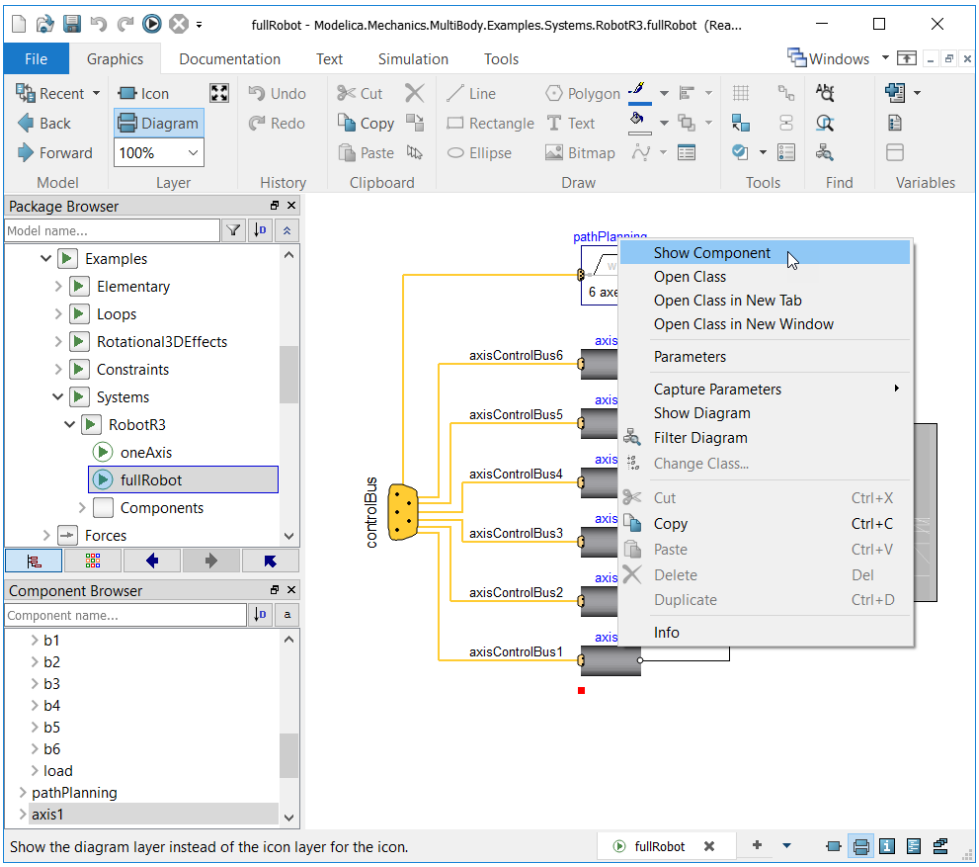


Here you can click the first symbol, that is the symbol of the first level of the hierarchy. As an alternative, you can use the **Graphics > Back** command 

(To go back, in this case, you can open the original demo again by the command **File > Demos > Robot**. The following figures are from the original demo.)

Please, note that robot.mechanics also has components axis1, ..., axis6, but those are just connectors. You must inspect for example robot.axis1 (see figure below).

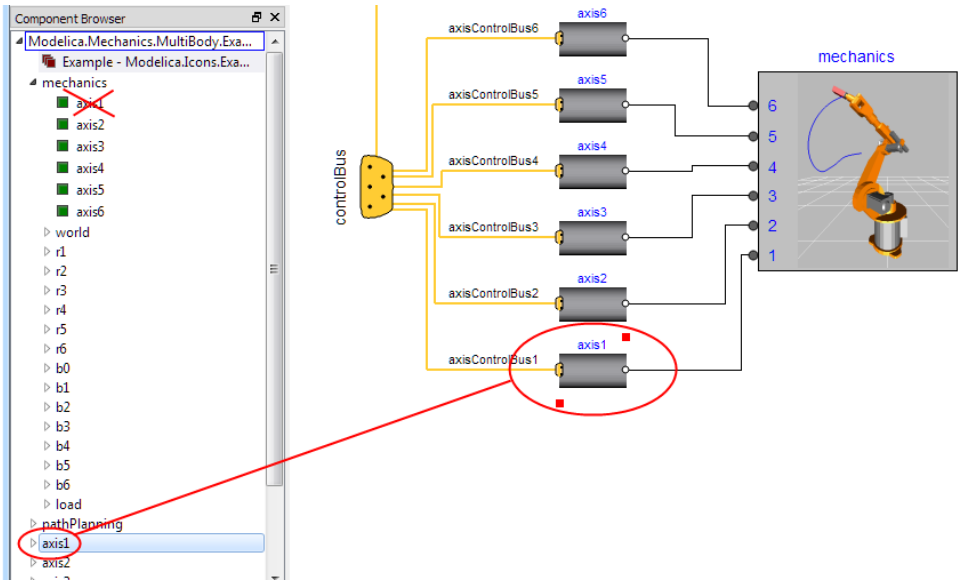
Displaying the components of axis 1.



Another convenient way is to use the component browser. Put the cursor on top of the wanted component in the browser and right-click to get the context menu. Select **Show Component** as shown in the figure below. (In this case also care must be taken not to select any axes inside the module mechanics. The component browser has been enlarged in the figure below to illustrate this.) Instead of using the context menu, you can also double-click the component in the component browser. Please recall that double-clicking on a component in the edit window by default pops up the parameter dialog[2]

[2] If double-clicking a component in the edit window brings up the parameter dialog, you can use **Shift+double-click** on the component in the edit window to display the subcomponents.

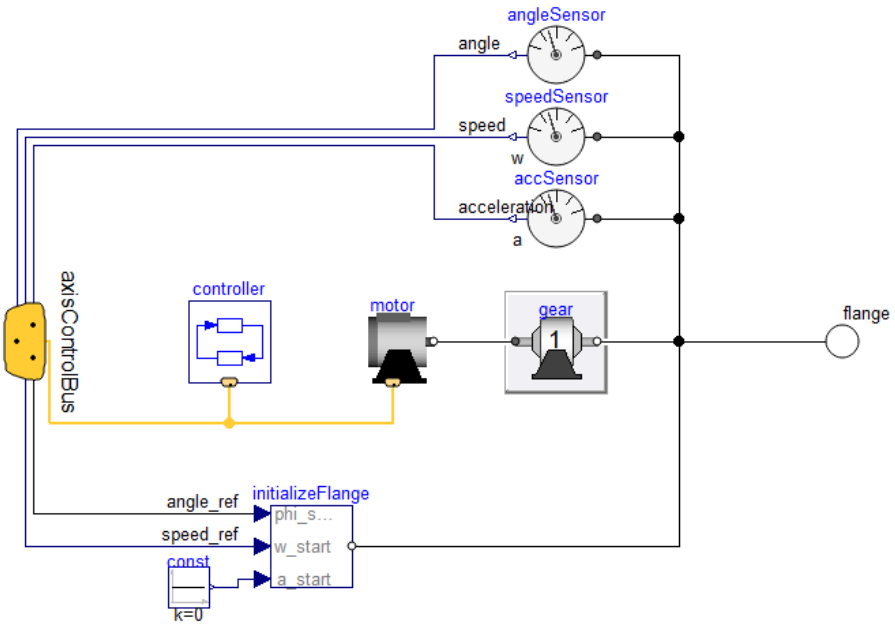
Displaying the components of axis 1.



Whatever method is used; the result will be the following figure in the edit window:

The robot drive train in Axis 1.





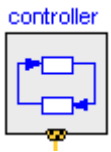
The drive train includes a controller. A data bus is used to send measurements and reference signals to the controller and control signals from the controller to the actuator. The bus for one axis has the following signals:

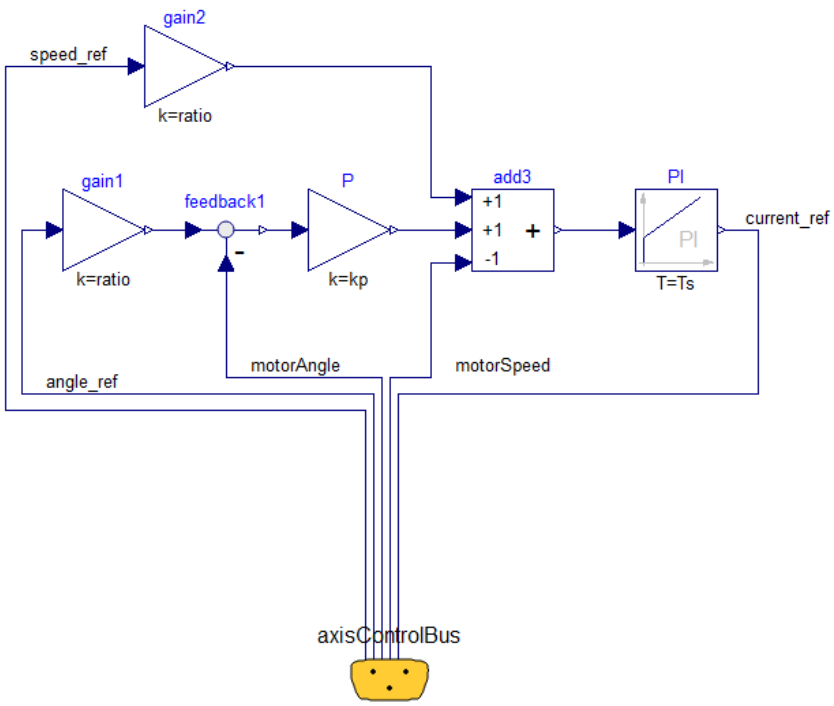
Name	Description
motion_ref	true, if reference motion is not in rest
angle_ref	reference angle of axis flange
angle	angle of axis flange
speed_ref	reference speed of axis flange
speed	speed of axis flange
acceleration_ref	reference acceleration of axis flange
acceleration	acceleration of axis flange
current_ref	reference current of motor
current	current of motor
motorAngle	angle of motor flange
motorSpeed	speed of motor flange

The bus from the path planning module is built as an array having 6 elements of the bus for an axis.

The figure below is displayed by selecting the controller and showing the components of it.

The robot controller in the robot drive train.

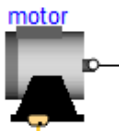


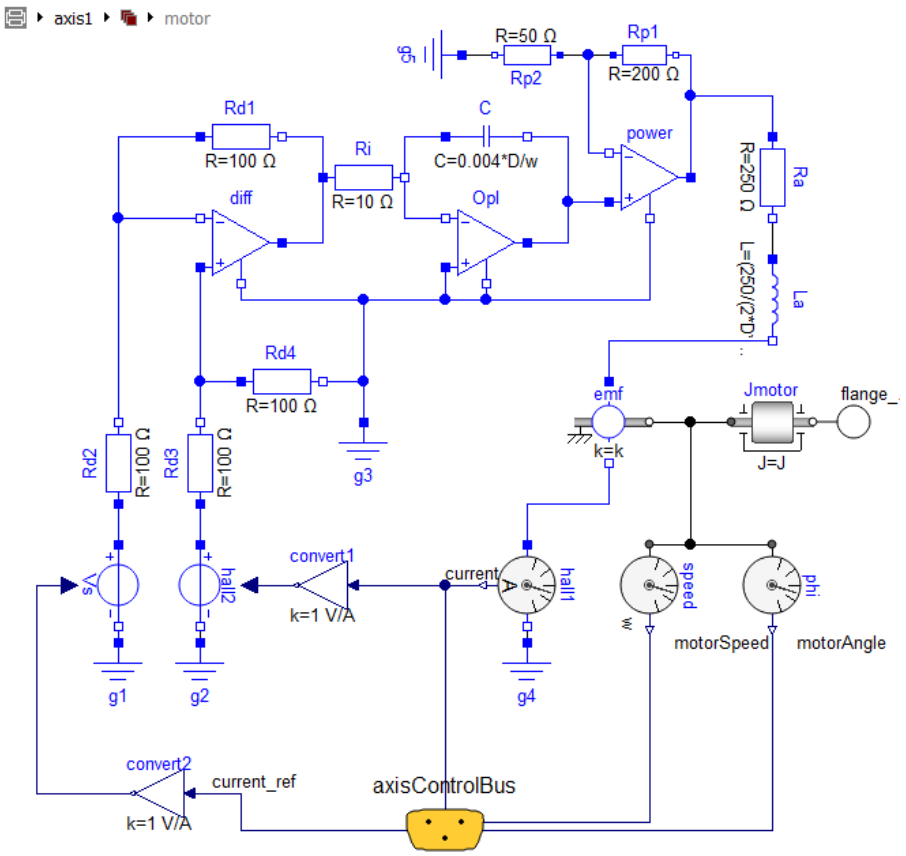


The controller of an axis gets references for the angular position and speed from the path planning module as well as measurements of the actual values of them. The controller outputs a reference for the current of the motor, which drives the gearbox.

The motor model consists of the electromotorical force, three operational amplifiers, resistors, inductors, and sensors for the feedback loop.

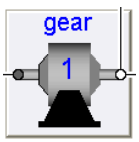
The robot motor.

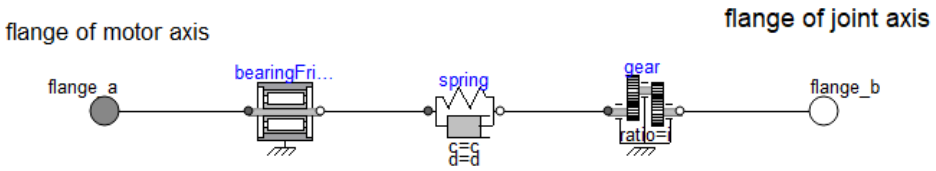




View the component gear in a similar way as for the other components. It shows the gearbox and the model of friction of the bearings and elasticity of the shafts.

The robot gearbox.





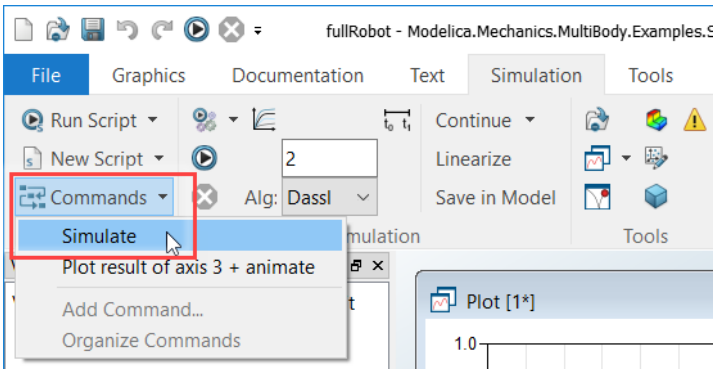
2.2.2 Simulation

Let us simulate the robot model. To use simulation commands, select the **Simulation** tab.

Simulating/manipulating with given values

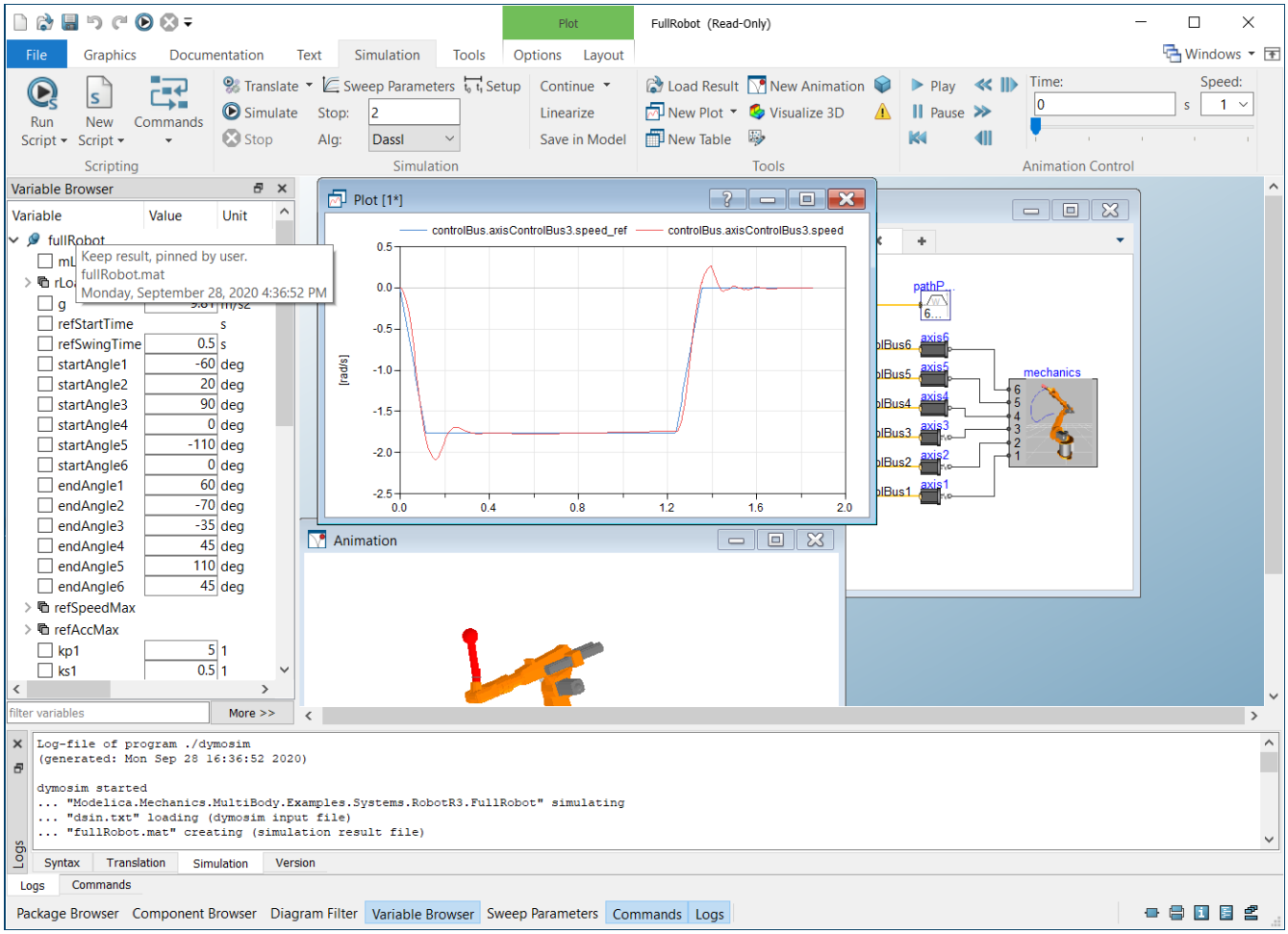
The **Simulation** tab contains commands to setup and run simulations. However, the demo has also been prepared with a command script that will simulate the model. The script is executed selecting **Commands > Simulate** in the **Simulation** tab. Please use that command.

Simulating the demo (by running a script).



The model is now translated and simulated automatically. The script also contains some setting of a plot window and an animation window. A model view window displaying the diagram layer of the model is displayed automatically (in the figure below it is partly hidden behind the plot window).

After maximizing the Dymola main window it may look the following:



Note that if you select the plot window or the animation window, you also activate corresponding tabs in the top bar. Those tabs contain commands for plotting and animating, respectively (as you will see).

(To be able to switch between the subwindows, you can use the **Windows** commands Windows to the top right of the main Dymola window, clicking on it switches active window, you can use the arrow to select directly which window should be active.)

To the left a variable browser is seen, displaying the variables and their values.

To the right a window for working with variable sweeping is displayed.

In the lower left part of the Dymola window a log window and a command window is displayed. They are by default both docked; you see them as two tabs **Logs** and **Commands**, above the last line in status bar at the bottom of the Dymola window.

Animating the robot

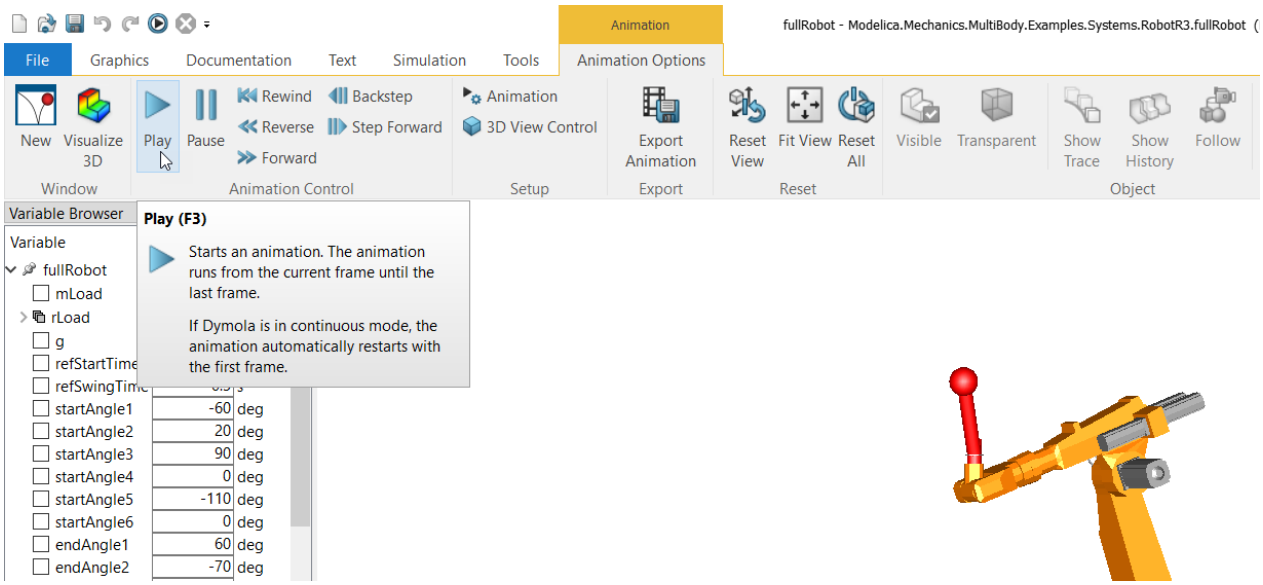
Let us start to animate the robot.

Start by maximizing the animation window displaying the robot. This will also mean that the **Animation** tab is displayed and selected in the top bar, and the header will contain [Animation] to show that the animation window is active.

Note also that you now have the common buttons for window handling to the right of the **Minimize Ribbon** button when having maximized the window.

To be able to see the animation commands all the time, click the **Minimize Ribbon** button until you see both the commands in the tab as well as the name of the command group (under the commands).

Now you can start the animation by selecting **Play** (note the corresponding tooltip that appears when you pause over the button):



This command is one of the commands in the Animation Control group in the tab, you have other buttons for pausing, rewinding, and stepping forward and backward, respectively. Also the time flow is shown and there is a possibility to set the speed of the animation (higher figures means higher speed)

If the animation window by mistake is deleted, a new can be displayed using the command **New Animation** in the Simulation tab.

Direct manipulation of the view in the animation window using the mouse has been implemented. The view can be moved, rotated and zoomed using mouse movements in combination with Meta keys:

Operation	Meta key	Mouse move (dragging)	Arrow keys
Moving up/down/left/right	none	Up/Down/Left/Right	Up/Down/Left/Right
Tilt (Rotate around x-axis)	Ctrl	Up/Down	Up/Down
Pan (Rotate around y-axis)	Ctrl	Left/Right	Left/Right
Roll (rotate around z-axis)	Ctrl+Shift	Clockwise/Counter-clockwise	Left/Right
Zoom in/out	Shift	Up/Down	Up/Down
Zoom in/out	none	Wheel	
Zoom in/out	Ctrl	Wheel	
Zoom in/out	Ctrl	Right mouse button Up/Down	

To set the rotation center on a selected component, use the context menu of the object and select **Current Object > Set Rotation Center**.

The arrow keys pan and tilt in fixed increments of 5 degrees, in addition **page up**/**page down** tilt 45 degrees. The **Home** key resets viewing transformation.

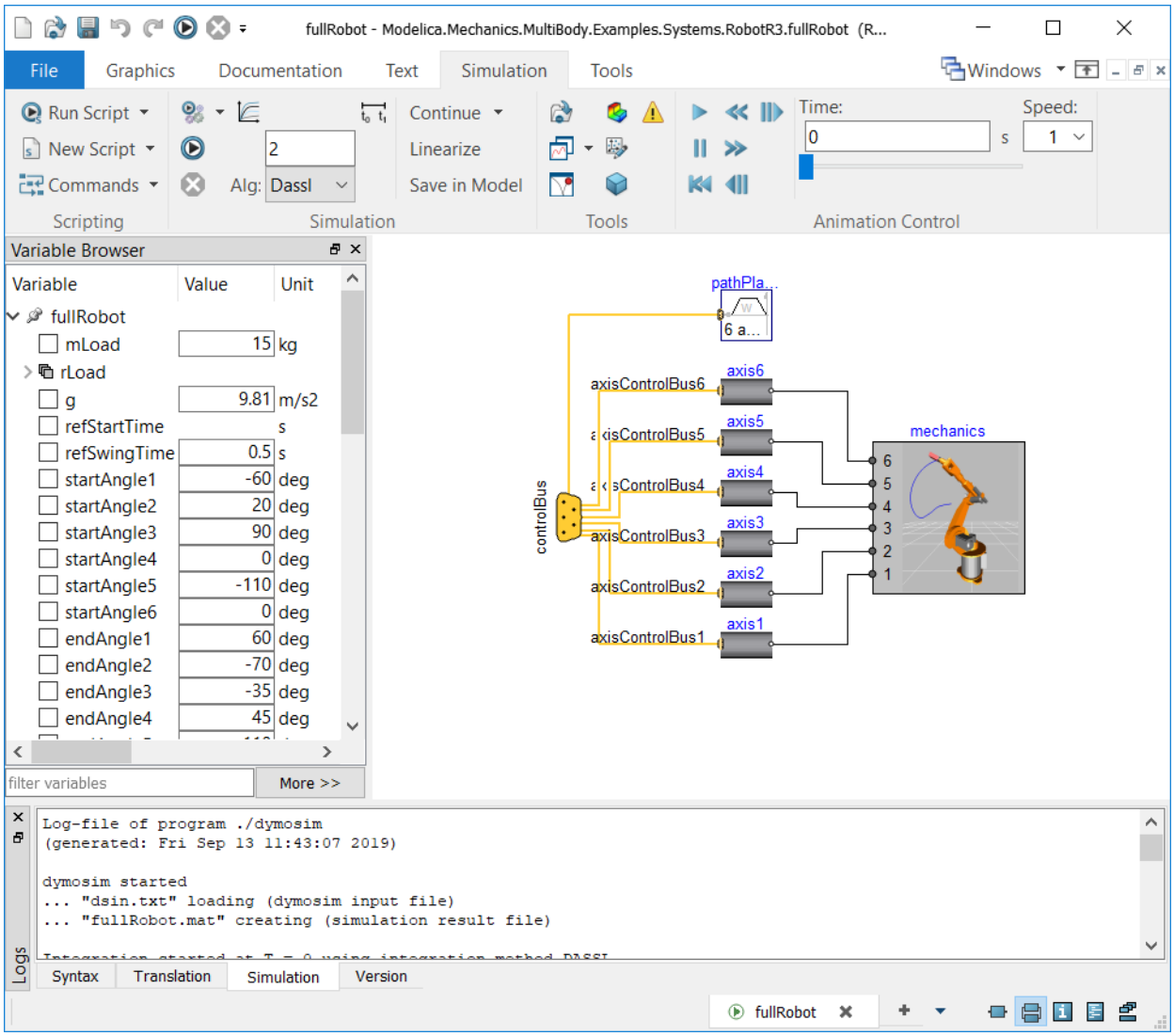
The plot window - later

Let us now look at the plot window. The plot shows the speed reference and actual speed of the third joint. We leave the plot window for the time being, we will work with it more later.

Working with the diagram when simulating – the model view window

A very convenient way to display the variables for a certain component that is of interest is to use the model view window when simulating. The model view window enables the user to follow a simulation by displaying variables and to control it by setting parameters. The user can descend into any level of the model in order to plot or display variables.

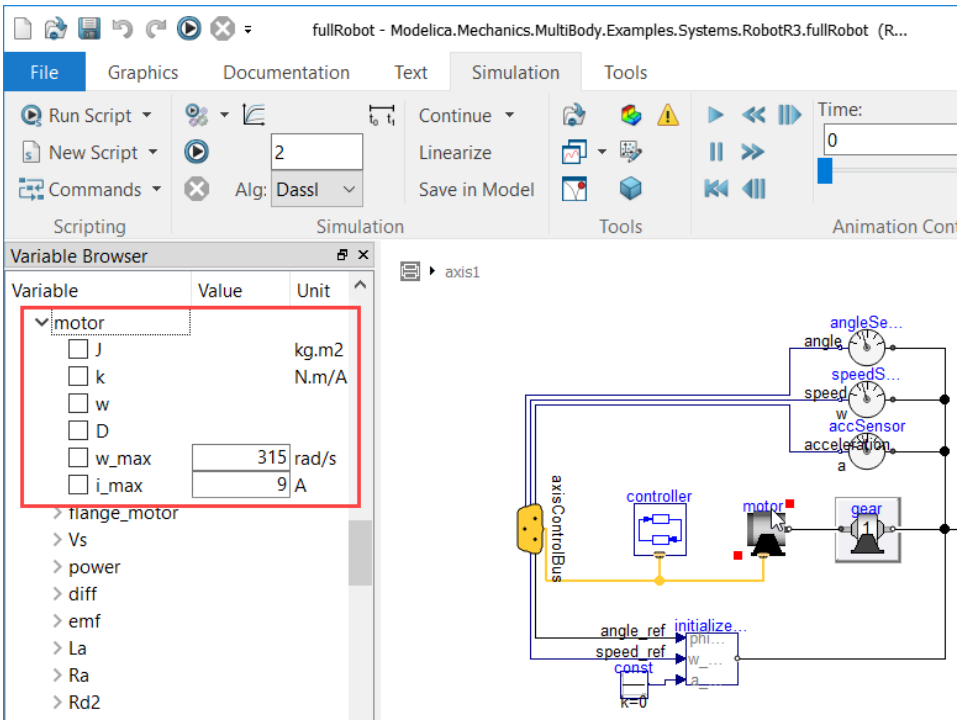
Click in the model view window to select it, and then maximize it. It will now look like:



You can now navigate exactly the same way as when working with the diagram earlier – right-click on the component you want to look into and select **Show Component** to look into it. The easiest way to go back is click what you want to show in the navigation path in the upper part of the model view window, the same as you did before (note that this path is not seen when being in the top level, but see figure later).

When having reached the interesting component, click it. That will open and highlight the selected component instance in the variable browser. In the figure below, the variables of the motor in Axis 1 (see red frames) are displayed in this way.

Using the diagram layer to select variables to be shown.



Please note that if the diagram layer window is active, selecting another component in the variable browser will also change the selection in the diagram layer window.

Changing and saving start values

Using the variable browser, you can change values, and then simulate again. It is possible to save changed start values directly into the model, see section “Saving changed start values to the model” on page 114 for more information about this possibility.

2.2.3 Other demo examples

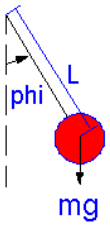
Other demo examples can be found under the **File > Demos** menu. After selecting an example, it can be simulated by running a corresponding script file as was done for the robot example.

The exception is the demos in the Vehicle Demos and the Thermal Management Demos. They contain several demos, and the relevant demo has to be opened first by expanding any package “Examples” in the package browser by clicking the expand icon ▷ before it, and then double-clicking on the relevant demo to open it. Please note that to see and run any of these demos you have to have a number of licenses for commercial libraries. Please see the description presented when opening the library for more information.

2.3 Solving a non-linear differential equation

This example will show how to define a simple model given by an ordinary differential equation. We will simulate a planar mathematical pendulum as shown in the figure.

A pendulum.



The variable m is the mass and L is the distance from the support to the center of mass. Let us assume the string is inextensible and massless, and further, let us neglect the resistance of the air and assume the gravitational field to be constant with g as the acceleration of gravity. The equation of motion for the pendulum is given by the torque balance around the origin as

```
J*der(w) = -m*g*L*sin(phi)
```

where J is the moment of inertia with respect to the origin. Assuming a point mass gives

```
J = m*L^2
```

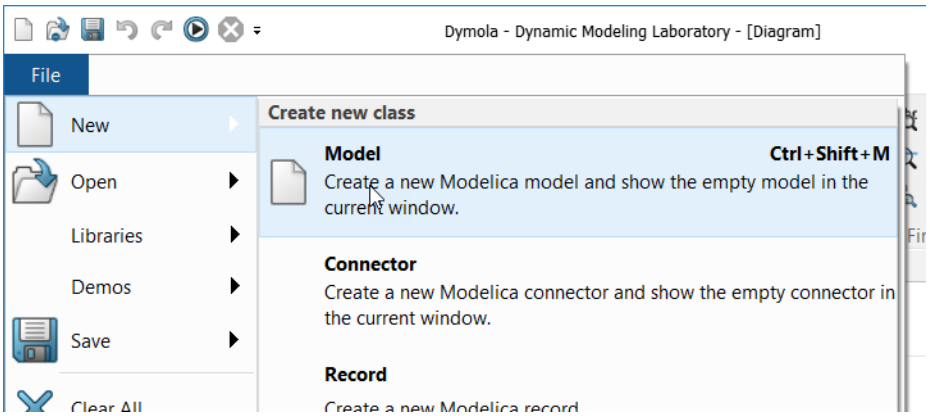
The variable w is the angular velocity and der(w) denotes the time derivative of w, i.e., the angular acceleration. For the angular position we have

```
der(phi) = w
```

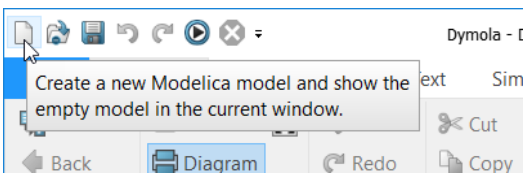
Start Dymola or if it is already started then give the command **File > Clear All** in the Dymola main window.

To create a new model, select **File > New > Model.**

The first step to create a new model.



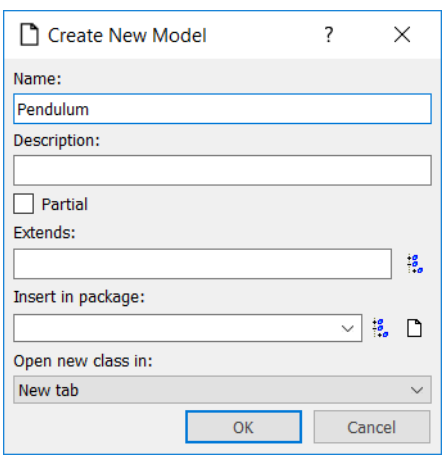
(You can also perform the opening command either by clicking the top left button in the main window:



Having several alternatives doing the same thing is not uncommon in Dymola.)

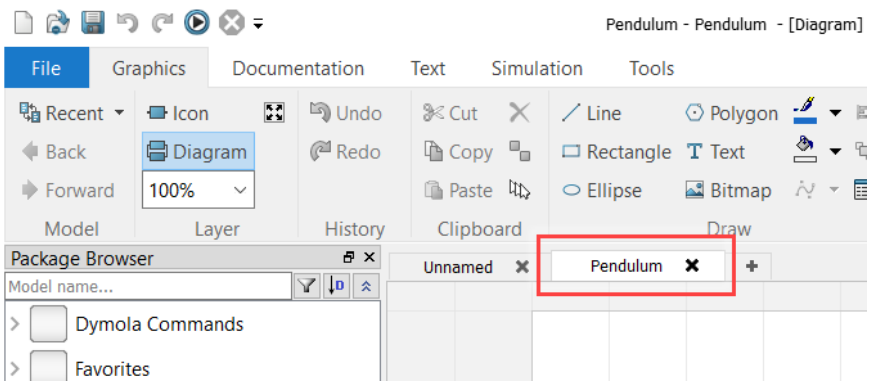
A dialog window opens. Enter **Pendulum** as the name of the model.

The dialog to name a new model component.



Click **OK**. This model will be added at the top-level. You should in general store your models into packages, as will be described later.

A new model tab will be opened with the new model; the model tabs are seen in the upper left part of the Edit window. The tabs work like working areas in for example Internet Explorer.



A model can be inspected and edited in different views. When specifying a behavior directly in terms of equations, it is most convenient to work with the model as the Modelica Text; that is, working in the Modelica text layer of the edit window.

To display this layer, select the **Text** tab. You will by default activate the Modelica editor, but note that other editors are also available in this tab, for example the Documentation editor.

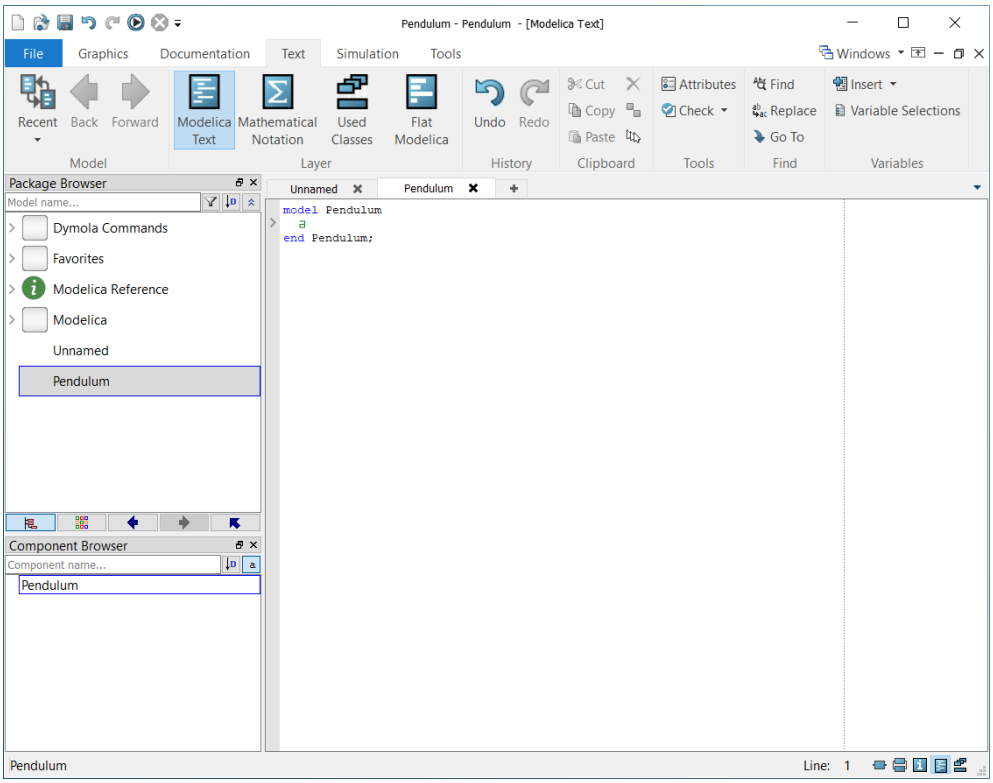
As a shortcut, you can click the **Modelica** command to the lower right in the main window:



Note that these buttons also display selected editor, you can see that right now the diagram editor is selected, indicated by gray background.

The edit window can now be used as a text editor.

The model presented in the Modelica text layer.



To declare the parameters and the variables, enter as shown the declarations for the parameters m, L and g, which also are given default values. The parameter J is bound in terms of other parameters. Finally, the time varying variables phi and w are declared. A start value is given for phi, while w is implicitly given a start value of zero.

```
model Pendulum
  parameter Real m=1;
  parameter Real L=1;
  parameter Real g=9.81;
  parameter Real J=m*L^2;
  Real phi(start=0.1);
  Real w;
equation
  der(phi) = w;
  J*der(w) = -m*g*L*sin(phi);
end Pendulum;
```

New text will be syntax highlighted (color coded) as you type, except types; e.g. Real. To get also types color coded, right-click and select **Highlight Syntax** or press **Ctrl+L**. Apart from implementing color codes for types, the command will also give a message if the syntax is not correct. It is a good idea to use the command regularly. The command does not, however, change the formatting of the text (tabs etc.). Such change is implemented by selecting the part

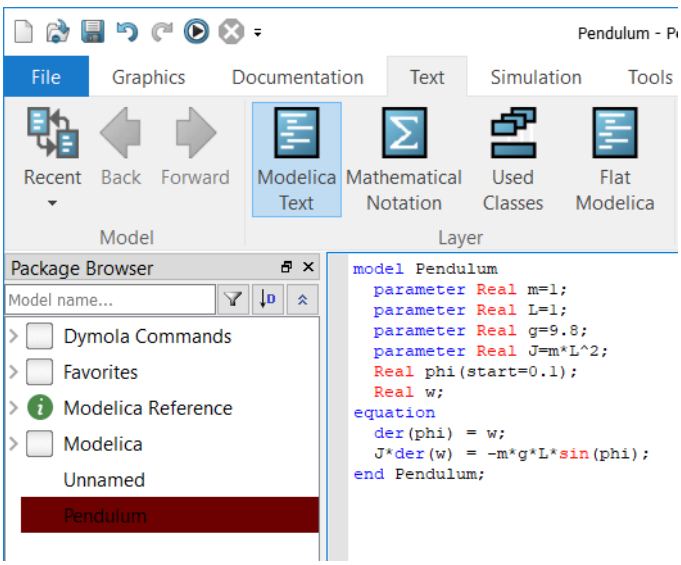
of the text that should be reformatted, and then right-clicking and selecting **Reformat Selection** (or press **Ctrl+Shift+L**).

The color codes are:

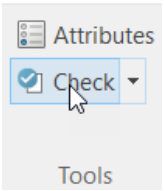
blue	keywords
red	types, operators etc.
black	values, variables, parameters etc.
green	comments

If the text seems too small to work with, the text size can be changed using the **Tools > Options** command and temporary changing the **Base font size** in the **General** tab. It is a good idea to set it back afterwards.

Declaration of parameters, variables and equations.

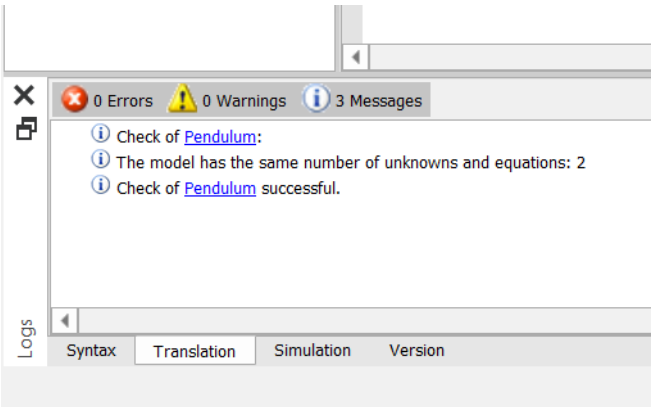


The reason that the Pendulum is red in the package browser is that it is not yet saved.



Since the model should be simulated, it should be checked before saved. The check command performs some symbolic and syntactic checks that will capture a number of errors that might have been introduced by the user. The check can be made clicking on the **Check** icon as in figure or by using the **F8** function key. The result of a check of the above text is that the logs window is displayed as a docked window in the lower part of the Dymola window. It looks the following:

Check of model.



The model is now ready to be saved. Select **File > Save > Save**. Call the file pendulum and keep the file extension to .mo and place it in a working directory. If you want, you can close the log window by clicking the cross in the vertical header of that window.

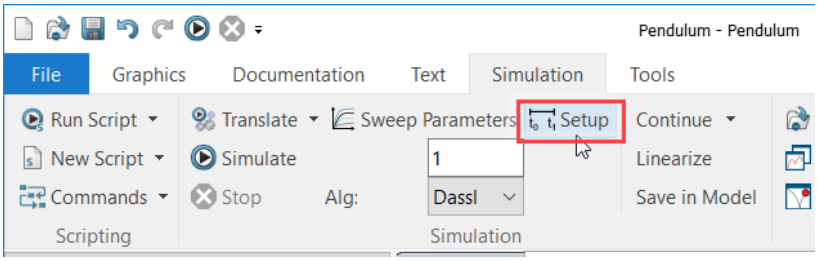
2.3.1 Simulation

Now it is time to simulate. Click the tab **Simulation**. The simulation commands are now displayed. (In rare cases you might have to click on Simulation twice, for example, in the case you have previously had an active animation window for another model.)



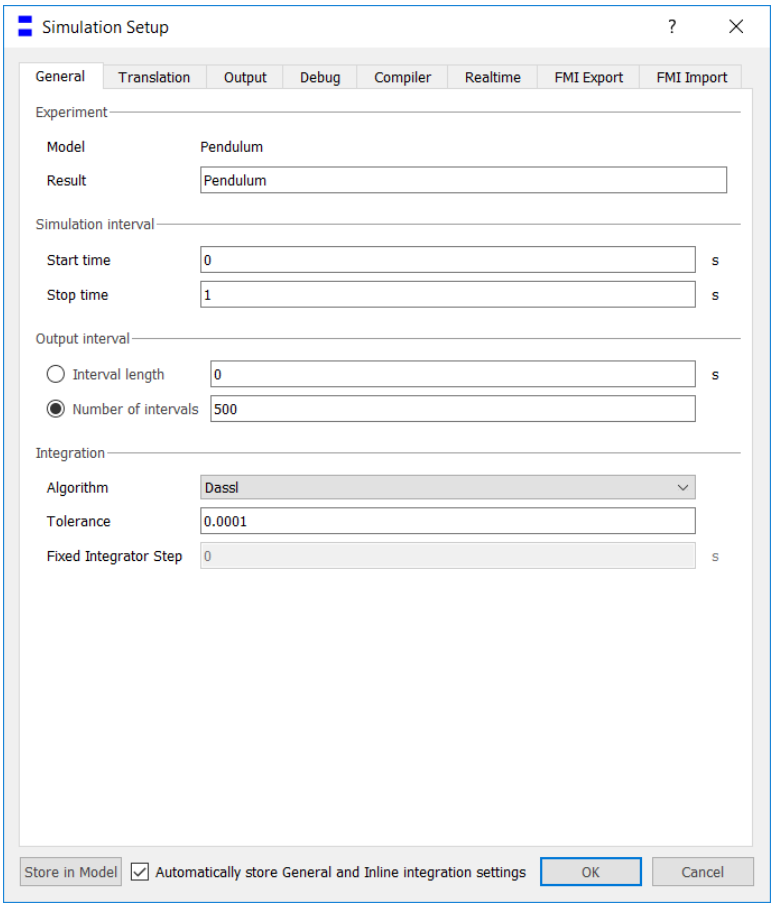
To set up the simulation select **Setup**.

Selecting Setup in the Simulation tab.



The simulation setup menu will be displayed:

The Simulation Setup menu.

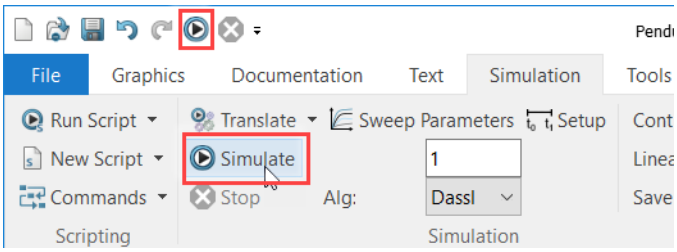


Set the **Stop time** to 10 seconds. Click **OK**.



To run the simulation select **Simulate**.

Selecting Simulate in the Simulation tab.



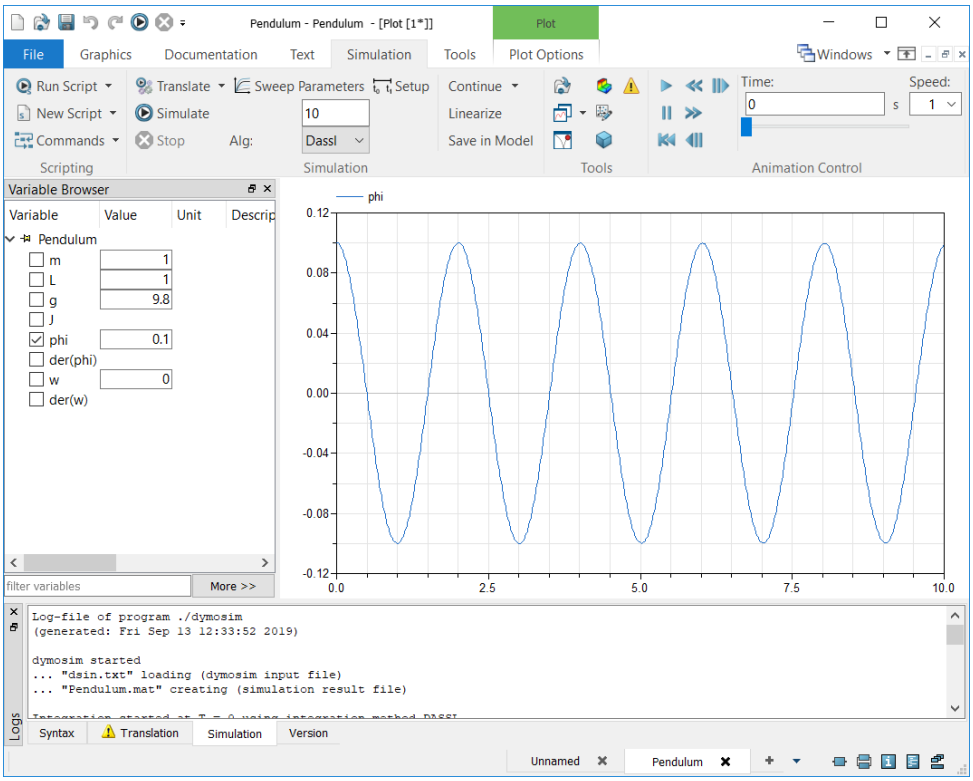
(Note that the command is also by default available in the quick access bar at the top left of the Dymola window.)

Dymola first translates and manipulates the model and model equations to a form suitable for efficient simulation and then runs the simulation. (You may explicitly invoke translation yourself by selecting **Translate** .)

You will get a warning that the initial conditions are not fully specified. (The warning can be seen in the Translation tab of the log window that will appear – you must click the Translation tab to see the warnings since the Simulation tab is displayed now.) However, Dymola will select default initial conditions, so the simulation will work. We will discuss how to get rid of the warnings later. For now, you can just leave the log window.

When the simulation is finished, the variable browser displays variables to plot. To see the plot better, maximize the plot window in the edit window. Then click in the square box in front of phi to get the angle plotted as shown below.

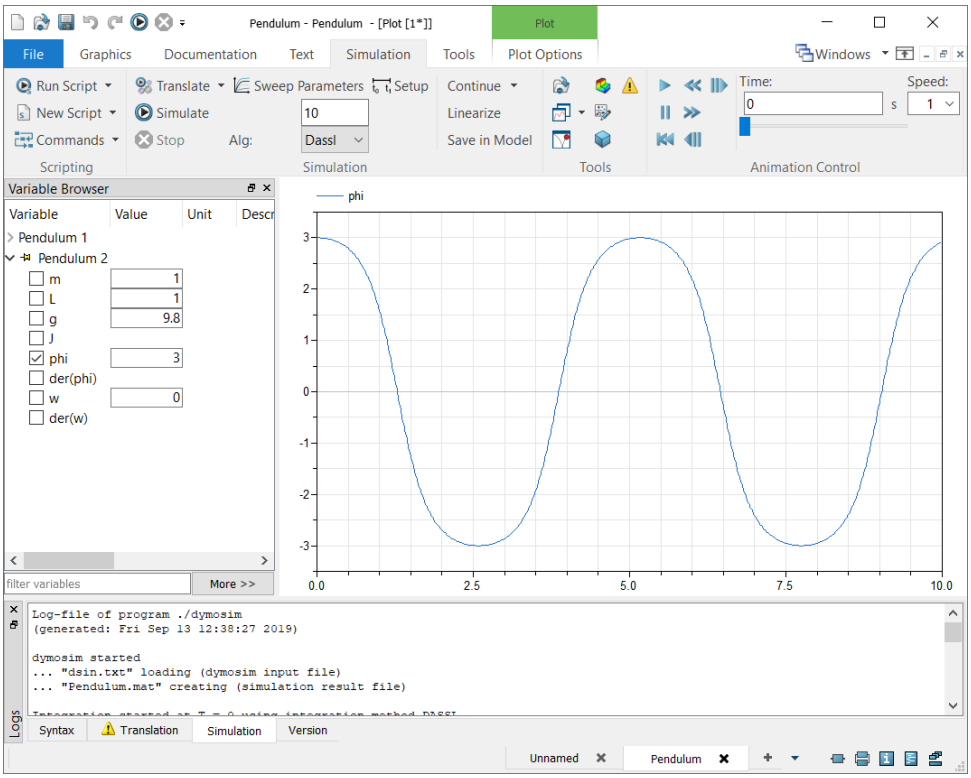
Plotting the angle.



Let us study a swing pendulum with larger amplitude and let it start in almost the top position with phi = 3. It is easy to change initial conditions. Just enter 3 in the value box for phi, click outside the box or press **Enter** to validate, and click on the **Simulate** button.

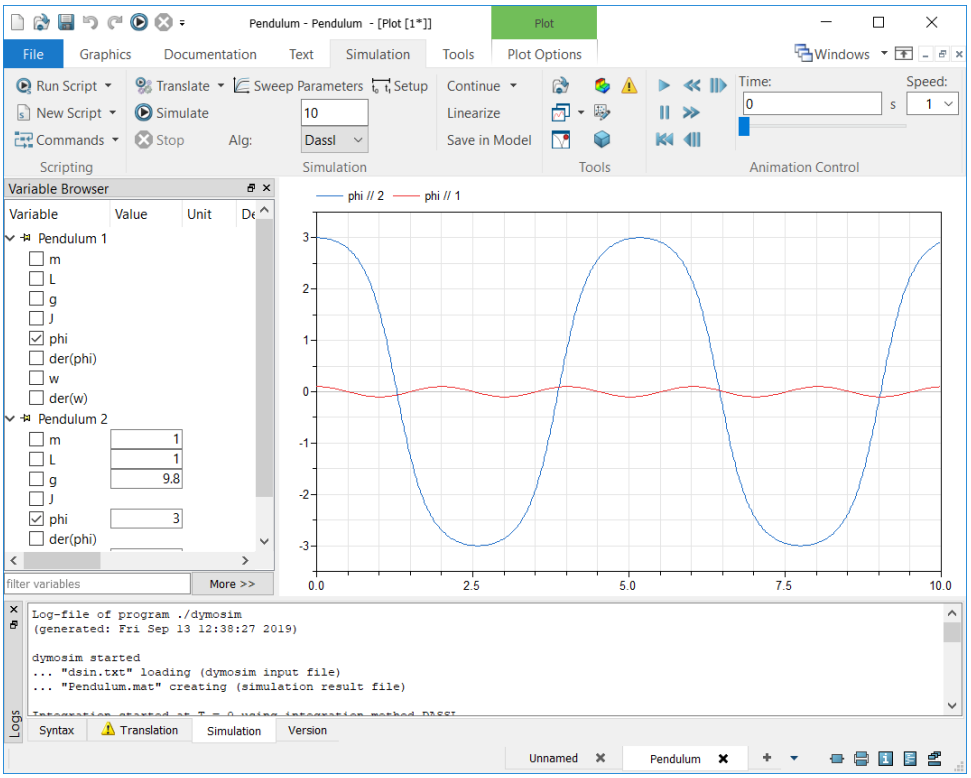
You can also change start values and parameters by typing in the command window; e.g. type `phi=3` (followed by carriage return) for a change of phi.

Pendulum angle when starting in almost the top position.



The results of previous simulations are available as the experiment Pendulum 1 in the Variable browser. We can open it up and have phi of the two runs plotted in the same diagram by expanding “Pendulum 1” in the variable browser and check the checkbox for “phi”.

Results from two simulations.



Values of parameters are changed in a similar way. To simulate another length of the pendulum, just enter a new value for L and click on the simulate button.

2.3.2 Improving the model

Using pre-defined physical quantities

The parameters and variables are more than real numbers. They are physical quantities. The Modelica standard library provides type declarations for many physical quantities. Using these instead of declaring variables/parameters yourself gives two advantages:

- The declarations will be faster.
- Additional features will automatically be present; e.g. units and limits.

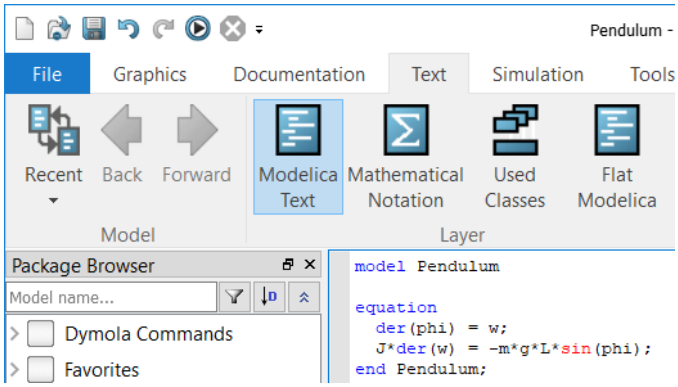
So let us change the model, using ready-made physical quantities. The starting point is the Pendulum model built previously. (If Dymola is not started, start it and use **File > Open > Open** to open Pendulum.mo.)

Click the **Text** tab to show the Modelica text layer.



(Note that you can also, in the bottom of the Dymola window, to the right, click directly the icon for the Modelica text layer to open it.)

Mark the declarations of variables/parameters in the text and delete them. Do not delete the equation, that one we will keep. The result will be:

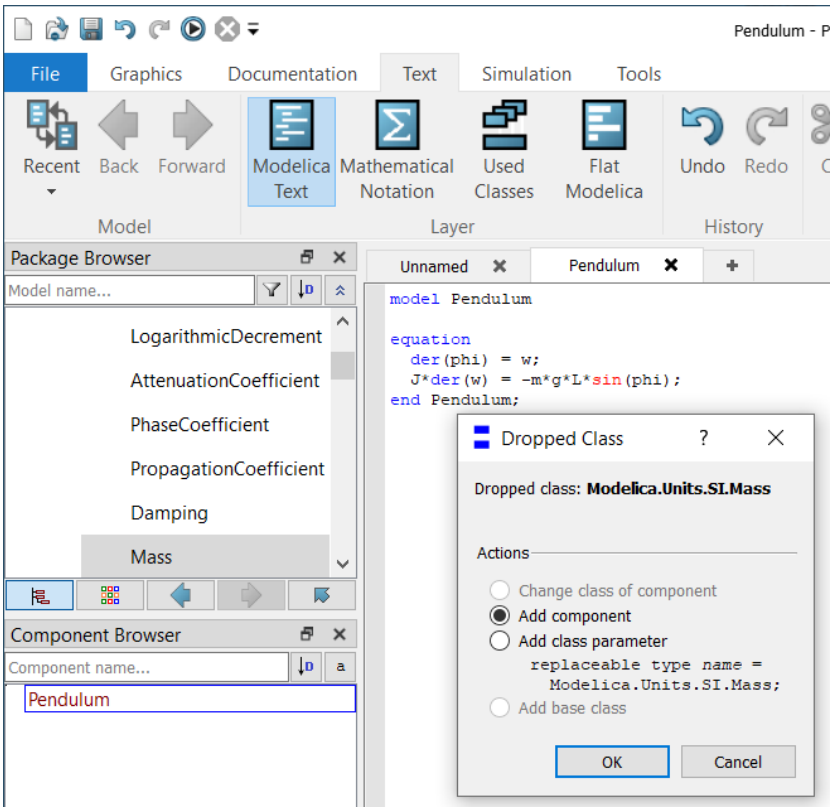


Open Modelica.Units.SI in the package browser by first expanding the Modelica package by clicking on the expand icon ▷ in front of it, then expanding “Units” by clicking the expanding icon in front of it, and then finally expanding “SI” by clicking on the expand icon in front of it. Now a number of physical types should be visible.

What is to be done is to redo the declaration part of the model presented on page 55 using the physical types available in Modelica.Units.SI. The mass is the first one to be declared. To find “Mass” in the package browser, click the package “SI” (to get a good starting point) and then press **m** on the keyboard. The first physical type starting with “m” will be displayed. Pressing **m** again will find the next physical type starting with “m” and so on. (Another way to find it fast is by sorting the physical types in alphabetical order, you can do that by clicking the button  in the header of the package browser.)

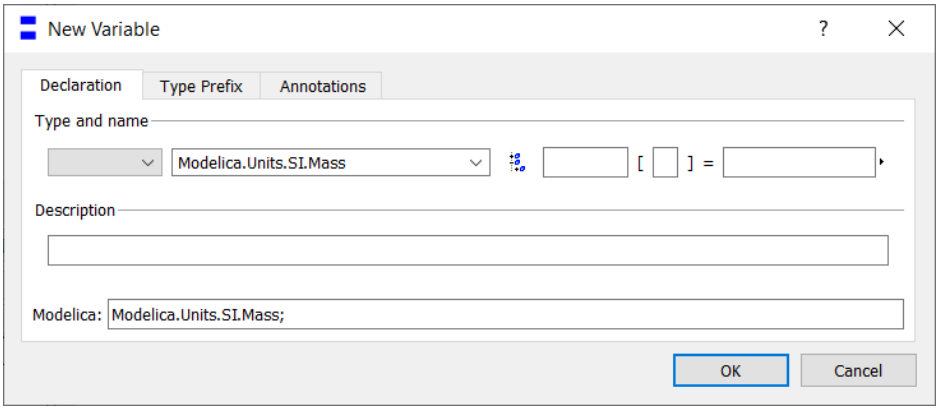
Once “Mass” is found, drag it to the component browser. The following menu appears:

Adding a physical type.

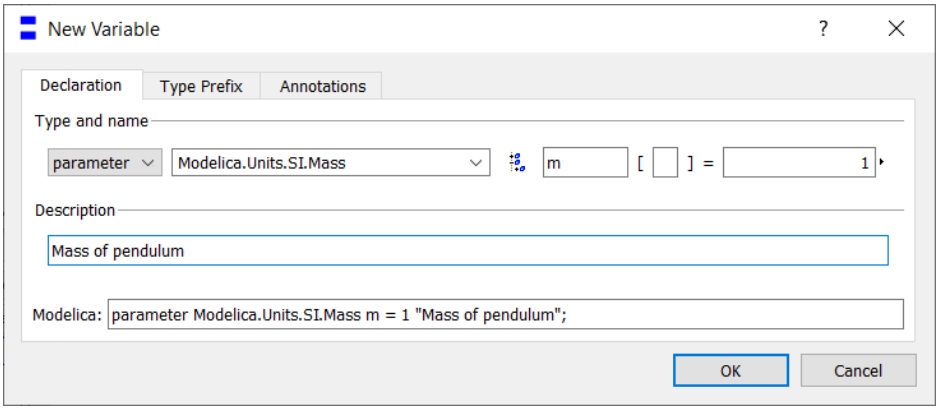


The choice to add a component is pre-selected. Click **OK**. A menu to declare a variable appears:

Declaring a variable.

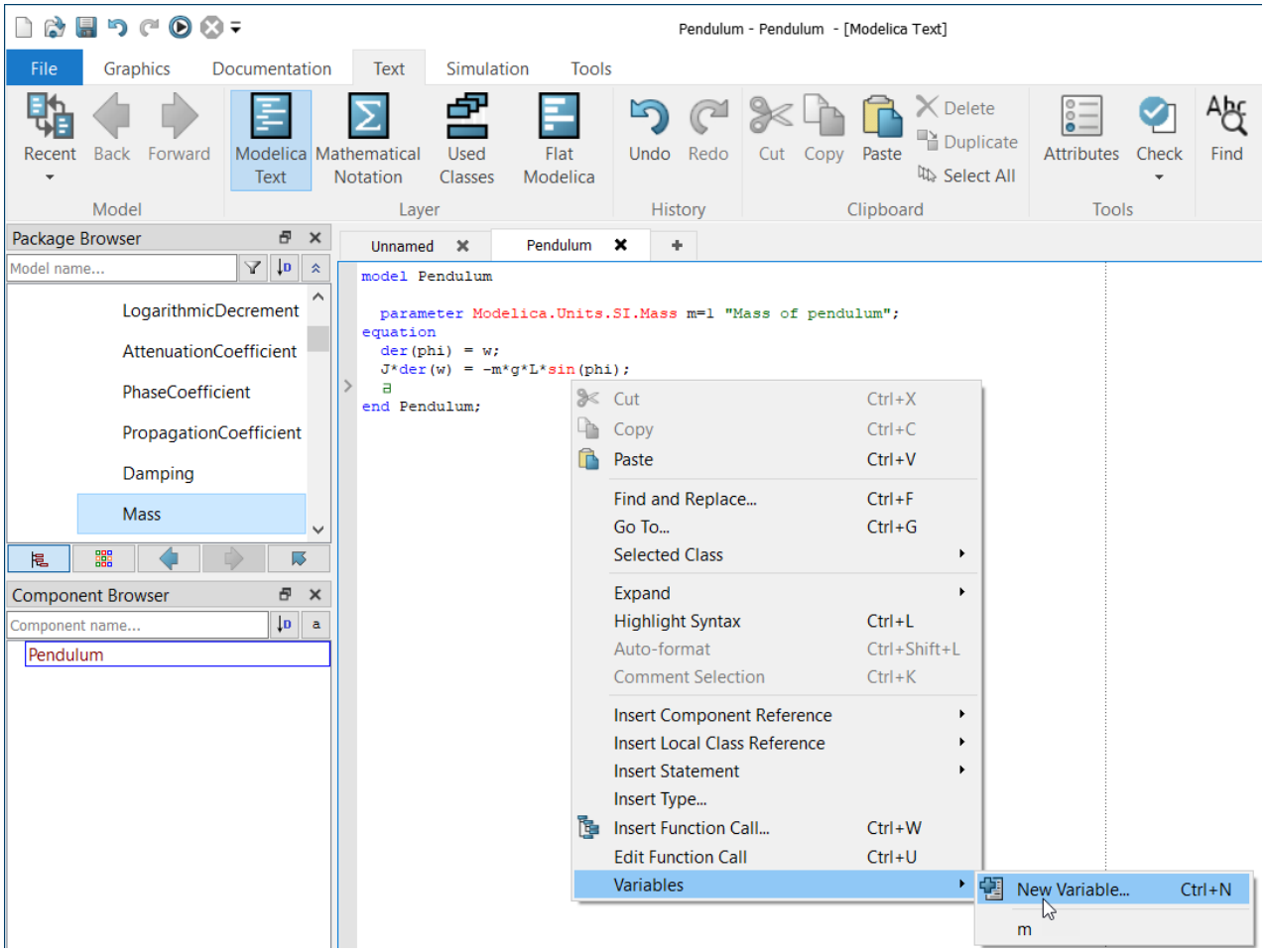


Now we can specify the type prefix (parameter), the name, value and description etc. (By keeping the cursor on top of an input field for a while a tooltip text pops up explaining the field.) Complete the description in the following way:

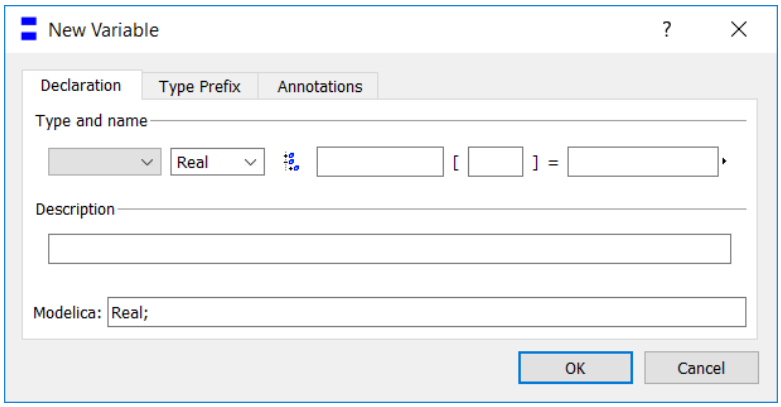


Click **OK** and the text appearing in the bottom row is inserted into the Modelica text window.

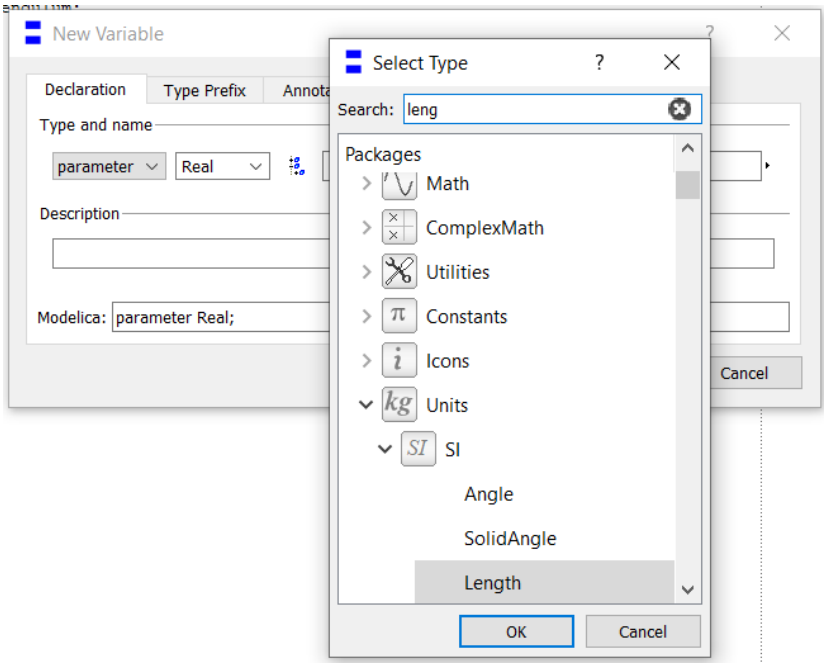
You can also declare variables by using context menu commands. Let us define the next parameter by right-clicking in the Modelica Text layer and select **Variables > New Variable…**.



The result is the declaration dialog, just as before (note by the way from the figure above that this is a way to edit present variables as well, we could have selected m in the menu here):



After having selected “parameter” from the dropdown menu, we realize that the default type “Real” is not what we want in this case; neither are any of the other dropdown alternatives relevant for us. What we really would like is some “length” type. However, we can browse for any type available in the package browser by the browse button. And, on top of that, we can type in the **Search** input field. The search is dynamically updated while typing. This is illustrated in this figure:

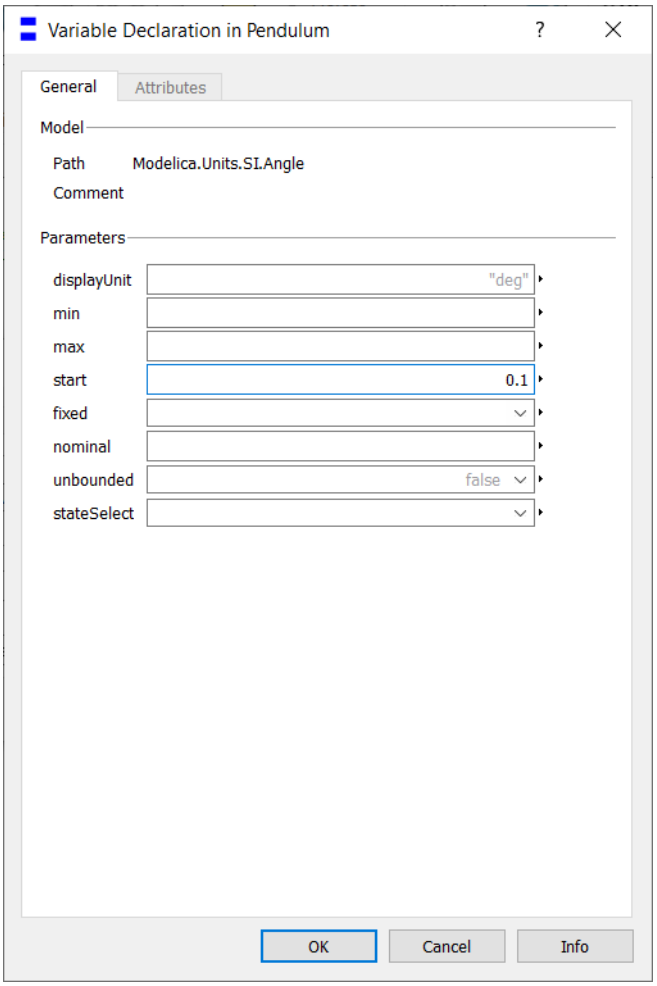


This was exactly what we wanted. It is sufficient to click **OK** to get the type, since it is preselected. (If we wanted another of the displayed types, we could have selected that before clicking **OK**.)

The other quantities are defined in analogue ways. (It is a good idea to try both ways above to see which one you like the best.) When coming to the variables “phi” and “w”, they are variables, not parameters. Nothing should be entered in the type prefix input field for those.

When completing the form to declare the angle phi, the start value of the angle is defined by clicking on the small triangle to the right of the value field and selecting **Edit**. A submenu pops up. Enter 0.1 for start. The result will look like:

Entering a start value.



Click **OK**.

The following result is displayed:

The result.

Unnamed | Pendulum

```
model Pendulum
  parameter Modelica.Units.SI.Mass m=1 "Mass of pendulum";
  parameter Modelica.Units.SI.Length L=1 "Length of pendulum";
  parameter Modelica.Units.SI.Acceleration g=9.81;
  parameter Modelica.Units.SI.MomentOfInertia J=m*L^2 "Moment of inertia";
  Modelica.Units.SI.Angle phi(start=0.1) "Pendulum angle";
  Modelica.Units.SI.AngularVelocity w;
equation
  der(phi) = w;
  J*der(w) = -m*g*L*sin(phi);
end Pendulum;
```

The annotation icon almost at the end of the text indicates the presence of graphical information or annotations. It may be displayed. This can be done in two ways, either by clicking on the expand icon or on the annotation icon, or by right-clicking to get the context menu, and then select **Expand > Show Entire Text**. Either way, it is revealed that the annotation is an annotation documenting which version of the Modelica standard library was used. Dymola uses this information to check if compatible versions of libraries are used and to support automatic upgrading of models to new versions of libraries. A model developer can provide conversion scripts that specify how models shall be upgraded automatically to new versions of a library.

Please note that if any parameter/variable should be changed later on, the context command **Variables** can be used to select the variable that should be edited.

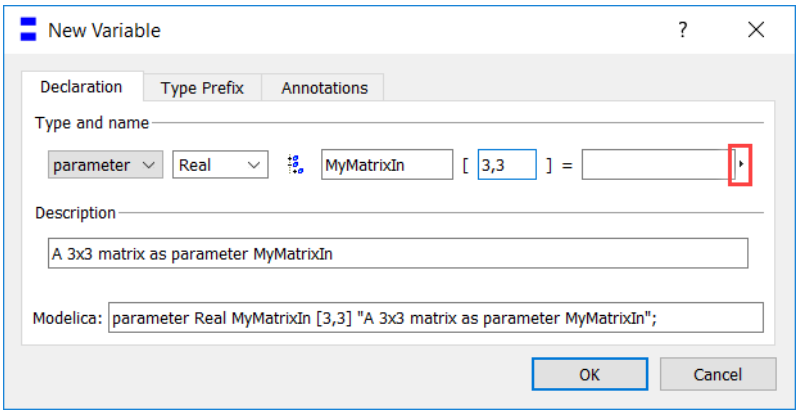
(Another way of inserting pre-defined physical quantities is to recognize that these are types. Types can be inserted in the code by right-clicking to pop the context menu and then selecting **Insert Type**. The menu that pops enables searching by typing in names. The advantage is that all types in all open packages are searched. It will be easy to find a known type even if it located in another package. However, to also get help from the **Declare variable** menu as above, the user has to

- Insert the type on an empty line
- Enter a space and the name of the variable
- Conclude with a semicolon
- Use the context command **Variables** and select the new variable)

[ ]

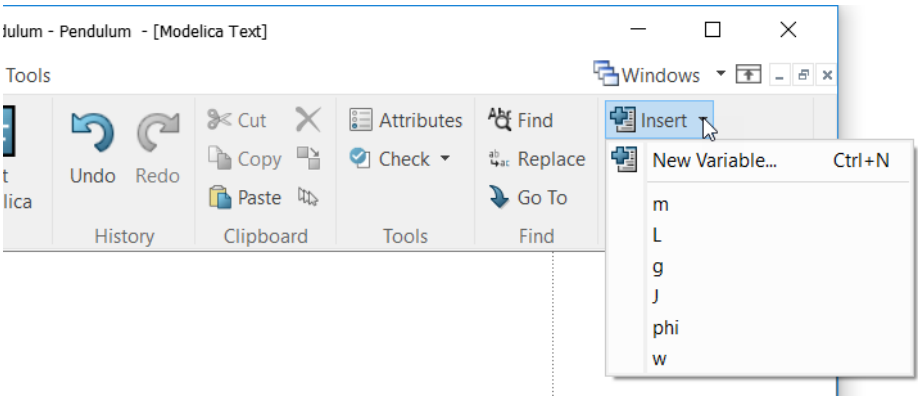
Note (outside the example) the ease of defining arrays/matrices using **Variables > New Variable…**. In the Declare Variable dialog that is displayed, an “Array dimensions” field is present after the “Variable Name” field. By entering “3” in this field, an array containing 3 elements is defined, entering “:” defines an expandable array. Entering “3,3” defines a 3 x 3 matrix, entering “:,:” defines an expandable matrix.

Example of defining a 3 x 3 matrix Real parameter.



To enter values or/and define the size, click on the arrow after the last field (see red marking in the figure above), then on the **Edit** command  . You will get a dialog to enter the values.

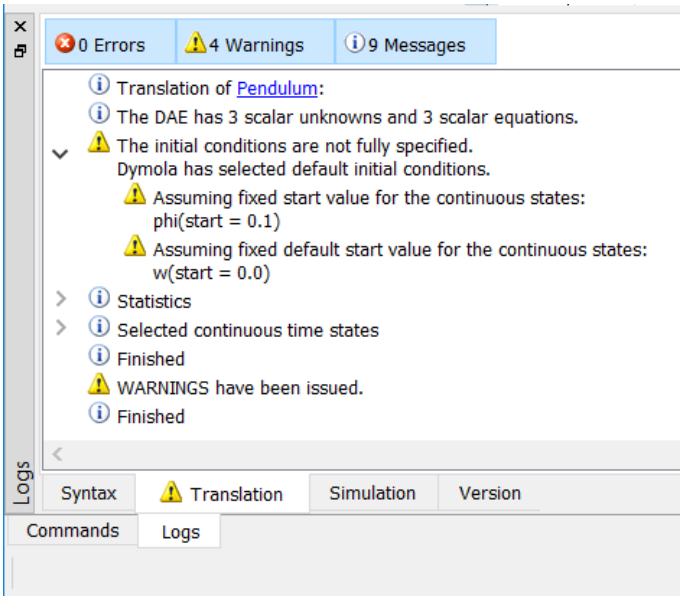
Finally, please note that you don´t have to be inside the Modelica editor to define or modify variables, you can always use the **Insert** command in the **Text** tab (it is also available in the **Graphics** tab):



If you click on the arrow, like in the figure, you can select from the present variables, or define a new one. You can also click directly on the icon, then the menu for creating a new variable will appear.

Handling of warnings

When simulating, we still get a warning that the initial conditions are not fully specified. By clicking on the **Translation** tab in the log window and scroll to the top the following can be seen (when you have expanded “the warning” node by clicking on the expand icon ▷before it):



In order to have sufficient number of initial conditions, Dymola look at the possibility to use variables as states and attribute fixed start values to such states. (For more information, please see chapter “Introduction to Modelica”, section “Initialization of models” and (more advanced) the chapter “Advanced Modelica Support”, section “Means to control the selection of states”.)

Dymola presents in the warnings what variables have been selected as states with fixed start values.

Dymola assumes fixed start values phi=0.1 and w=0. We have set the start value of phi to 0.1 (above), while the start value of w was implicitly given a start value of zero (default). However, the attribute fixed is the problem. Fixed=true means that the start value is used to initialize the model; it must be satisfied during initialization. Fixed=false means that the value is just a guess-value for a non-linear solver. For variables fixed is default false. Since the intention was to use these variable values as initialization values, the best is to explicitly set the fixed-attribute to true – and also explicitly specify the start value of w to zero. The resulting code (generating no warnings) will be:

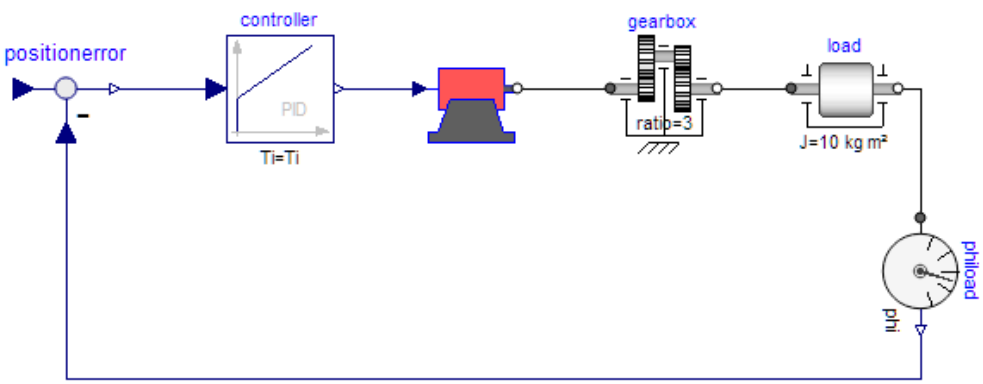
Unnamed | Pendulum

```
model Pendulum
  parameter Modelica.Units.SI.Mass m=1 "Mass of pendulum";
  parameter Modelica.Units.SI.Length L=1 "Length of pendulum";
  parameter Modelica.Units.SI.Acceleration g=9.81;
  parameter Modelica.Units.SI.MomentOfInertia J=m*L^2 "Moment of inertia";
  Modelica.Units.SI.Angle phi(start=0.1, fixed=true) "Pendulum angle";
  Modelica.Units.SI.AngularVelocity w(start=0, fixed=true);
equation
  der(phi) = w;
  J*der(w) = -m*g*L*sin(phi);
end Pendulum;
```

2.4 Using the Modelica Standard Library

In this example, we will show how a model is built up using components from the Modelica Standard Library. The task is to model a motor drive with an electric DC motor, gearbox, load, and controller.

Motor drive built with standard components.



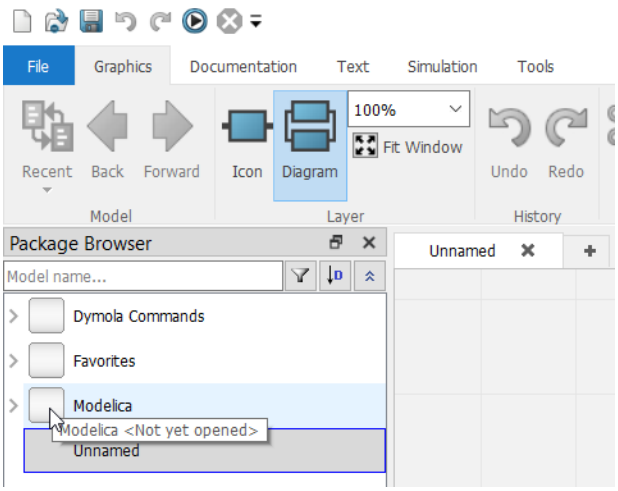
As when building a real system, there are several approaches. One extreme approach is to build the system from scratch. However, it is often a difficult and time-consuming task. Another approach is to investigate if the system already is available on the market or if there is some product that easily can be adapted or modified. If not, build the system from components available when possible and develop only when necessary.

The idea of object oriented modeling is to support easy and flexible reuse of model knowledge. Modelica has been designed to support reuse of model components as parts in different models and to support easy adaptation of model components to make them describe similar physical components. The design of Modelica has also been accompanied by the development of model libraries.

2.4.1 The Modelica Standard Library

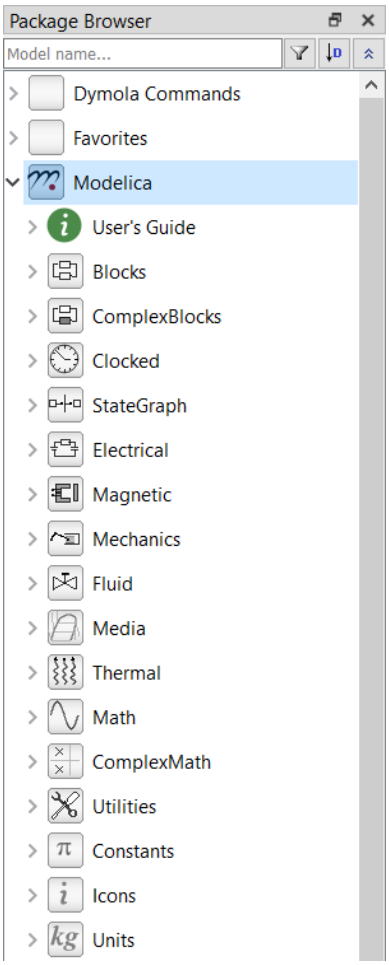
We will now have a look at the Modelica Standard Library to see what is available and how we access the model components and their documentation. To open the library, double-click on Modelica in the Package browser.

Opening the Modelica Standard Library.



Dymola reads in the library. The Modelica Standard Library is hierarchically structured into sub-libraries.

The sub-libraries of the Modelica Standard Library.



As shown by the package browser, the Modelica Standard Library includes

- **User´s Guide** with a short user´s guide on the whole library. Each subpackage below may have their own user´s guides.
- **Blocks** with continuous and discrete input/output blocks such as transfer functions, filters, and sources.
- **ComplexBlocks** provides basic input/output control blocks with complex signals. (This library is especially useful e.g. in combination with the library Modelica.Electrical.QuasiStationary in order to build up very fast simulations of electrical circuits with periodic currents and voltages.)
- **Clocked** to precisely define and synchronize sampled data systems with different sampling rates. The library has elements to define periodic clocks and event clocks that

trigger elements to sample, sub-sample, super-sample, or shift-sample partitions synchronously. The library is suited for implementation of control systems.

- **StateGraph** for modeling of discrete events and reactive systems by heretical state machines. Please note that a more advanced library is available using the command **File > Libraries > State Graph**. For more information about this library, please see section "Libraries available in the File menu by default" starting on page 132.
- **Electrical** provides electric and electronic components (for analog, digital, machines and multi-phase models) such as resistor, diode, DC motor, MOS and BJT transistor.
- **Magnetic** contains magnetic components to build especially electro-magnetic devices.
- **Mechanics** includes one-dimensional and 3-dimensional translational, rotational and multi-body components such as inertia, gearbox, planetary gear, bearing friction and clutch.
- **Fluid** contains components to model 1-dimensional thermo-fluid flow in network of vessels, pipes, fluid machines, valves and fittings. All media from Modelica.Media can be used. A unique feature is that the component equations and the media models as well as pressure loss and heat transfer correlations are decoupled from each other.
- **Media** includes property models of media.
- **Thermal** provides models for heat transfer and thermo-fluid pipe flow.
- **Math** gives access to mathematical functions such as sin, cos and log and operations on matrices (e.g. norm, solve, eig, exp).
- **ComplexMath** contains complex mathematical functions (e.g. sin, cos) and functions operating on complex vectors.
- **Utilities** contain functions especially for scripting (operating on files, streams, strings and systems).
- **Constants** provide constants from mathematics, machine dependent constants and constants from nature.
- **Icons** provide common graphical layouts (used in the Modelica Standard Library).
- **Units** with SI units and non-SI units. SI units contain about 450 type definitions with units, such as Angle, Voltage, and Inertia based on ISO 31-1992. Conversions between SI units and non-SI units, and vice versa, are also included.

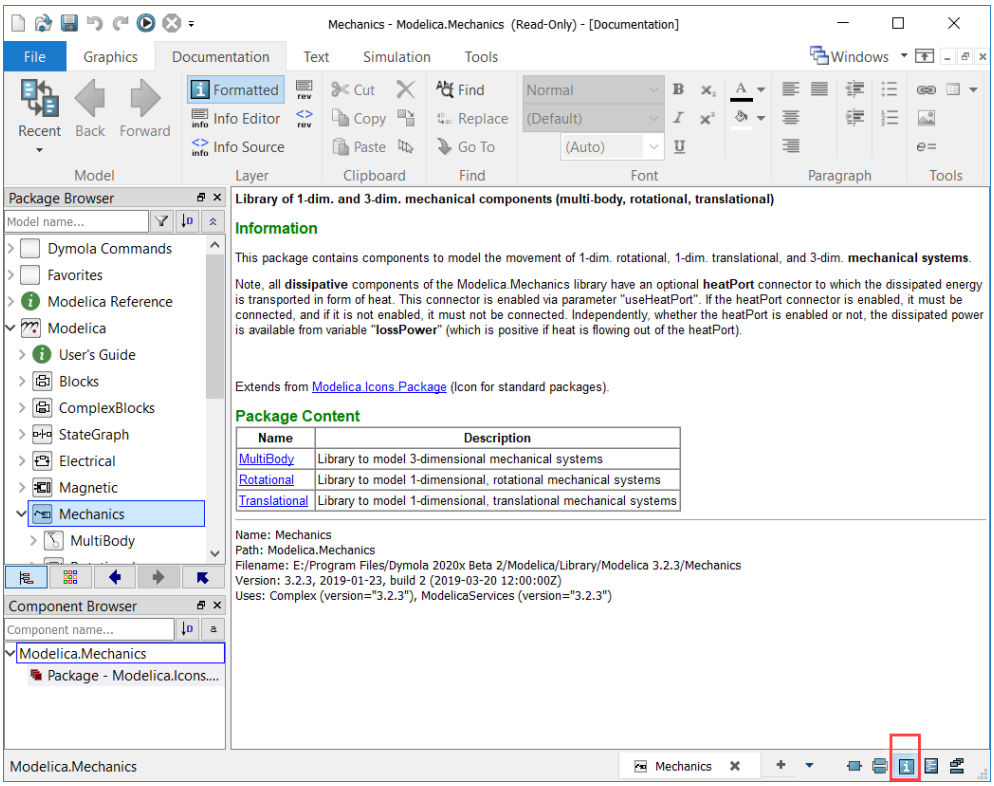
To get documentation for the entire Modelica Standard Library, place the cursor on Modelica, right-click and select **Info**. An information browser is directed to an HTML file containing documentation for Modelica. This documentation has been generated from the Modelica description of the library. There is basic information such as the content of the library, conventions and conditions for use.

Dymola comes also with other free model libraries. A list of these libraries is given by the command **File > Libraries**.

The package menu gives direct access to the sub-libraries. We will need components from various sub-libraries. We will need rotational mechanical components as well as electrical components for the motor.

To open the Modelica.Mechanics, double-click on Modelica.Mechanics in the Package browser. To show the documentation layer of the library, click the **Documentation** tab.

Opening Modelica.Mechanics.

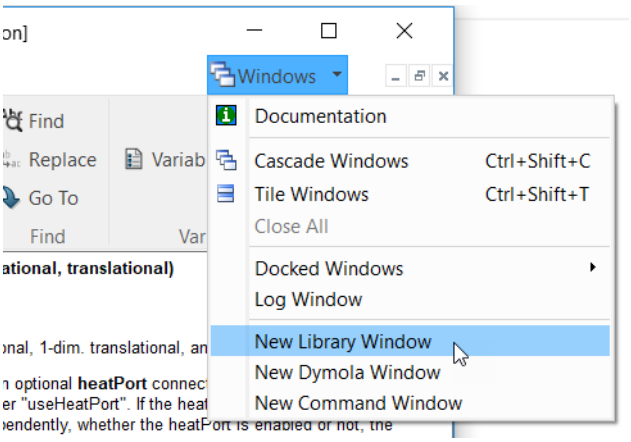




Note that you can also click on the documentation layer icon in the toolbar to the lower right of the main Dymola window (framed in red in the figure above).

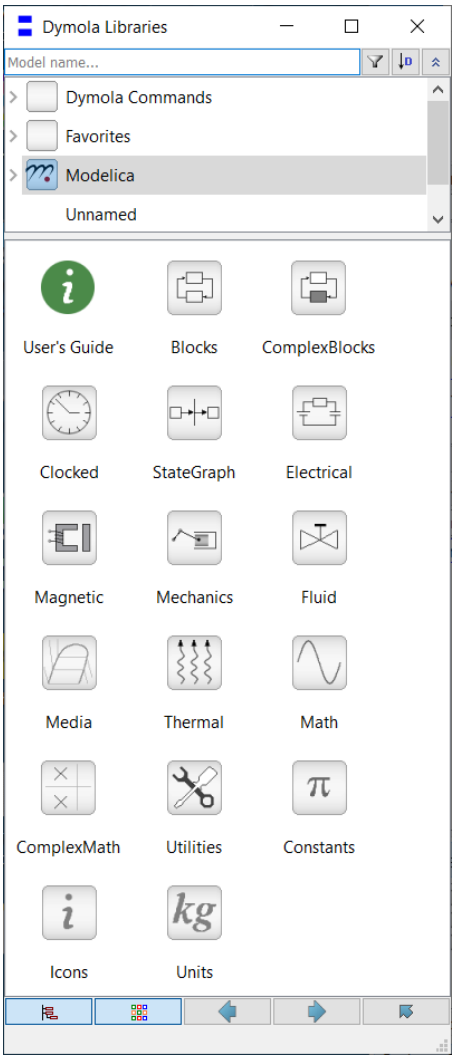
Another way to see the documentation is to, as previously demonstrated, place the cursor on Mechanics, right-click and select **Info**.

Besides using the package browser of the Dymola window, it is also possible to open a library window that also contains a browser. It can be done in two ways. If the library window should contain the Mechanics package, select “Mechanics” in the package browser and right-click to get a menu. Select **Open Library Window**. If the window should contain the “top package” in the browser (Modelica in this case), use the Windows menu in the top of the Dymola window (click the arrow) and select **New Library Window**.



Using the latter, selecting Modelica in the Package browser in the upper part of the window (and adapting the window) will display the following:

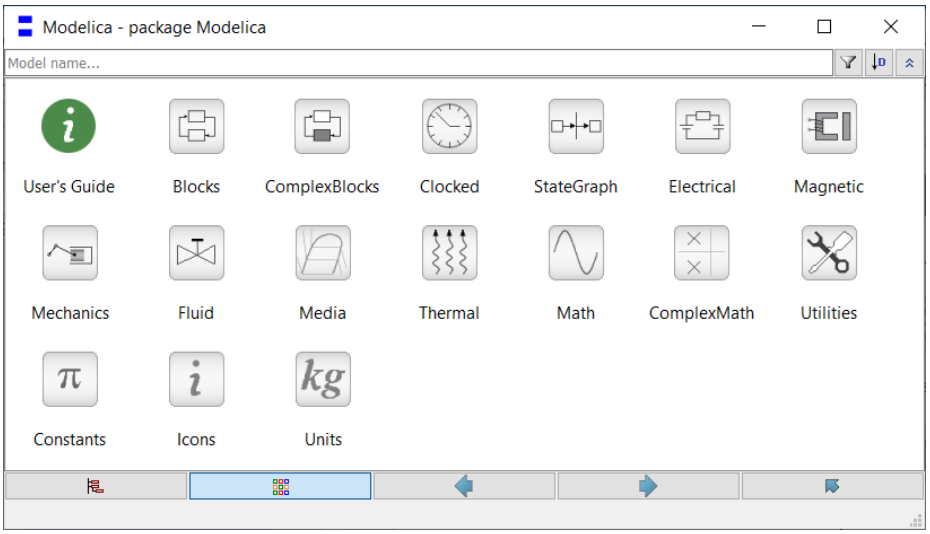
A library window displaying the Modelica Standard Library.



A Library window includes a package browser, where the components of the selected sub-library are displayed in a part of the window.

By closing the package browser by toggling the button to the bottom left, double-clicking on the icon for Modelica and adapting the window to the content the following will be displayed. Please note that now the name of the package will be displayed in the window title bar.

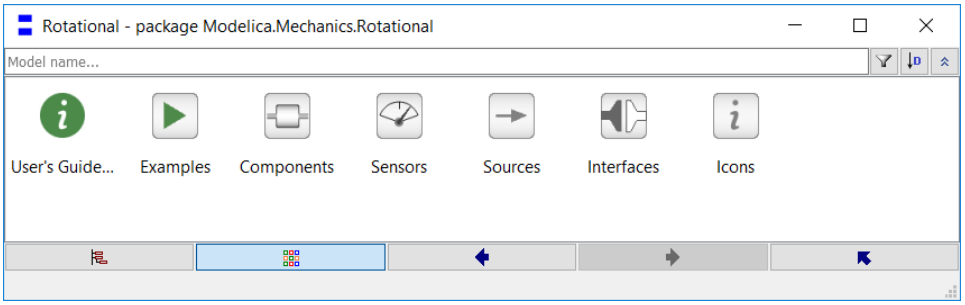
A library window displaying the components of the Modelica Standard Library.



By using the right button at the bottom, it is possible to go up in the package hierarchy and by double-clicking on the icons in the window it is possible to go down in the hierarchy. The left and right arrow buttons allow going back and forth as in an ordinary web browser.

Open Modelica.Mechanics.Rotational in the library window by first double-clicking on the icon for Mechanics and then on the icon for Rotational. The package Rotational contains components for rotating elements and gearboxes, which are useful for our modeling of the electrical motor drive.

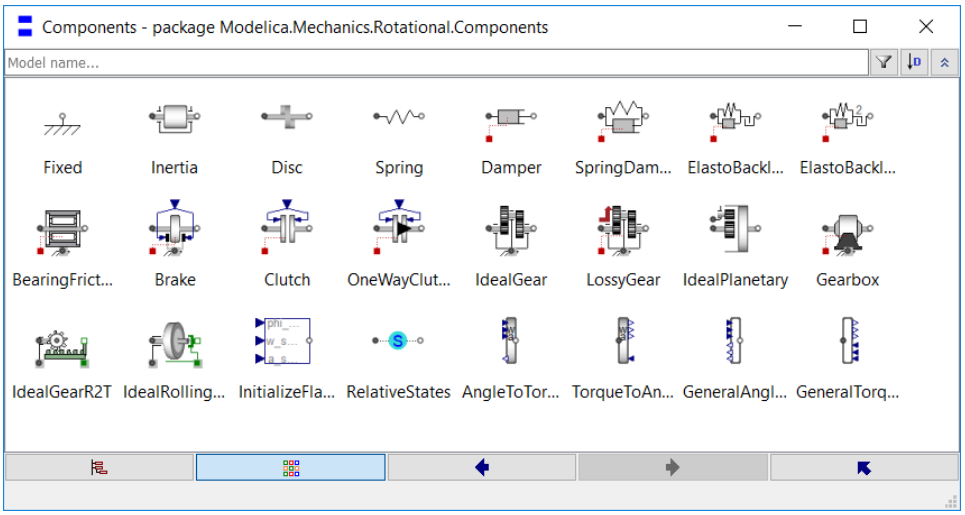
The rotational mechanics library window.



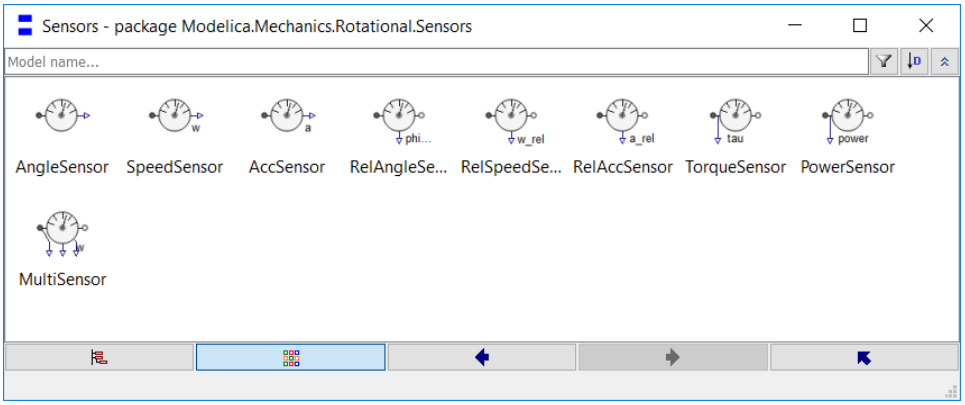
The Info for Modelica.Mechanics.Rotational contains important information on the package content.

By navigating in the packages/libraries present here (using double-clicking and left arrow) we will find a number of components that might be of interest to us. As examples, this is how the libraries Components and Sensors will look like:

The Components library.



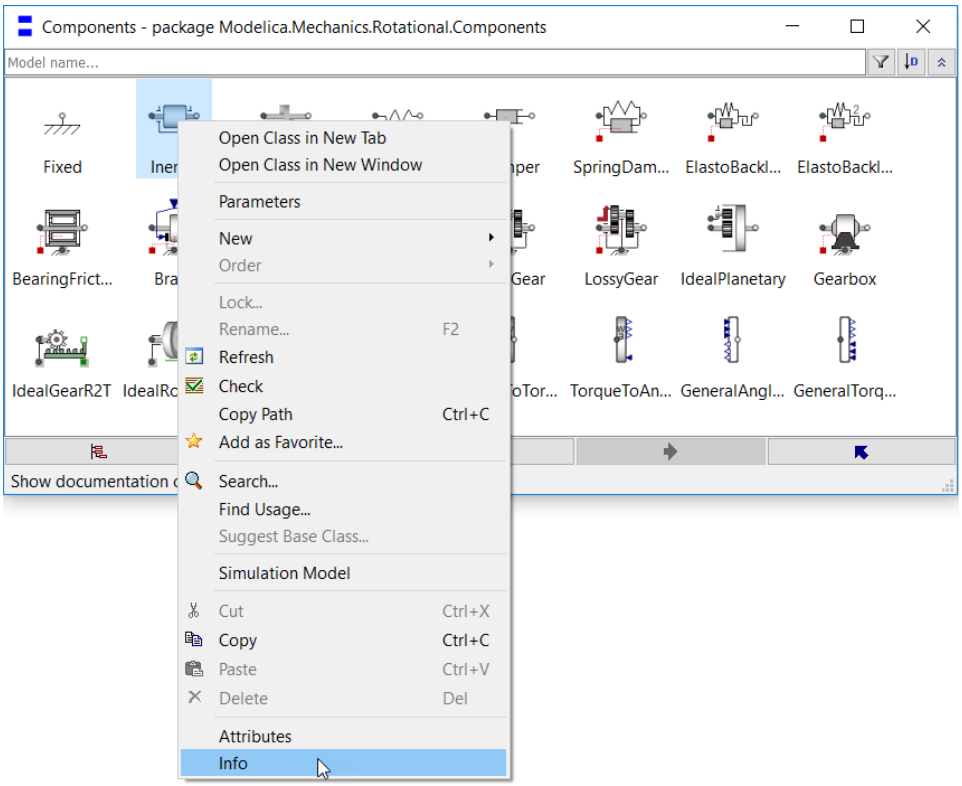
The Sensors library.



A quick scan of the Component library indicates that the model Inertia may be of interest for us.

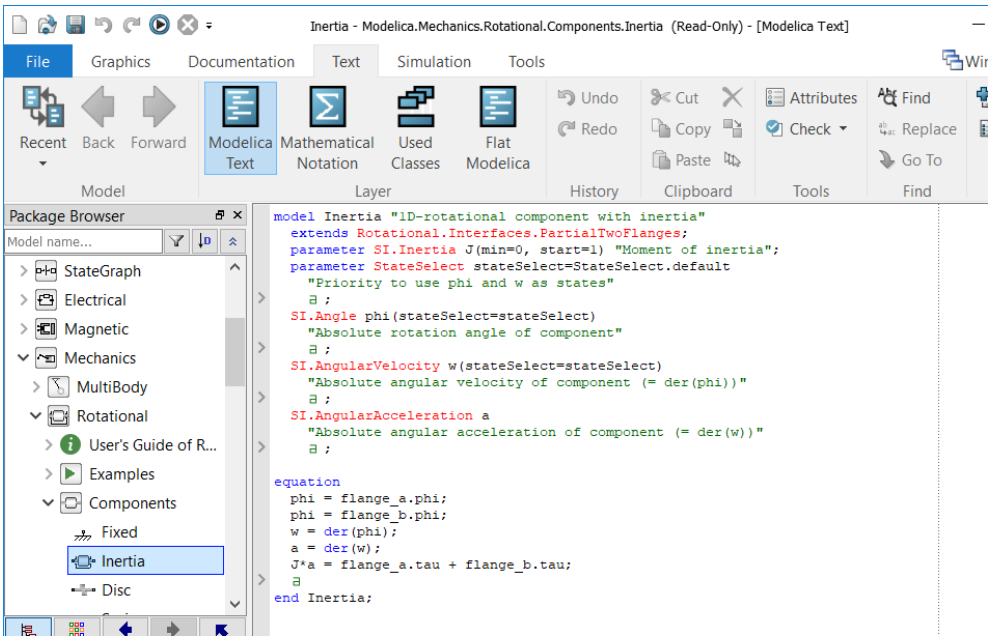
Right-click on Inertia for a context menu. Select **Info** to get documentation for the model.

The context menu for a component.



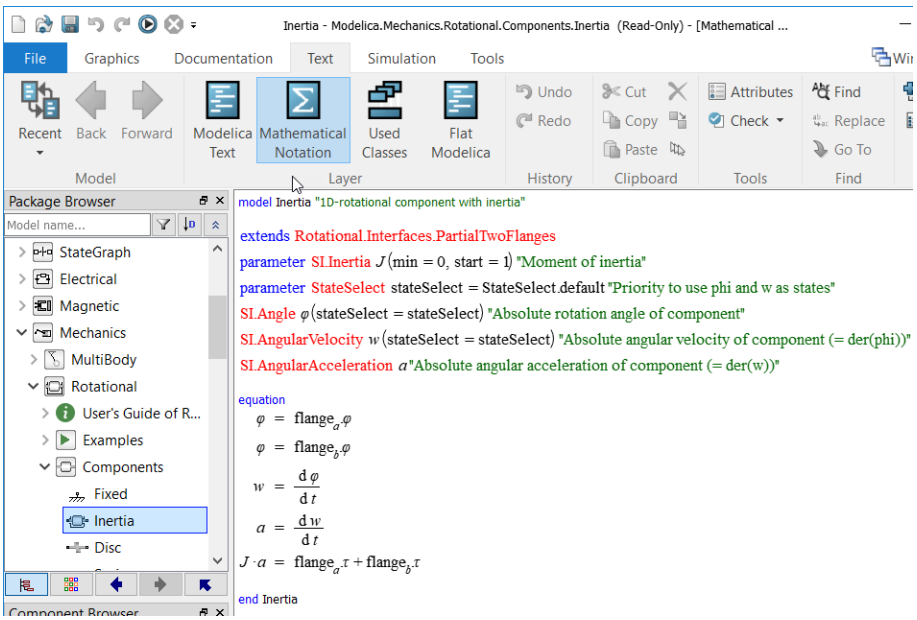
To get a model window for Inertia select **Open Class in New Window** in the same context menu. A window for the model Inertia is created. Switch to the **Modelica Text** representation, where you find Euler's equation as the last equation.

Mathematical definition of a rotational inertia.



If wanted, it is possible to look at the content with equations rendered with mathematical notation. To do that, select **Mathematical Notation** (like in the image below). The result will be:

Displaying using mathematical notation.

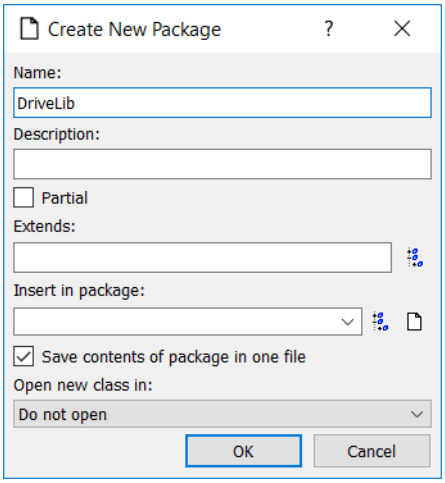


After this introduction of how to access model components and documentation of a library, we will continue by actually building a model for an electric DC motor. This task will give us more experience.

2.4.2 Creating a library for components

It is a good idea to insert developed components into a library. It is a good way to keep track of components and it supports also the drag and drop feature when you will use the models as components. Let us collect all developed components in a library called DriveLib. (To have a fresh Dymola to work with, you can select **File > Clear All** or restart Dymola.) Go to the Dymola window, and select **File > New > Package**. This will pop the dialog:

Creating a new Modelica package.



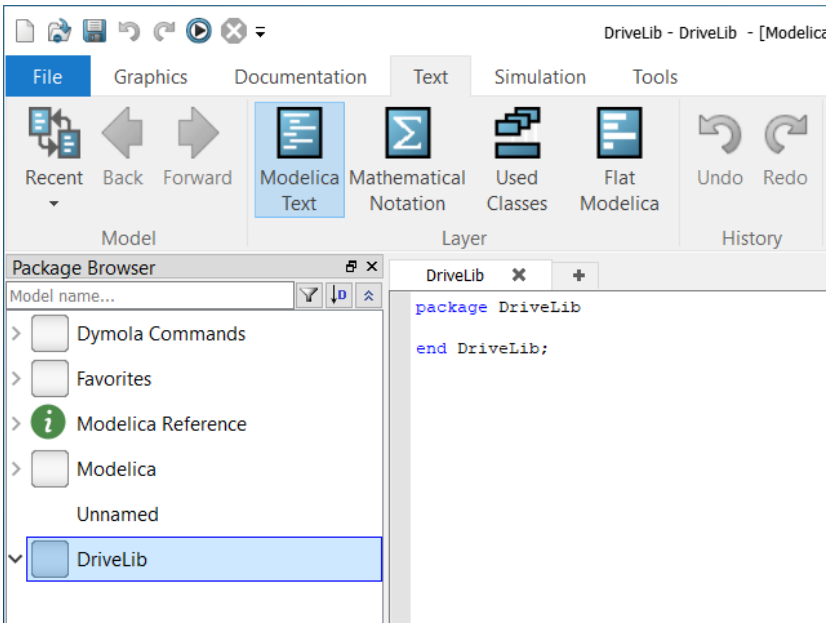
Enter **DriveLib** as the new name of the package and click **OK**.

A package DriveLib is created and made visible in the package browser (if scrolled).

Double-click the package to open it. The tab/layer shown in the package created (Modelica Text layer, Diagram layer etc.) depends on what layer was shown when creating the package. The layer shown can easily be changed by the tab buttons, or the buttons in the lower right part of the window. (Note that there are layer selections also inside the tabs, for example, selecting between Modelica text and Mathematical notation, as described above.)

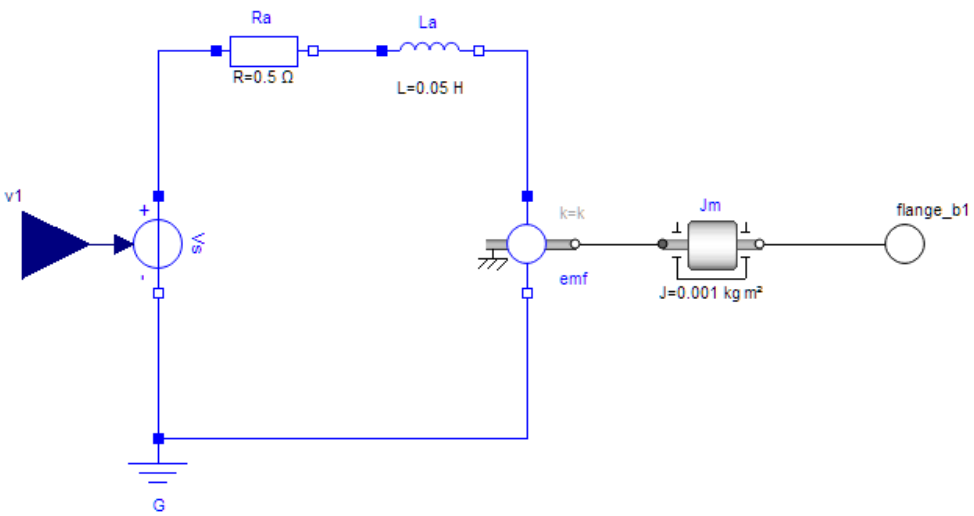
Select the **Text** tab (if not already selected) to see the Modelica representation, which at this stage just specifies a package with no contents (the reason that it is red in the package browser is that it has not yet been saved):

DriveLib created.



2.4.3 Creating a model for an electric DC motor

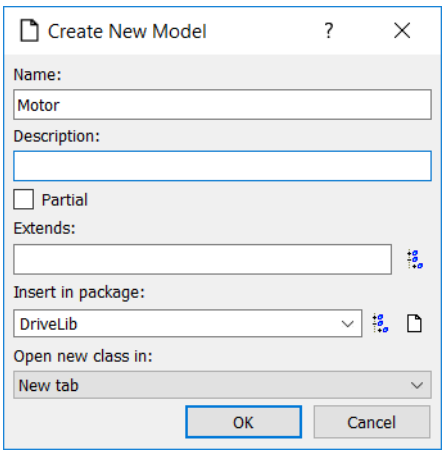
An electrical DC motor.



A model of the complexity indicated above will be developed for the electric DC motor. For simplicity, the voltage supply is included in the motor model. The model includes an ideal controlled voltage source. The electric part of the motor model includes armature resistance and armature inductance. The electromotive force (emf) transforms electrical energy into rotational mechanical energy. The mechanical part includes the mechanical inertia of the motor.

Let us start building the motor model. Select in the Dymola window **File > New > Model**. Enter **Motor** as name of the new model. To have the Motor model being a part of DriveLib, we need to enter **DriveLib** for **Insert in package**. This can be done in several ways. Dymola provides alternatives to be selected from and DriveLib is an available alternative. There are no other alternative because all other open packages are write protected. It is also possible to use the drag and drop feature and drag DriveLib into the slot. In the package browser, put the cursor on DriveLib and press the left mouse button. While keeping it pressed, drag the cursor to the slot for **Insert in package (optional)**, release the button and the text DriveLib will appear in the slot. It is also possible to browse for a package where to insert the model, clicking on the browser symbol to the right.

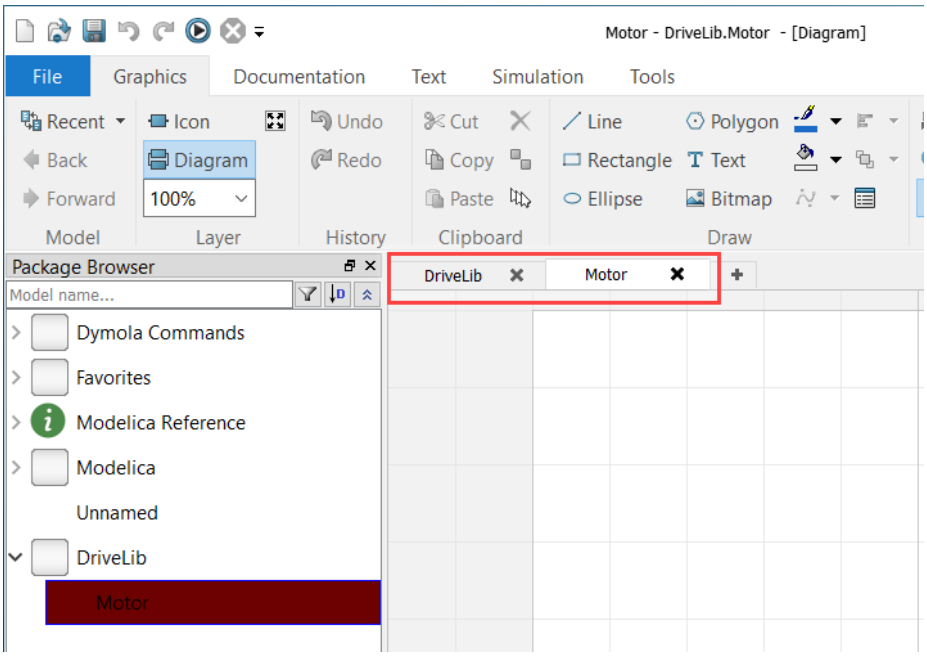
Inserting Motor in DriveLib.



Click **OK**.

The package browser shows that DriveLib has a component Motor as desired. The picture below shows the model when being in the **Graphics** tab (with the diagram layer displayed) - compare with the package created above. (To change what model layer to display, you can click the buttons in the lower right corner of the window.)

An empty Motor model.



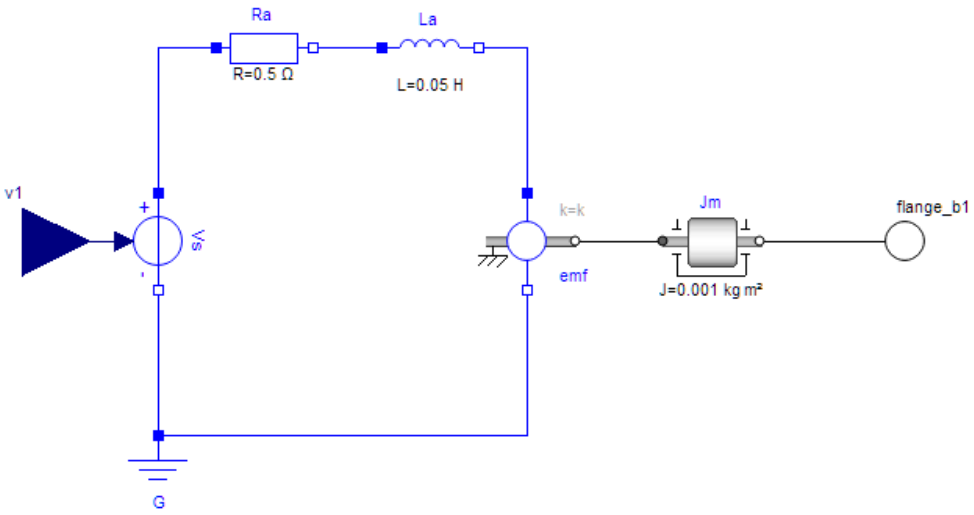
The model window now contains an empty Motor model. (The reason it is red in the package browser is that it is not yet saved. Note that the red color is only shown when the model is open.) The edit window has a gray frame and grid to indicate that the component is not write-protected. It is possible to toggle the grid using a button (the now blue one in the upper right of the image above).

Looking at the upper left corner of the Edit window (see red frame in the figure above) you can see two model tabs, one for the DriveLib that we created previously, and one for the Motor. The one for the Motor is the current active one, the Motor model is now open. You can click the tab you want to open.

Before building the motor model, please note that selecting a package in the package browser by just clicking on it does not mean that it is open and displayed in the edit window (and component browser). Yes, it is indicated in the package browser by blue (or red if not saved), but the one displayed stays the same (and is indicated by a blue frame in the package browser, the name in the window header, the top name in the component browser and the name in the bottom left in the window). By *double-clicking* (or right-clicking and selecting **Open Class**) on the package in the package browser the displayed package is changed, however.

We will now start building the motor model. To make it easier to follow the instructions, the result is displayed below:

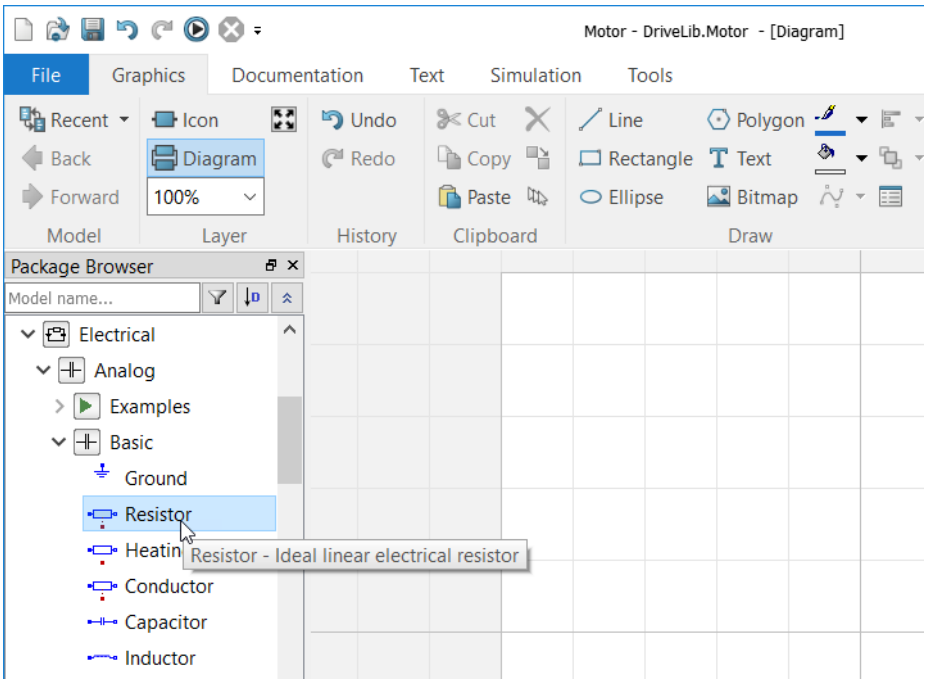
The finished motor model with all components.



We need a model component for a resistor. It can be found in Modelica.Electrical.Analog.Basic. The basic approach is to use drag and drop. You can drag and drop from the package browser or from a library window.

To drag from package browser, expand, in the package browser, in turn Modelica, Electrical, Analog and Basic. Note that title of the Dymola window is still DriveLib.Motor and also the component browser has DriveLib.Motor as top level to indicate that we are editing the motor model.

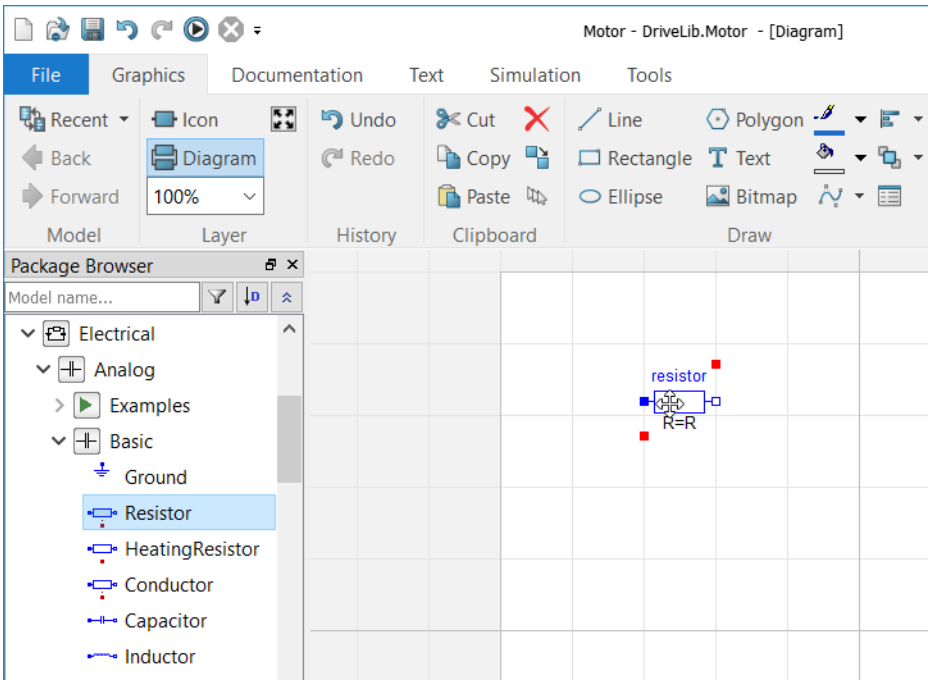
About to drag a resistor from the package browser.



(You can also drag from a library window.)

Drag a resistor from Basic to the Motor window and place it as shown below. The component browser displays that Motor has a component Resistor1.

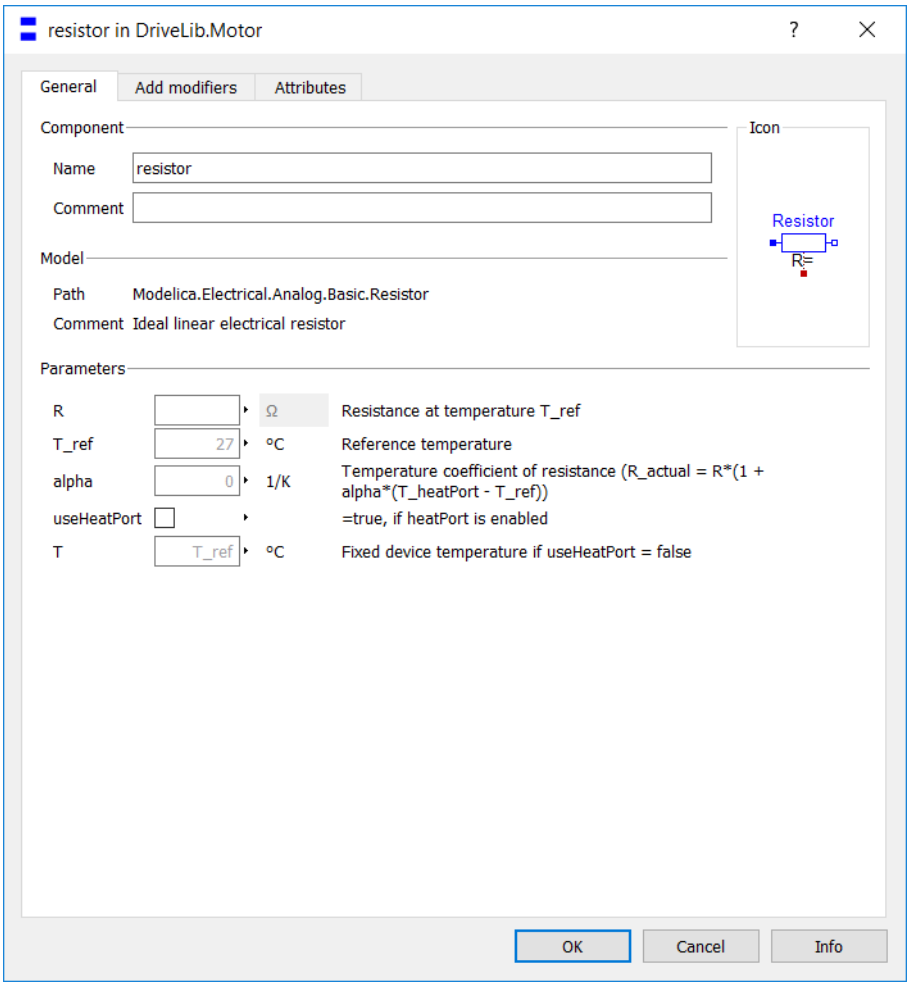
Inserting a resistor component.



When inserting a component, it is given an automatically generated name. The name may be changed in the parameter dialog. (The class name begins with an upper-case character; component instances begin with a lower-case character.) Double-click on the component, to get its parameter dialog[3]. The parameter dialog can also be reached by placing the cursor on the component, right-clicking and selecting **Parameters**.

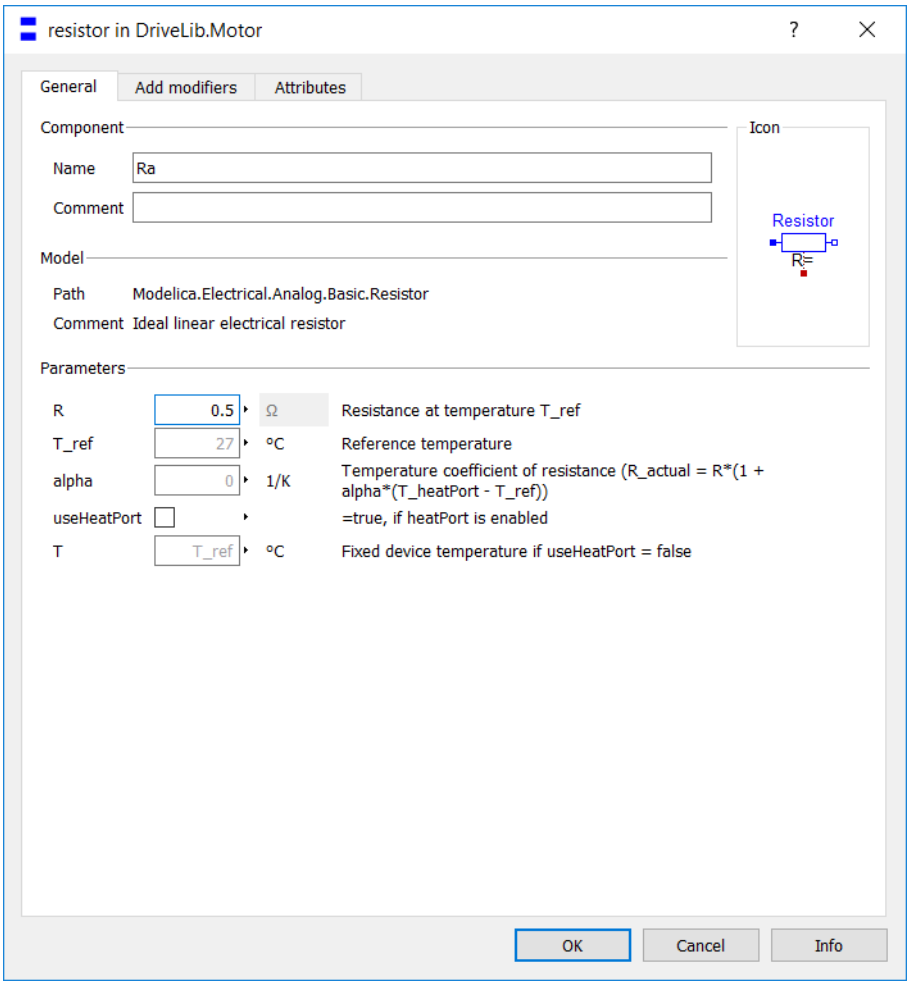
[3] If double-clicking is defined to perform some other action, you can use **Shift+double-click** to open the parameter dialog.

The parameter dialog of a resistor with default settings.



Change the component name to Ra. The parameter dialog allows setting of parameter values. To set the resistance parameter, R, select the value field of parameter R and input 0.5.

The parameter dialog of a resistor with new settings.



Click **OK**.

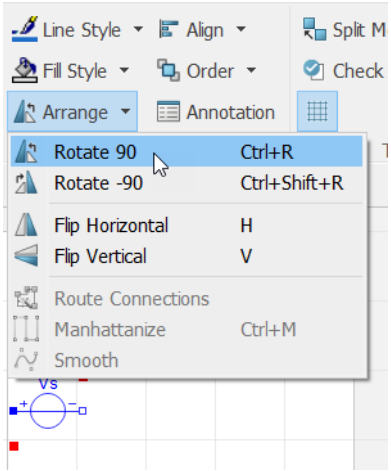
Similarly drag an inductor to the Motor window. Name it La and set the inductance, L, to 0.05.

Drag a ground component into the motor model. Name it **G**. The ground component is as important as in real electrical circuits. It defines the electrical potential to be zero at its connection point. As in the real world, never forget to ground an electrical circuit.

Drag an electromotive force, RotationalEMF, component into the motor model. Keep the name **emf**. (The component is grounded, meaning that the component has a support, where the support connector is fixed to the ground. Please keep in mind the difference from electrical grounding.)

A voltage source is to be found in Modelica.Electrical.Analog.Sources. Use the package browser (or a library window) to locate it. Select SignalVoltage and drag it to the model

window of Motor. Name it Vs. Let Vs be selected and use **Arrange > Rotate 90** to turn the signal input, Vs.inPort, from a top position to a left position:



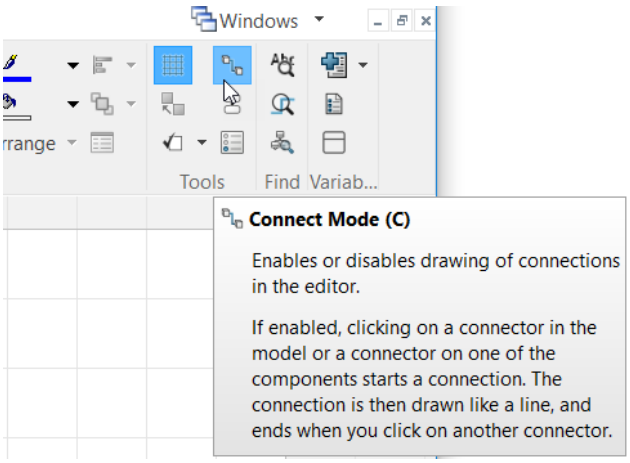
SignalVoltage produces, between its two electrical pins, p and n, a voltage difference, p.v-n.v, that is equal to the signal input. (This info can be displayed by right-clicking on the icon and selecting **Info**.). To get the proper sign we would like to have pin p in the top position. Since the pin p ("+") is the filled blue square, we must flip the component. To do that, use **Arrange > Flip Vertical**.

A rotating inertia component is to be found in Modelica.Mechanics.Rotational.Components. Drag and drop such an inertia component. Name it **Jm** and set the inertia parameter, J, to 0.001.

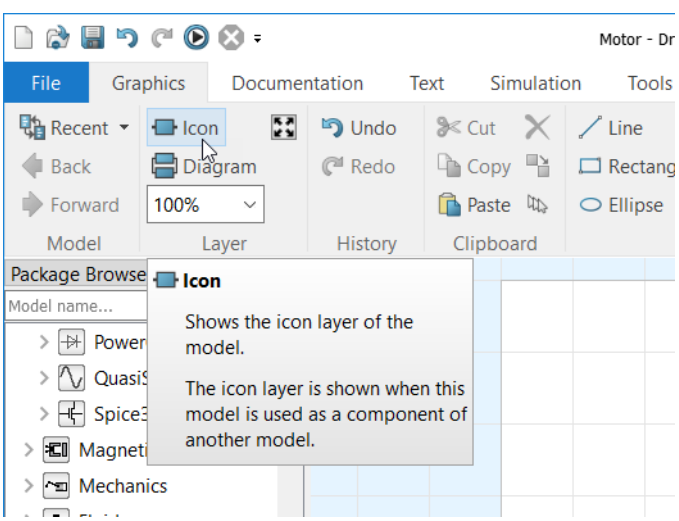
Now all model components are in place. Components are connected by drawing connections between connectors. Connect the resistor to the inductor by pointing at the right connector of the resistor (the small white square), press the left mouse button and keep it pressed while dragging it to the left connector of the inductor. The resistor and the inductor are now connected and the graphical result is a line between them. When connecting the voltage source and the resistor, break the line by clicking at an intermediate point. There is a possibility to obtain automatic Manhattanize of connections (non-endpoints of the connections are moved to make all line segments horizontal or vertical). Right-click the connection and select **Manhattanize**. Draw all connections. Note that we could have drawn a connection between two components as soon as we have the components and we have not to wait until all model components are in place. The points of the connectors can be moved, and new points can be inserted using the context menu of the connector.

Finally, we have to introduce a signal connector for the voltage control and a flange connector corresponding to the shaft of the motor so the motor can be connected to an environment. We would like to place the icon of the connectors at the border of the grid of the drawing pane, because the icon of a model component also includes the connectors. The connector inPort must be compatible with the connector of Vs.inPort. There is a simple way to get a connector inPort that is a clone of Vs.inPort. Start drawing a connection from Vs.inPort and go to the left until you reach the border of the grid. Then you double-click and select **Create Connector**

from the menu popped up. The connector flange_b is created in a similar way. If you would like to adjust the position of a connector it is easy to get into connect mode. This can be avoided by toggling the button **Toggle Connect Mode** (when the background is blue it is in connect mode):



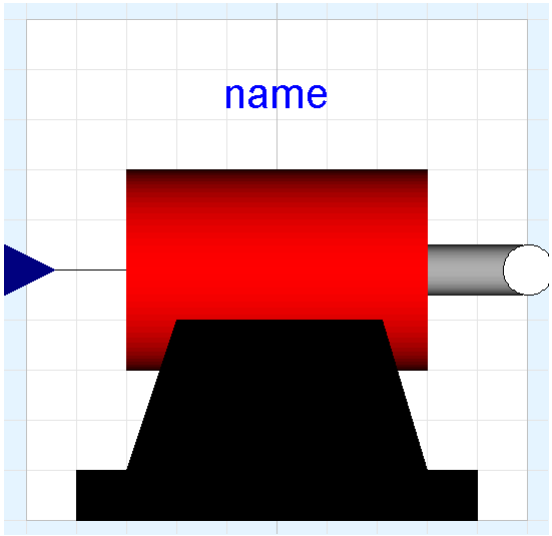
To go to the icon layer, click the **Icon** button:



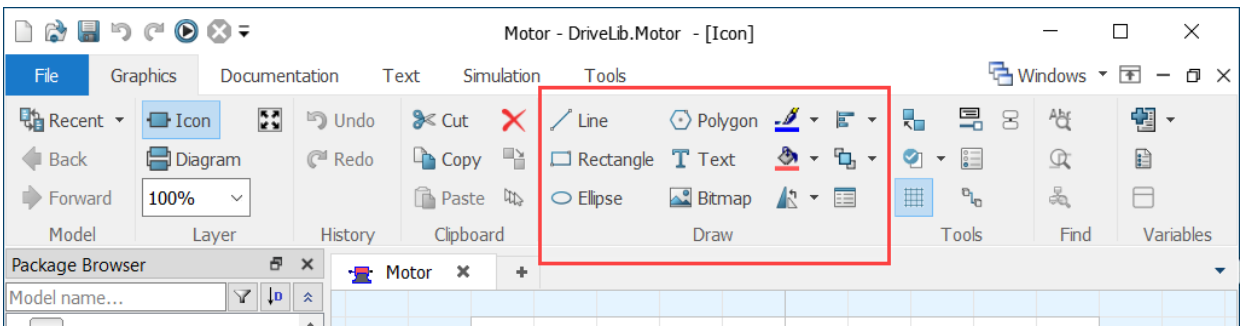
You will see the icons for the connectors. Note that the icon layer is indicated also by the area outside the normal working area is being blue instead of gray.

Let us draw an icon for the motor model. One design is shown below. (The thicker line to the right symbolizes the physical axis from the motor. It is a good idea to select that line and use the context menu **Order > Send to Back** to prevent any case where the axis seems to be lying outside the motor.)

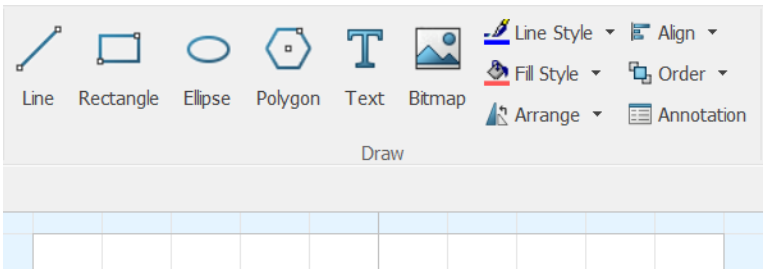
The icon of the electrical DC motor.



To draw it, we will use, in the **Graphics** tab, the **Draw** group for editing graphics. With the window size we use to be able to have the whole Dymola window in a picture where the texts are not too small, it looks the following:



But since you usually work with a maximized window, the group looks like:



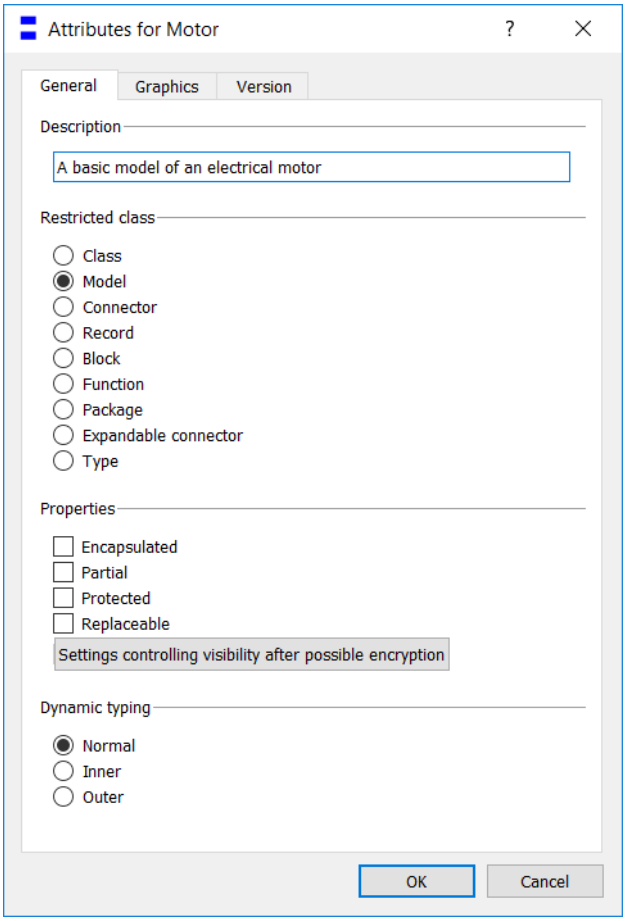
Start by drawing the big red cylinder (shaded rectangle); Click the **Rectangle** button and draw a rectangle. Let it be selected. Click on the arrow next to **Fill Style**  . Select **Color** and

then select a red color. To select the gradient, click once again on the arrow next to **Fill Style**. Select **Gradient > Horizontal**. Draw the rest of the parts using **Rectangle** or **Polygon** in an analogous way. To enter the text, click the **Text** button (the button labeled T) and draw a rectangle that is as long as the cylinder and one grid squares high. In the window prompt for the string enter `%name` and click **OK**. The % sign has the magic function that when the model is used, the actual component name will be displayed.

2.4.4 Documenting the model

We have now edited the icon and the diagram. It is also important to document the model. When creating the model, the dialog has a slot **Description**. It is possible to edit this afterwards. Go to the **Text** tab, and in the **Tools** group, select **Attributes** to open the dialog.

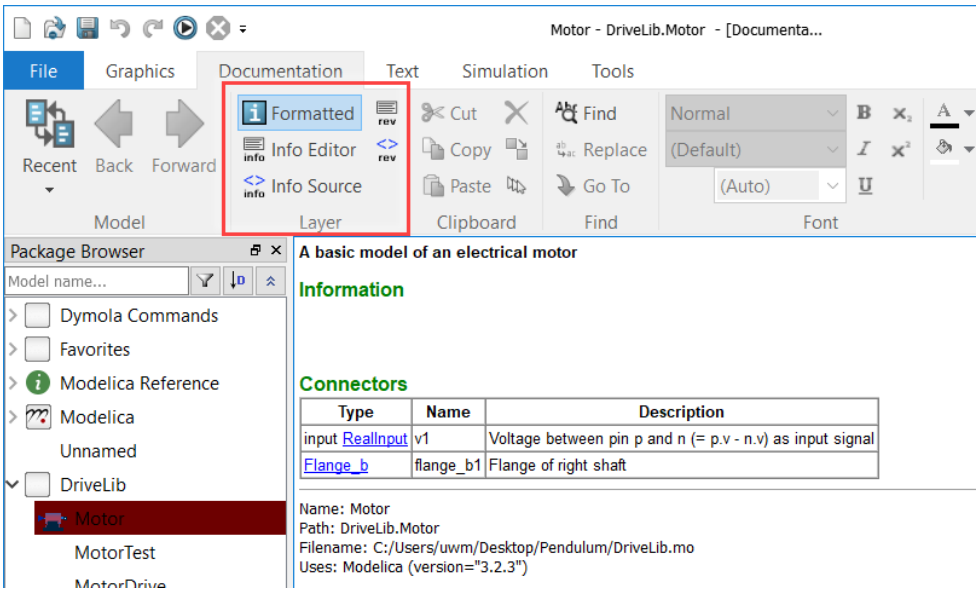
Model attributes.



Enter a description and click **OK**.

A longer documentation can be provided using the **Documentation** tab. Clicking that tab gives:

Documentation tab.

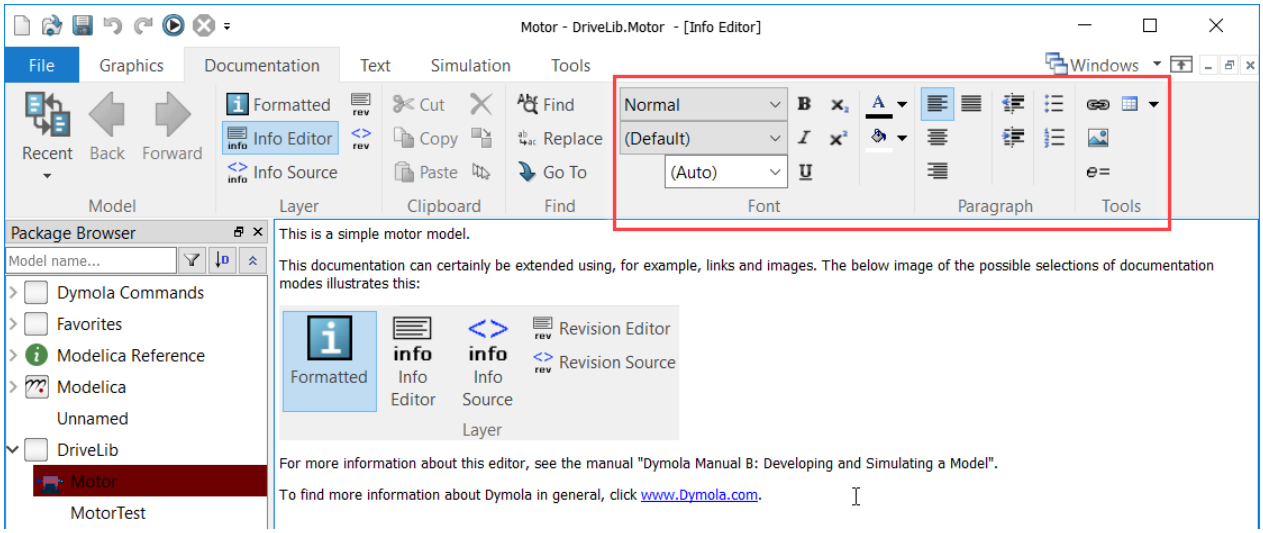


You can use different documentation modes. You select which mode from the Layer group in the Documentation tab (framed in red in the image above). This group looks the following when the Dymola window is maximized:



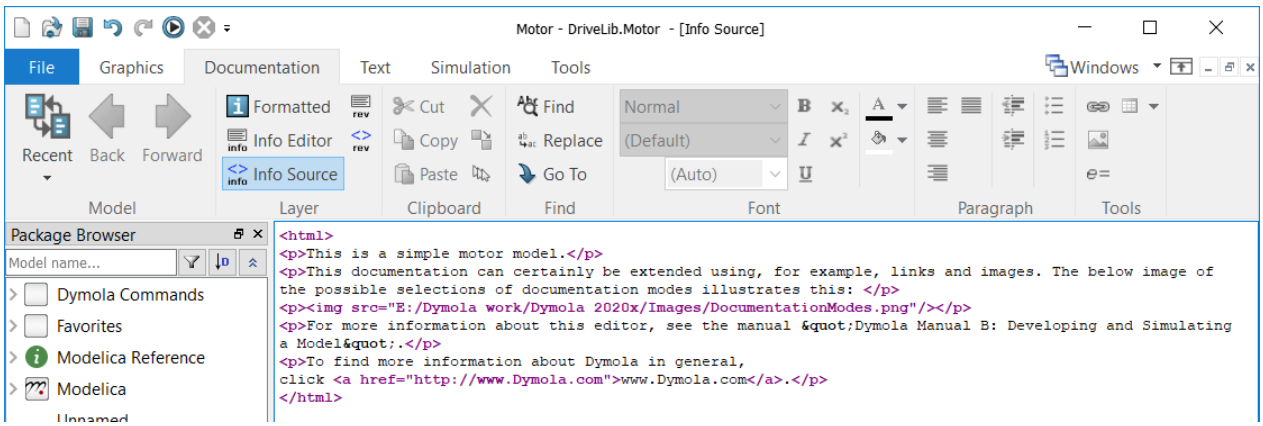
Documentation editor.

To enter a description, click, in the above group, **Info Editor**. You now have access to a documentation editor for the model/class with text-editing possibilities as well as link creation and image insertion (framed in red in the image below). After insertion of some text, an image and a link the result can look like:



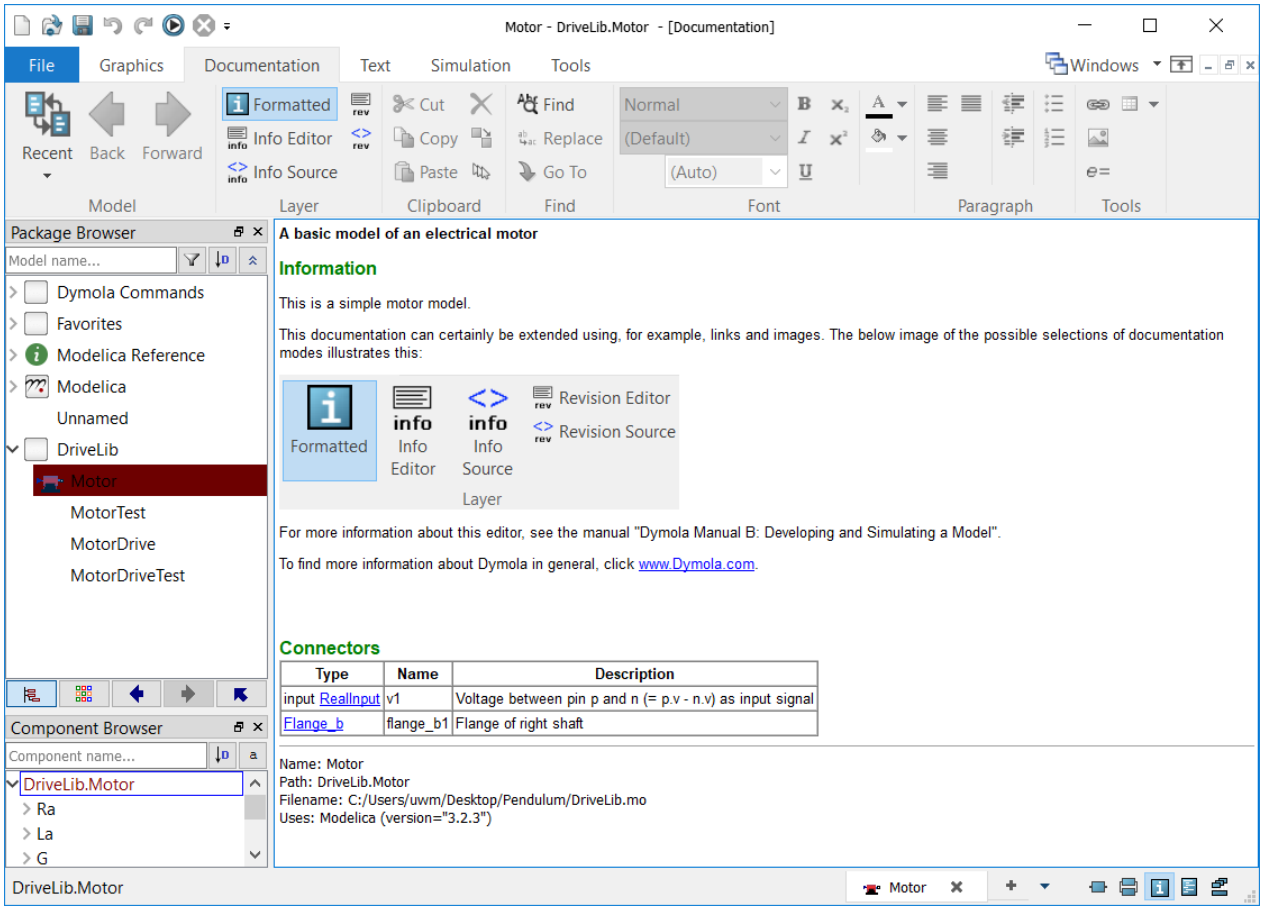
The corresponding html source code.

The created source code in html formatting can be viewed by clicking, in the Layer group, the command **Info Source** . The result is (some line breaks added to see all text):



The final result.

By clicking, in the Layer group, the command **Formatted**, the final result will be shown:



The link can now be clicked; by hovering on it the URL is shown in the status bar of Dymola main window (bottom of window; not shown in figure above).

The revision information can be edited in a similar way using **Revisions Editor**.

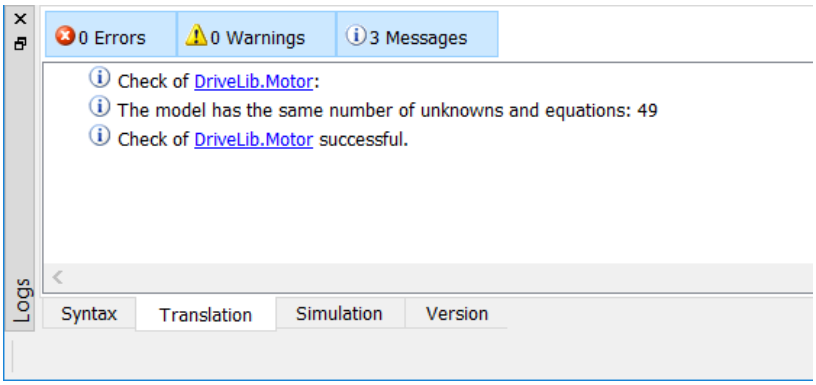
We have now created the model. Save it by the command **File > Save > Save**. Select **Save All**.

2.4.5 Testing the model



It is possible to check the model for syntactic semantic and structural errors. In the **Graphics** tab or the **Text** tab, click **Check**. Hopefully your model will pass the check and you will get the following message:

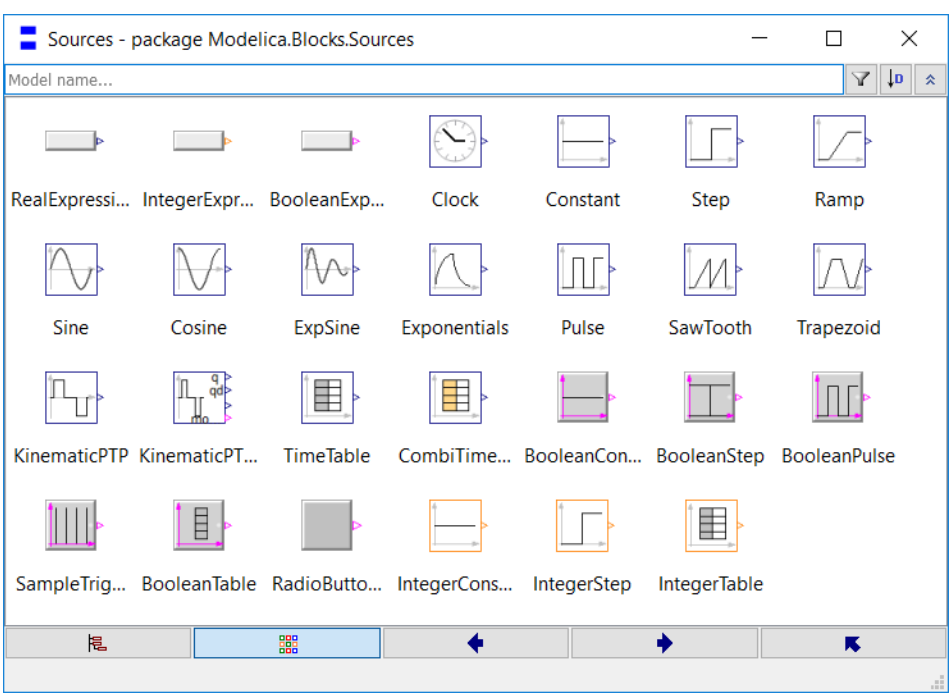
Checking the model.



The connector inPort defines the voltage reference, and should be defined for the complete model, but is viewed as a known input to the model.

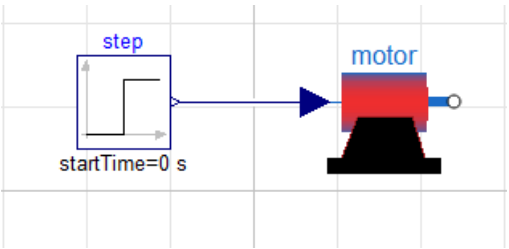
It is important to test all models carefully, because that eliminates tedious debugging later on. Let us connect the motor to a voltage source. Create a model called TestMotor and insert it into DriveLib. The easiest way is to use the command **File > New > Model**. It is good practice to keep testing models. Drag a Motor component from the package browser into the diagram layer of TestMotor. We need a source for the signal input to the motor. Signal sources are to be found in Modelica.Blocks.Sources.

Signal sources.



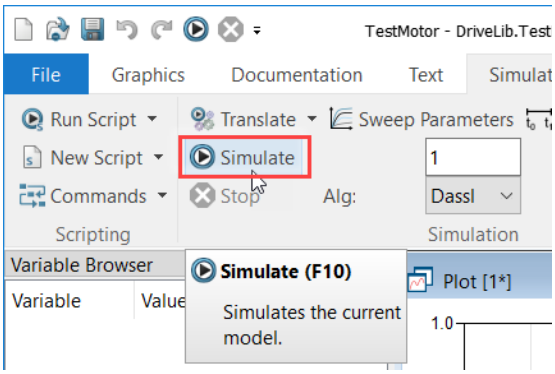
Drag, for example, over Step to the model window and connect it to the motor.

Model to test the motor.





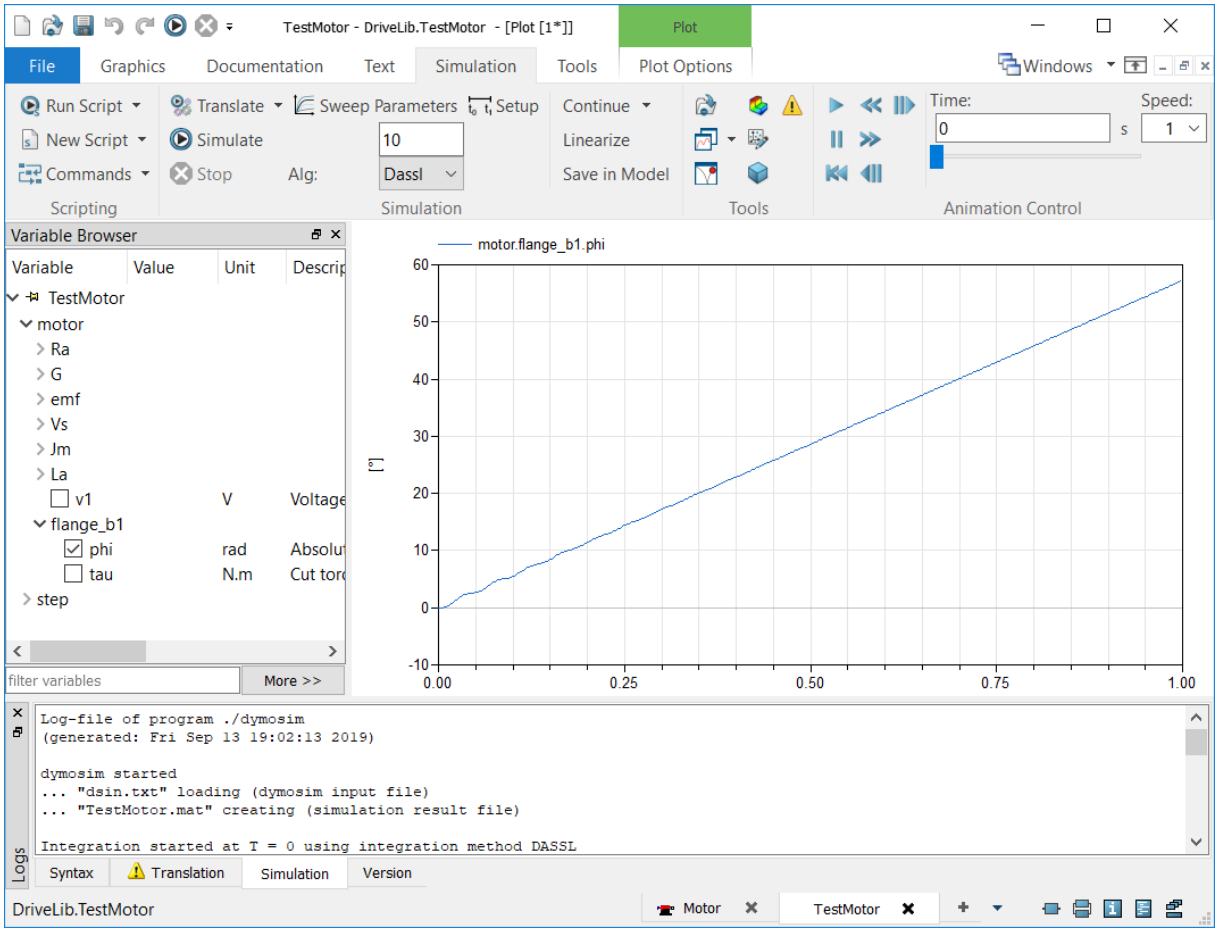
Now it is time to simulate. To display the commands related to simulation, click the **Simulation** tab. Now, to simulate the model, in the **Simulation** group, click the **Simulate** command:



Some warnings will be presented. Please see next section on how to get rid of them. However, they are warnings, so the simulation will work anyway.

Angular position.

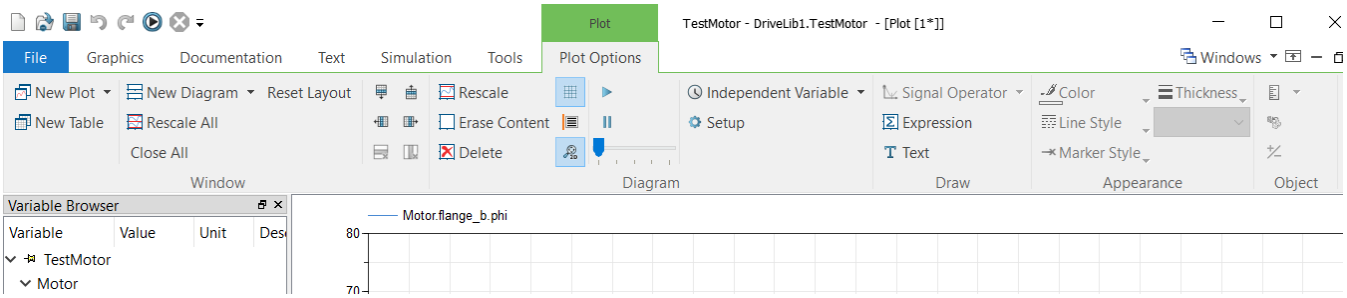
To inspect the result, we will first look at the angular position of the motor, motor.flange_b.phi. Open motor in the variable browser by clicking on the expand icon ▷ in front of motor. Open, in the same way, the flange_b and tick phi.



First, we may establish that a positive input signal makes angular position increase. The plot looks almost like a straight line. However, there are some wriggles in the beginning. Zoom in; use the mouse to stretch a rectangle over that portion of the curve you would like to see. We may also plot the angular velocity motor.Jm.w; there is an oscillation which dies out and the velocity becomes constant. There is much to be done to validate the model. However, model validation is out of the scope for this introduction to Dymola.

It is possible to show several curves in the same diagram. Simply tick the variables to be plotted. A curve is erased by ticking once more.

To display the commands for working with plots, click inside the plot window. That will open a **Plot Options** tab:



There are a number of commands here; as examples in the Window group you have commands for creating a new plot window (or table window), add a new diagram to the active plot, rescale the plots, and reset the layout. You also have commands for creating a 2D layout of plots. In the Diagram group you can, for example, erase the content of a plot or delete it.

There are numerous options for plots, you can, for example:

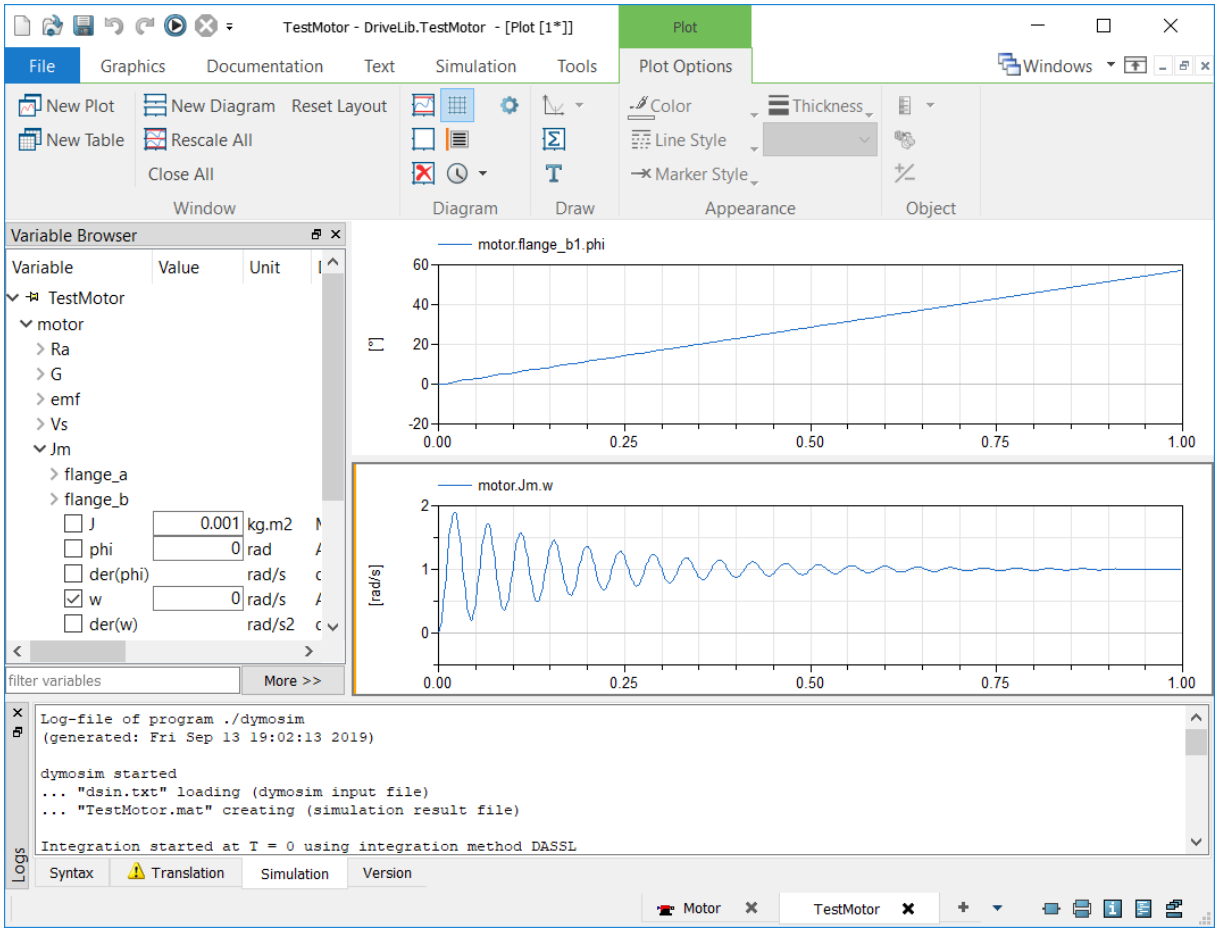
- Display several diagrams and work with them in the same plot window, in one column or as 2D layout.
- Display Boolean and enumeration signals, separated or overlapping.
- Display curves negated.
- Duplicate diagram to a new plot window.
- Display dynamic tooltips (and copy the values to clipboard if needed).
- Display tooltip for the legend to see more information about the corresponding signal.
- Display tooltip for the x-axis and the y-axis.
- Moving and zooming (one way of zooming is to just drag a rectangle over the area that is of interest).
- Apply time sliding window.
- Change time unit on x-axis.
- Change the display unit of a signal.
- Use a measurement cursor
- Drag and drop of curves between plots/diagrams.
- Drag and drop of curves between plots and tables.
- Lock plots and tables.
- Add a y-axis to the right (secondary axis).
- Assign different colors to curves from different simulation result files.
- Display signal operators.
- Plot general expressions.

- Plot 1D discretizations.
- Plot dependencies.
- Select multiple curves in a diagram (for e.g. copying the curve values to Excel).
- Display a table instead of curves, or create tables from curves.
- Display constant arrays as matrices.
- Apply color-coding to simulation result file tables.
- Plot parametric curves.
- Create scatter plots
- Change color, line/marker style and thickness of a signal.
- Plotting bar charts or area charts.
- Add titles and documentation to plot windows.
- Set diagram headings and axis titles and ranges.
- Insert and edit text objects in the plot.
- Change the appearance, layout and location of the legend.
- Display the component where the signal comes from in a new window containing the diagram layer.
- Go back to a previously displayed plot window (or any other window previously displayed.)
- Insert plots and its corresponding command in the command window.

For more information about these options, please see the chapter “Simulating a model”, section “Plot window interaction”.

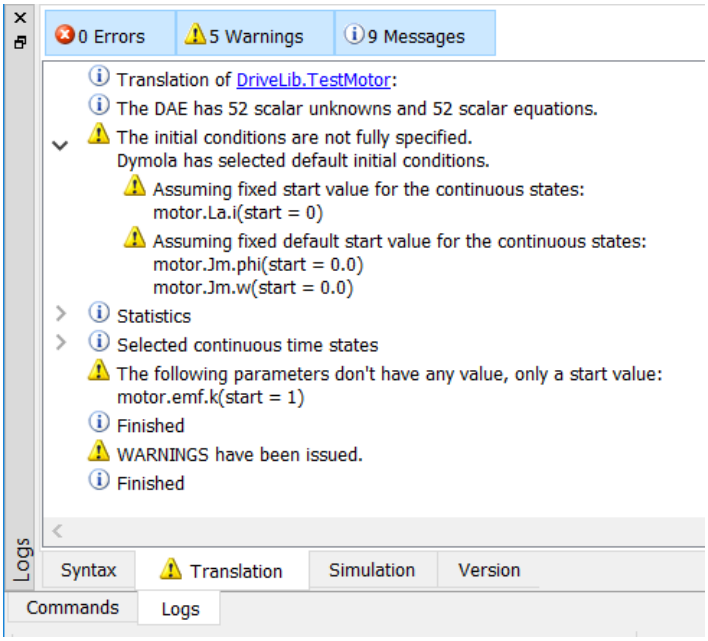
Angular velocity.

Using the option of adding a new diagram and ticking `motor.Jm.w` the result shown below is obtained.



2.4.6 Handling the warnings

When simulating the TestMotor, warnings can be seen by looking at the **Translation** tab of the Log window (you have to expand the first warning to see the submessages of that warning):



Two types of warnings are present for this example; a warning that initial conditions are not fully specified (at the top of the tab) and a warning that a parameter does not have any value, only a start value (at the bottom of the tab).

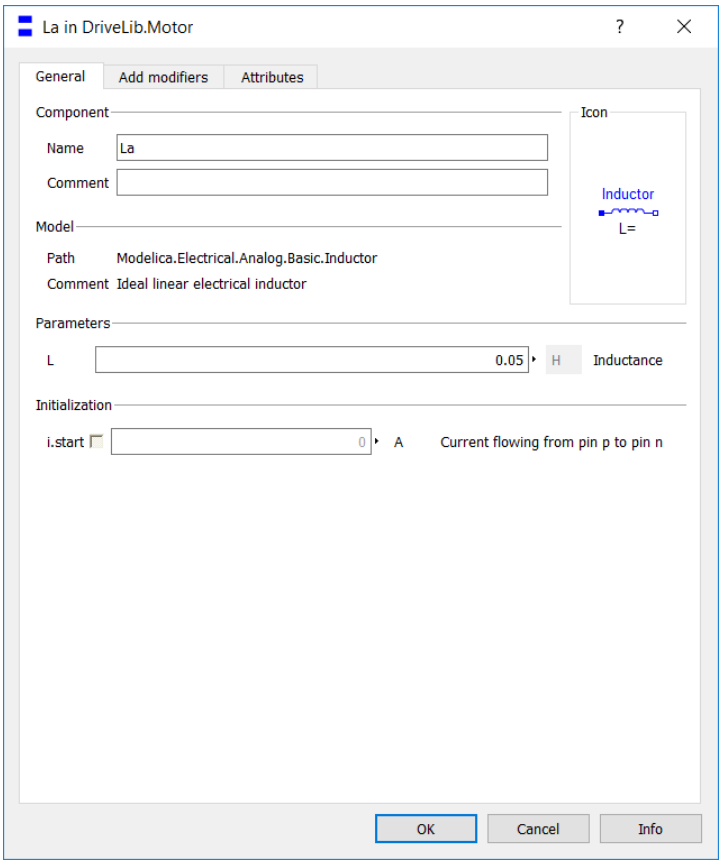
The first type of warning has been described previously, see section "Handling of warnings" on page 70. The difference here is that we can use the graphical user interface to set the variables to fixed and set the start values; we do not have to enter code.

Looking at the warnings, they all have to do with components inside the Motor model.

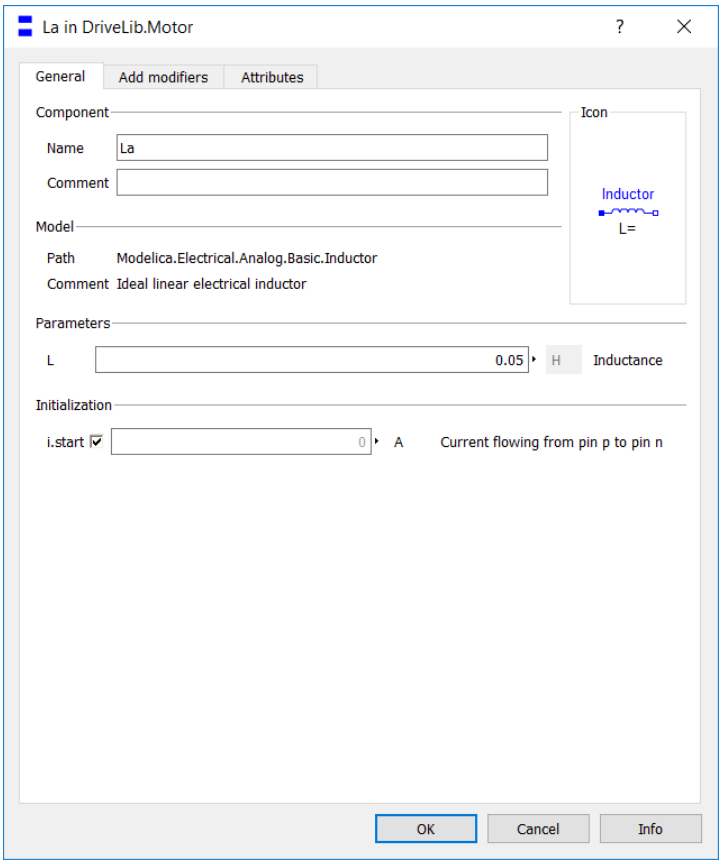
Click the **Graphics** tab to be able to work with the diagram layer. Then, by double-clicking on the Motor in the package browser, we return to this model. (You might also have to click on **Diagram** to display the diagram layer if you previously worked in the icon layer.)

The first variable of the upper warning section is located in the inductance component La. Double-clicking on this component in the diagram layer[4] will display the parameter dialog, where you find an Initialization section with i.start.

[4] We assume, as usual, the default setting of double-clicking will display the parameter dialog of the component; otherwise use **Shift+double-click**, or right-click the component and select **Parameters**.

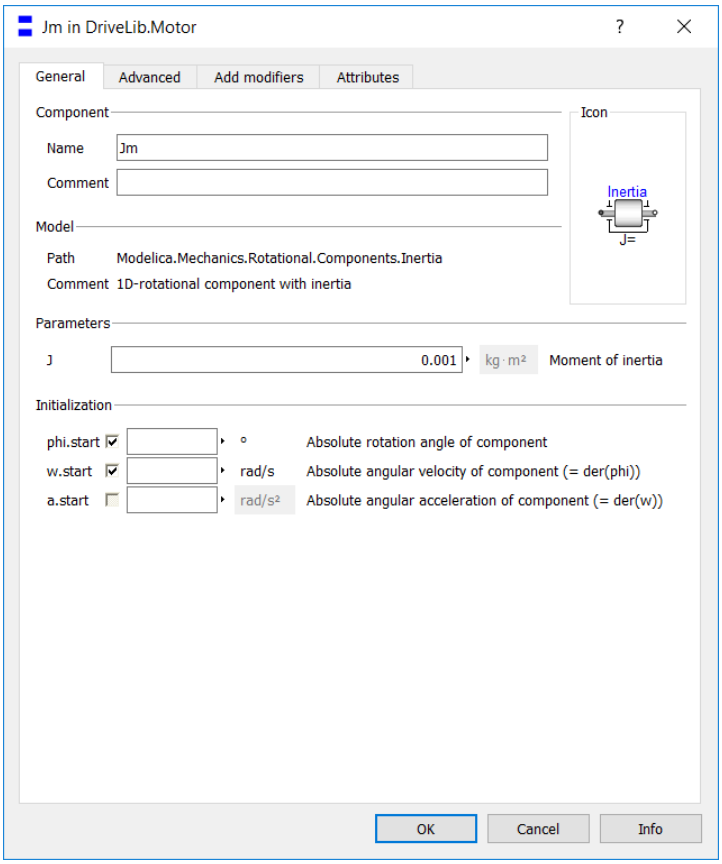


By clicking on the little square after i.start, a menu appears where we can select fixed to be **True: start-value is used to initialize**. The result will be:



Click **OK** to validate the change.

The two last warnings in this part have to do with the inertia Jm. Double-clicking on this component brings up the parameter dialog for it, and in the Initialization section phi and w are found. Like for the inductor, by clicking on the little square in front of each of these variables, a menu pops where we can select fixed=true. The result will be:



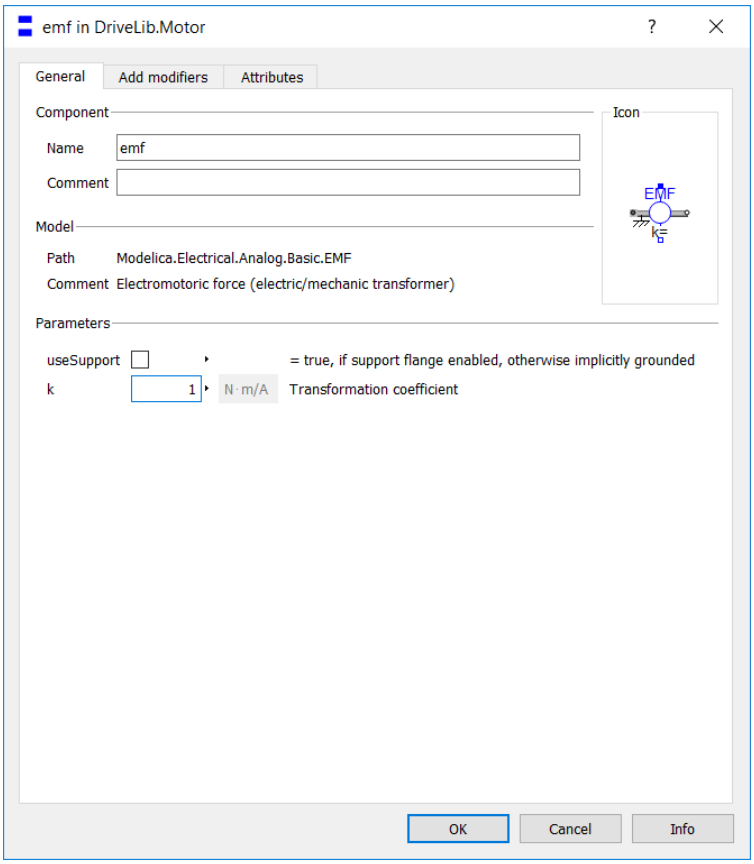
If we only do these changes, we will still have warnings that no start values are defined for these parameters (compare with the inductor, where there is a start value defined). To fix this, enter 0 for both parameters in the input box. Click **OK** to confirm the changes.

If simulated again, the upper warnings in the Translation tab will not be present anymore.

The lower warning states that the parameter motor.emf.k(start=1) only has a start value (start=1) but no value.

The warning implies that this parameter value is generic and that it ought to be set by the user. (More about such warnings can be read in the chapter “Developing a model”, section “Advanced model editing”, sub-section “Parameters, variables and constants”.)

The way to handle this is to double-click on the emf component and set the value of k to 1.



Click **OK** to confirm the change.

When simulating again, no warnings will be given.

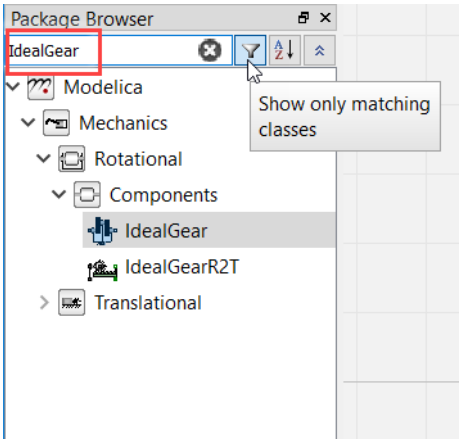
2.4.7 Creating a model for the motor drive

The task was to build a model for a motor drive and it ought now to be a simple task to complete the model. We will just give a few hints. Note that the full names of the components are given in the component browser at the lower left.

Give the model the name MotorDrive – we will refer to that name later on. Put it in DriveLib.

To generate the position error, you may use the model component Modelica.Blocks.-Math.Feedback. For the controller, there is Modelica.Blocks.Continuous.PID.

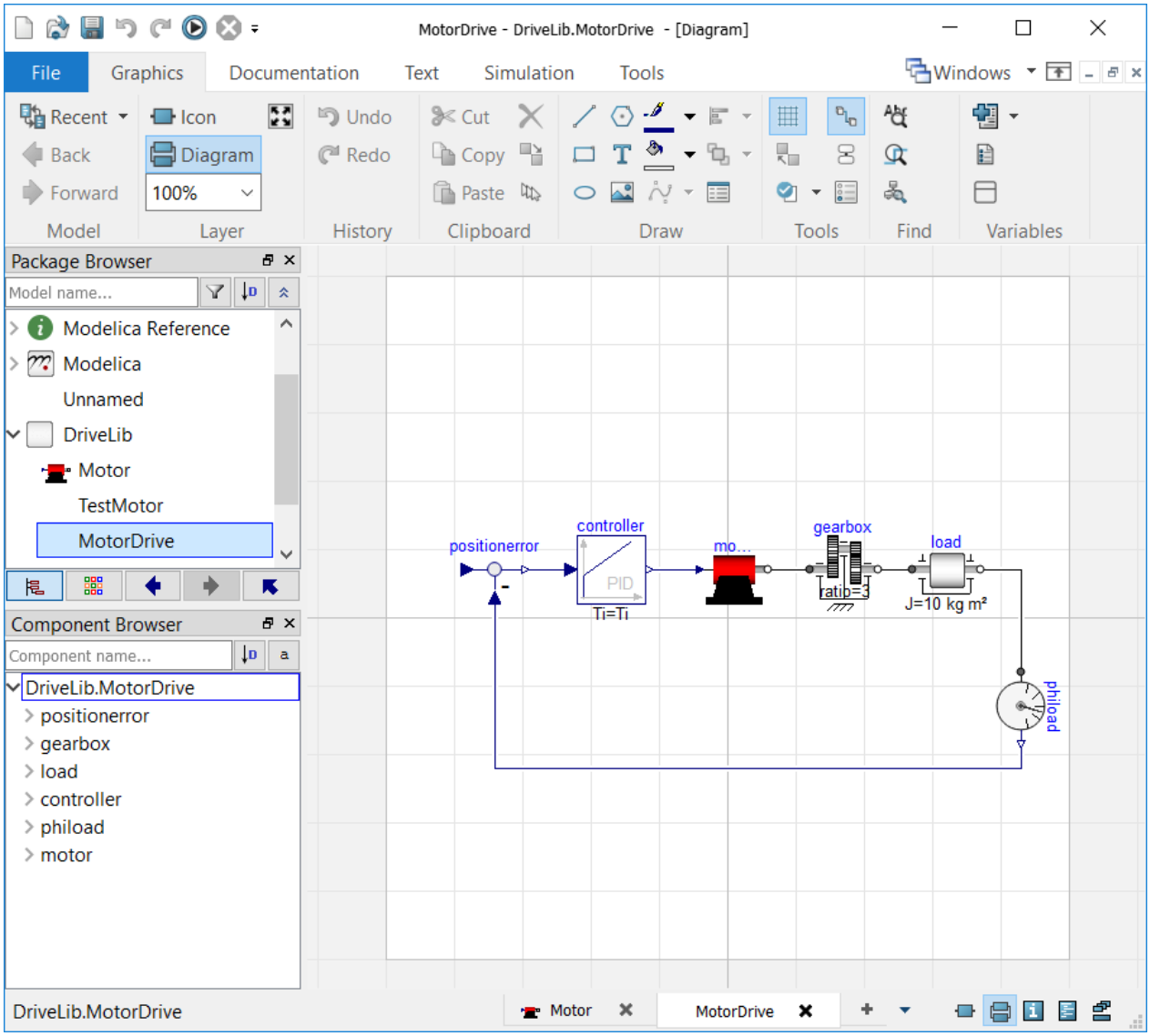
In this simple example we can select the component Modelica.Mechanics. -Rotational.Components.IdealGear as gearbox. Note the tools for facilitating finding a model in the package browser, for example, typing IdealGear in the search text field and activating the filter **Show only matching classes** gives:



(The button after the filter button is a toggle sorting button that you can use to, for example, sort in alphabetical order.)

For the meaning of ratio for the gearbox model please consult **Info** for the model. Set ratio to 3 as indicated. It means that the motor load rotates 3 times slower than the DC motor. The library Modelica.Mechanics. Rotational.Sensors contains a sensor for angles.

Inserting a load was dealt with when building the motor model. Set J=10 kgm2.



To test the model MotorDrive for normal operation, we need to define a reference for the position. This can be done in different ways. A simple approach is to add a signal source directly to MotorDrive. However, we may like to use MotorDrive also for other purposes. If we would like to use the drive as a component we could add a connector for the reference as we did for the electric DC motor model. However, here we will take the opportunity to show another useful way, namely use of extends. We will develop a new class, say MotorDriveTest, which extends MotorDrive. Select MotorDrive in the package browser, right-click and select **New > Extend From...** in the context menu. This gives the same dialog as **File > New > Model**, but with several fields filled out. (It extends from MotorDrive and is inserted in the same package, DriveLib.) Enter MotorDriveTest as the name of the model. Click **OK**. The

result is a model window, where the diagram layer looks exactly like that of MotorDrive. However, the components cannot be edited. Try to move or delete a component. It has no effect. (It is valuable to give some thoughts comparing “extend” to “copy/paste+add”. The former strategy allows a far cleaner structure, where an object can be reused in a number of locations. If that object is further refined, all objects extended from that one will gain from that refinement.)

“Smart connect.”

Now we want to drag a component Step from Modelica.Blocks.Sources to the diagram and connect it. This is a good opportunity to show another feature: Drag the component Step into the diagram in such a way that the output connector of the Step hits the input connector of the positionerror component. You will see a quick green circular area around the “overlapping” connectors – this means they have been automatically connected – this is part of the “smart connect” feature. When you now move the Step you will see the connection line created.

Check and save the model.

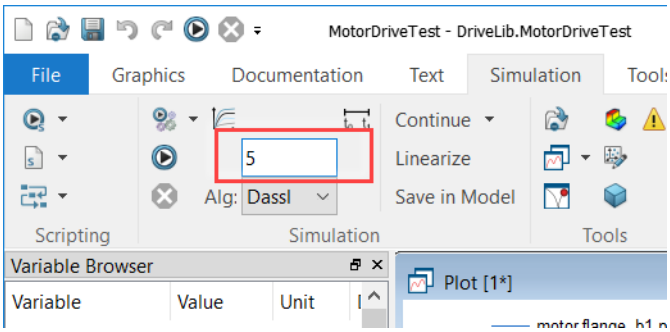


A model can be used for different tasks. One is tuning the controller manually. Click on the **Simulation** tab. Translate the model MotorDriveTest by, in the **Simulation** group, clicking **Translate**. (Also delete one of the plot windows and delete the previous curve.) Looking at the variable browser, we can expand the controller and note the common parameters of the PID controller: k, Ti, Td and Nd.

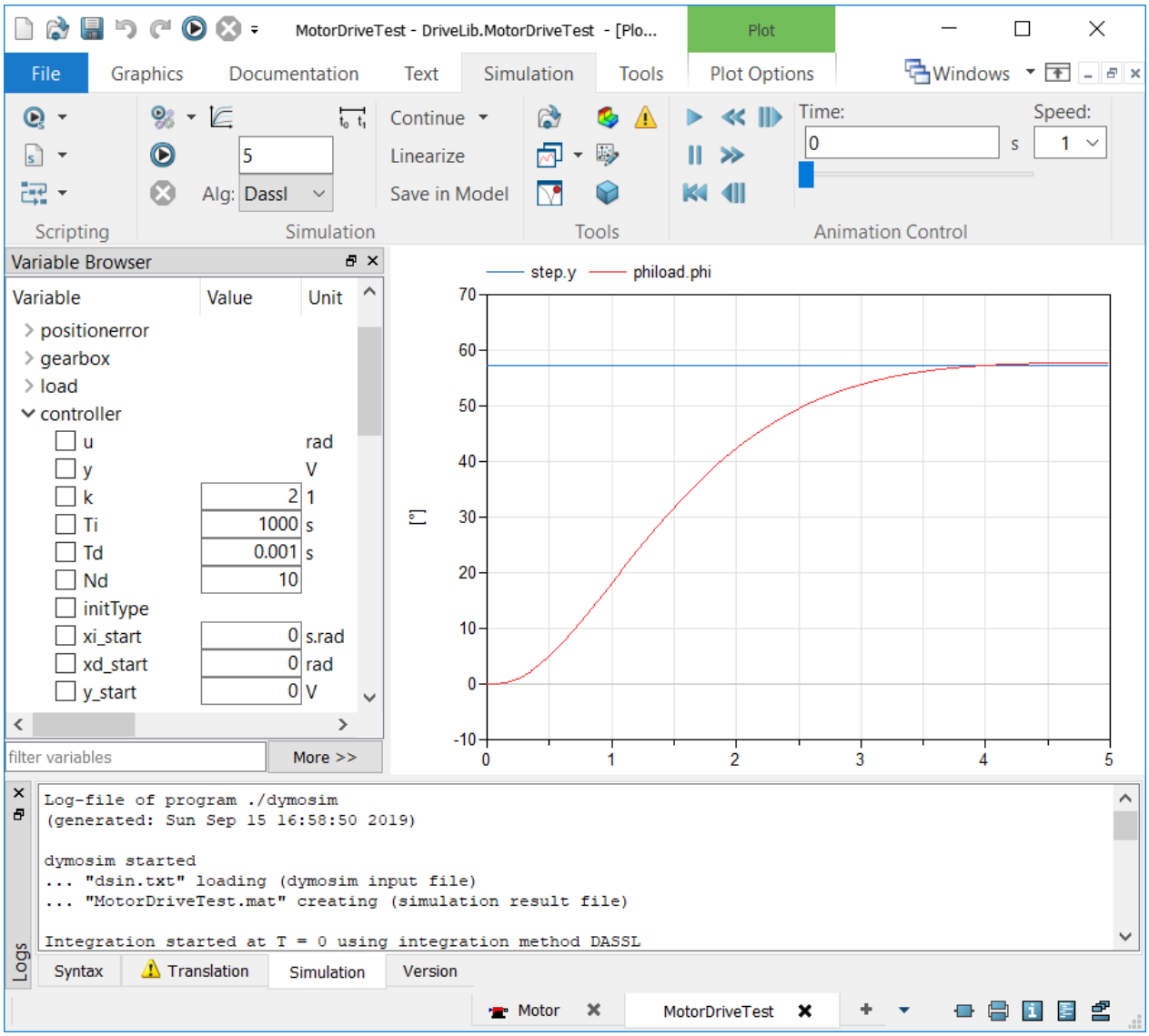
(There will be some warnings when translating. We will take care of those later.)

Tuning the controller.

One way to tune a PID-controller is by first disabling the integrator and derivative parts. This is achieved by setting a large value for Ti and a small value for Td. Set k=2, Ti=1000, and Td=0.001 and simulate for 5 seconds. The stop time can be set in the simulation tab directly:



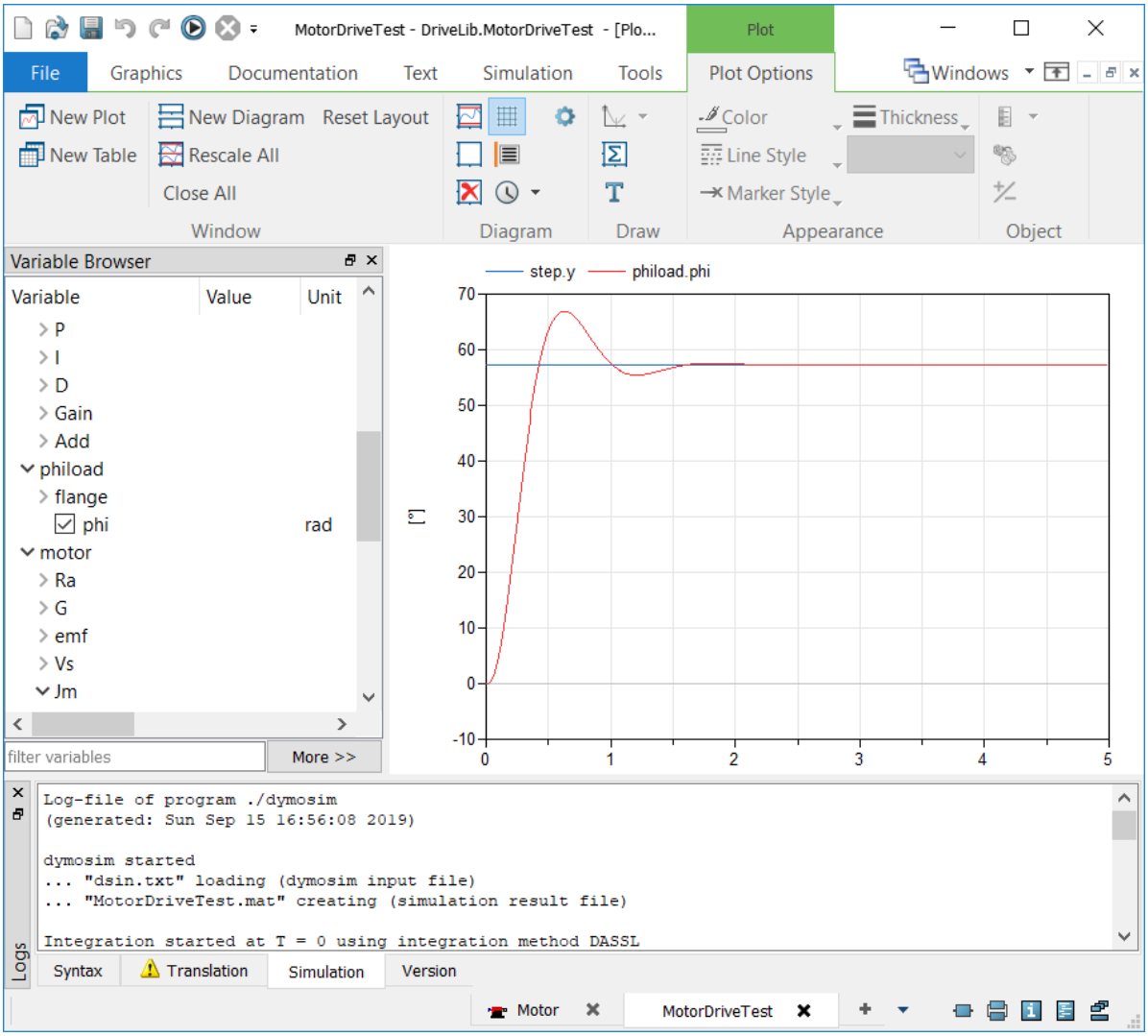
As can be observed by plotting the signals and step.y and phiload.phi, the step response is very slow:



Increase k to find out what happens. Make sure to also investigate the magnitude of the control signal, controller.y.

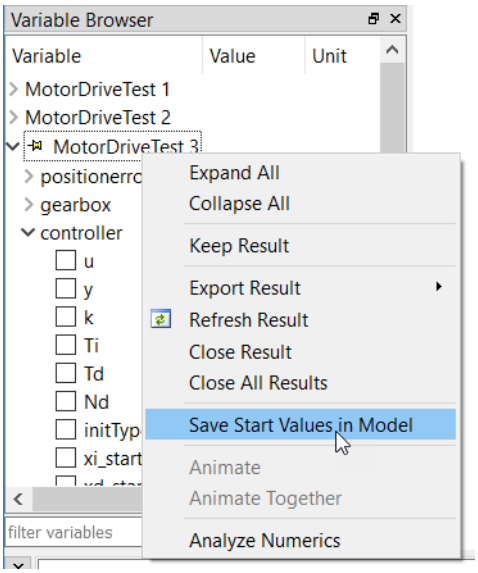
Tuning result.

The interested reader may next proceed to tune the controller parameters to obtain a nice step response. For example, aim at a rise time around 0.4 seconds, a settling time around 1 second, maximum overshoot of 10 percent, and a maximum control signal magnitude of 100. Enforce the last constraint by adding a Modelica.Blocks.Nonlinear.Limiter component between the controller and the motor in the MotorDrive model. Set uMax=100 and uMin= -100 for the limiter. (We will not show this component in the code.) The result before adding the limiter component might be something like:

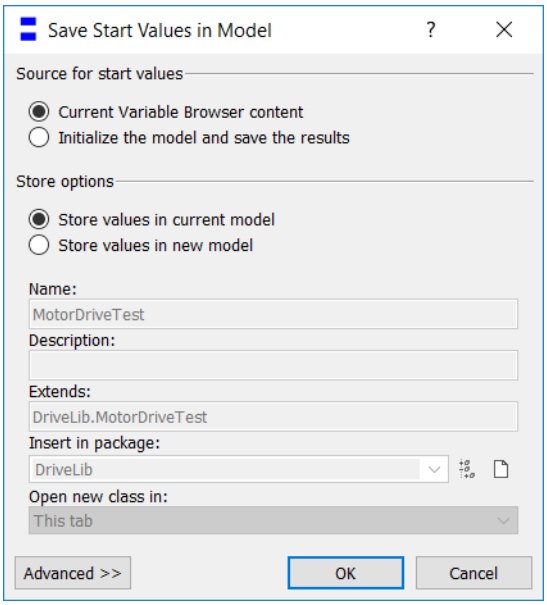


Saving changed start values to the model

Having tuned values when simulating like the interested reader above, it is a big advantage to be able to directly save them in the model (changing the Modelica code). This can be done by right-clicking on the result file in the variable browser and selecting **Save Start Values in Model** like below:



The following dialog appears:



If you just click **OK** with the default selection **Current Variable Browser content**, current changes made in the variable browser to parameters, initial values and guess values are saved in the current model. (If you instead select **Store values in new model**, you can change the name etc. and when you click **OK** the current model will extend to a new model with the name you have specified, and the changed values mentioned will be changed in *that* model. For the

idea of extends, see above, when you tested the MotorDrive by extending it to MotorDriveTest.)

It is also possible to have the model initialized, and the result read to the variable browser before saving, by selecting **Initialize the model and save the results**. For more information about the command **Save Start Values in Model** in general, see the index entry *"start values : save to model"* in the index in the end the manual.

In this example, just click **OK** to save the start values in the current model.

Handling the warnings

Simulating once more after having saved the start values in the model, some warnings about not having specified start values for Ti and Td in the controller have disappeared,

However, there will be one warning when simulating. In order to make the variable `controller.D.x(start=controller.D.x_start)` fixed, you have to double-click on the MotorDrive in the package browser, then select the controller, right-click and select **Show component**. Now you can double-click on the D component to see the parameter dialog, and since no pure x is available you have to select the **Add modifier** tab, enter `x(fixed=true)` and press **OK**.

2.4.8 Parameter expressions

Modelica supports parameter propagation and parameter expressions, which means that a parameter can be expressed in terms of others. Assume that the load is a homogeneous cylinder rotating long its axis and we would to have its mass, m, and radius, r, as primary parameters on the top level. The inertia is

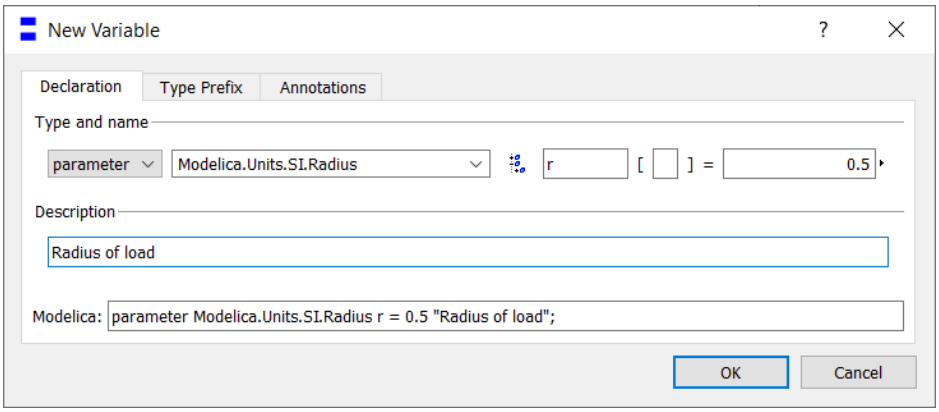
```
J = 0.5 * m * r^2
```

We need to declare m and r in MotorDrive. Click the **Text** tab, and then double-click MotorDrive in the package browser to open that model. By default, the Modelica text editor should be selected, but if not, click **Modelica** to select it.

The parameters and variables are more than real numbers. They are physical quantities. The Modelica standard library provides type declarations for many physical quantities. Expand Modelica.Units.SI and *select* Modelica.Units.SI by clicking on it in the package browser (note, do not open it; MotorDrive should still be open). For the parameter r, which is a radius, it is natural to declare it as being of type Radius. To find it, press r on the keyboard; the browser goes to the first component starting with r/R. If it is not the desired one, press r once again and so on to find it. (You can also search for it by typing radius in the search field of the package browser.)

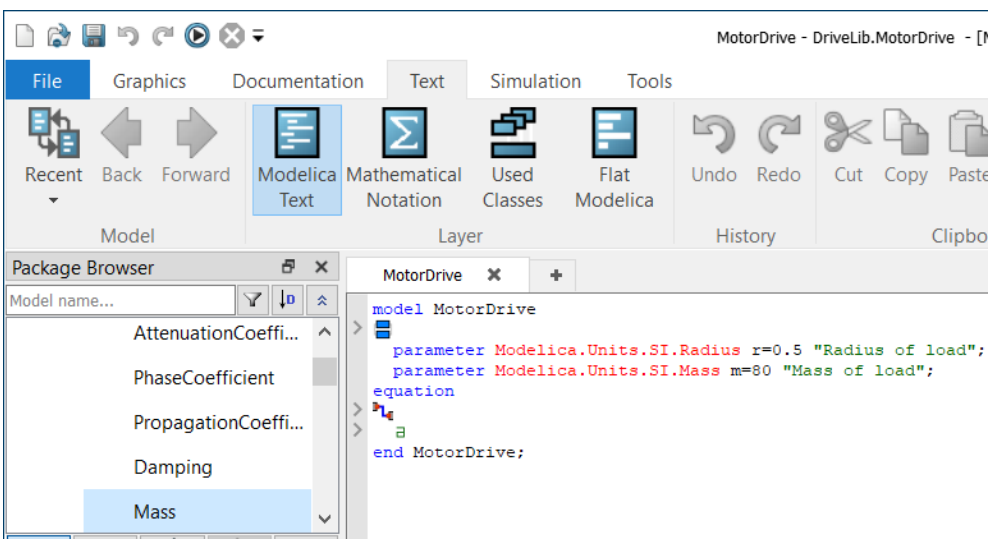
When you have found Radius, drag it to the component browser below. The choice to add a component is pre-selected. Click **OK**. A menu to declare a variable appears. Complete the declaration (do not forget to change to a parameter):

Declaration of parameter r.



Click **OK** and the text appearing in the bottom row is inserted into the Modelica text window. The parameter m (being 80 kg if the resulting inertia should be the same as previously) is defined in an analogue way. Note however that it might be necessary to drag it to the end of the component browser, there are some replacement alternatives for some present components in the component browser that are not what we want now.

Parameter declarations added to motor drive.



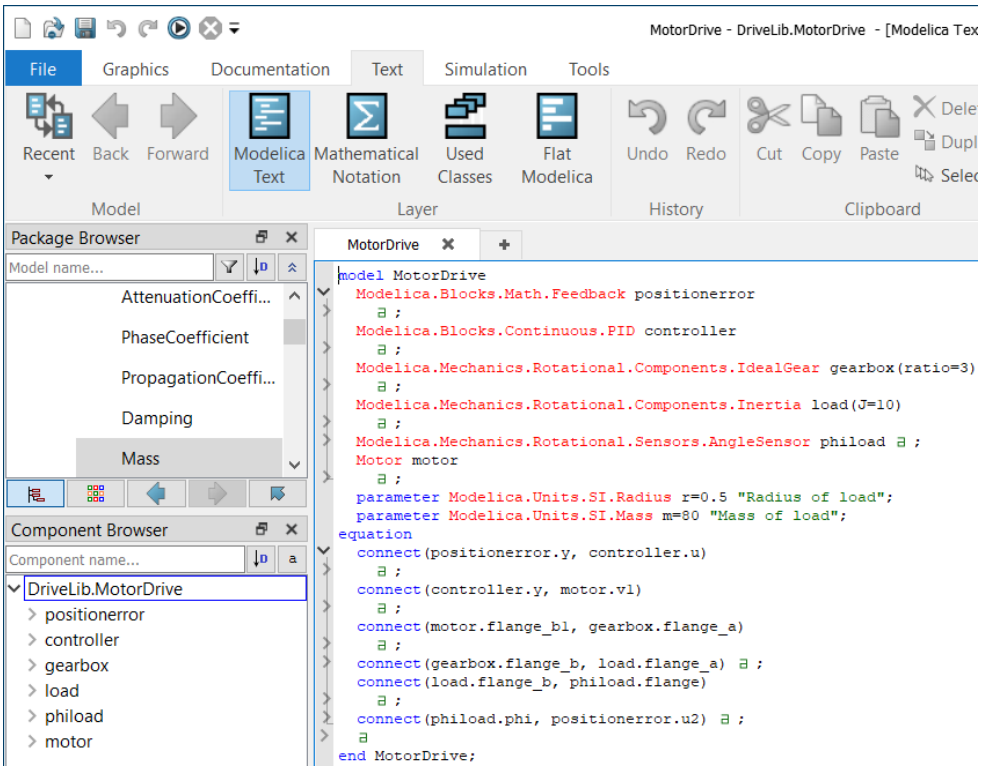
In Modelica Text representation above, the components and connections are indicated by an expand icon ▷ in the margin to the left of the text, and icons in the text. It is possible have them expanded textually in two ways.

One way is to work with individual annotation. It is possible to click on the ▷ or the corresponding icon to expand that annotation but no other. If “sub-annotations” are present, new ▷ and icons will be visible, which in turn can be expanded. (Of course they can also be collapsed using the collapse icon ◢.)

The other way is to work with all annotations at the same time. If that is wanted, right-click and select **Expand > Show components and Connections**. It is also possible to expand the annotations such as the graphics for the icon of the model, the positions and sizes of the icons of the components, the path of the connections etc. This is done by right-clicking and selecting **Expand > Show entire text**. However, we refrain from showing it in this document.

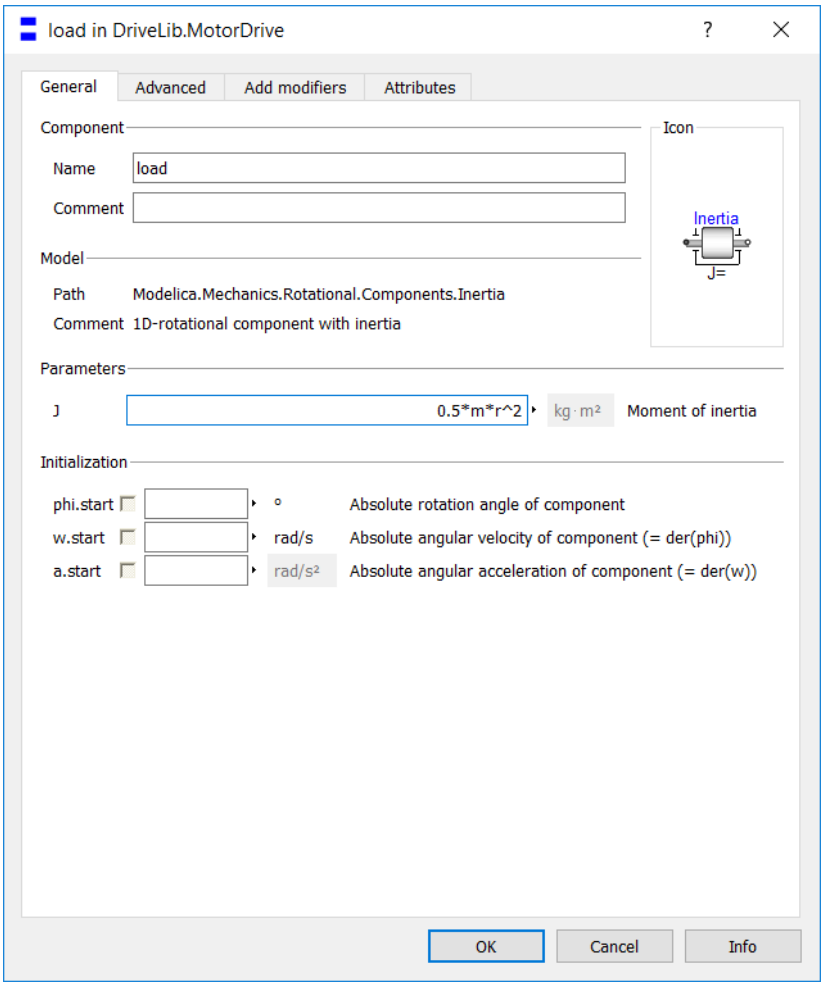
Below the Modelica text as a result of first selecting select **Expand > Show components and Connections** and then the annotation for Motor has been individually expanded by clicking on a corresponding ▷ in the margin (which means that now a ◢ is shown, indicating the possibility to collapse that annotation individually).

Expanded text representation.



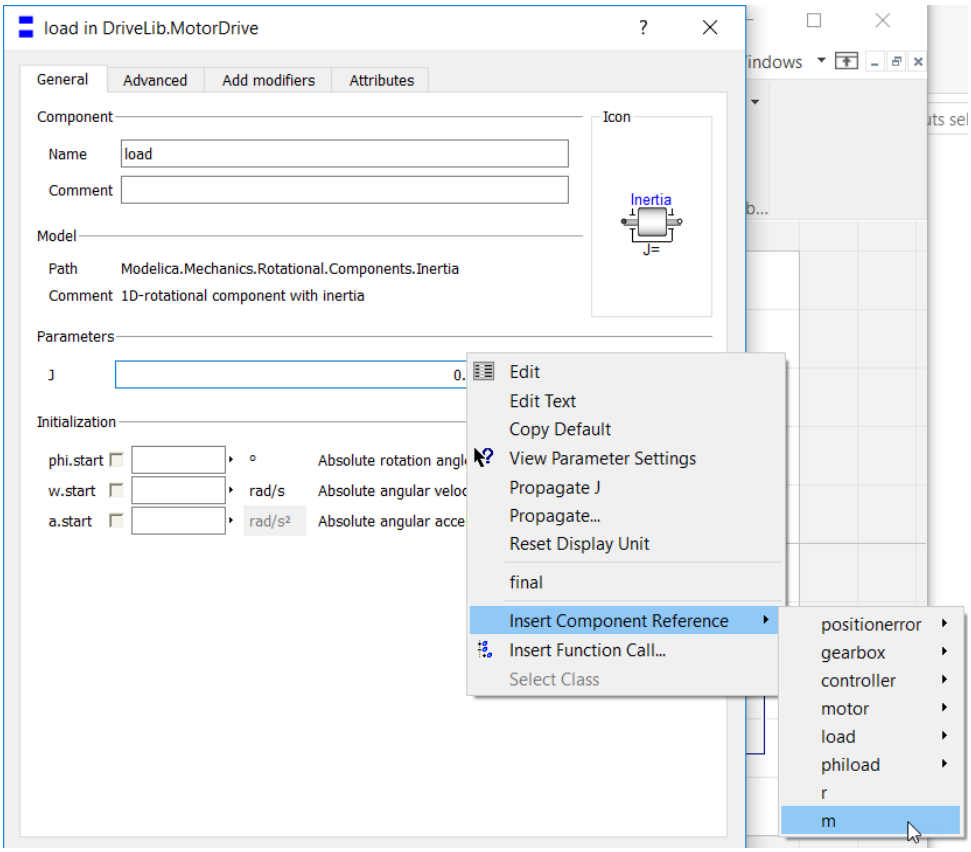
Ok, now click the **Graphics** tab to see the diagram representation. Double-click on the load icon to open the parameter dialog.

Binding a parameter to an expression.



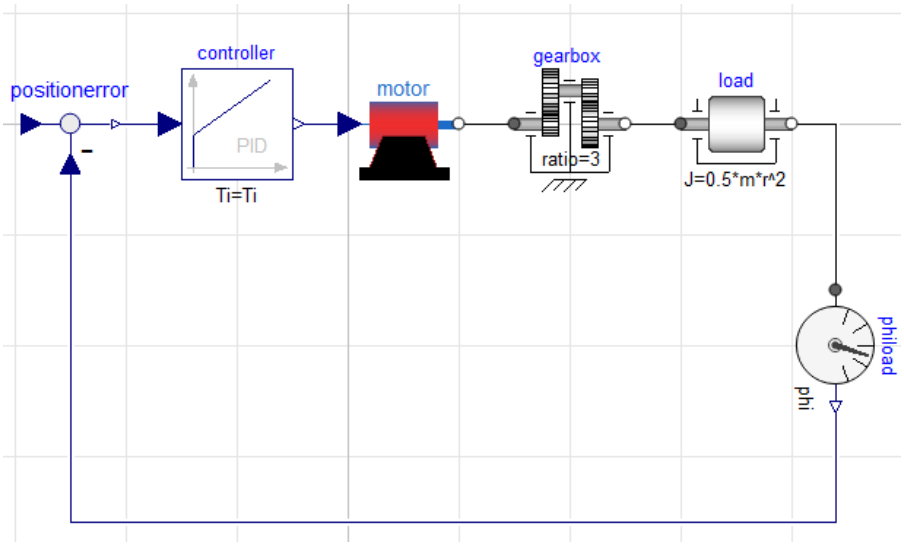
Click in the value field of J and enter the expression for J (like above). Click **OK**. The model window now displays the expression for the load inertia. When entering the expression, you are sometimes not sure about the exact name of the variables names, for example is the radius called r, r0 or r1? The problem is even more pronounced when you would like to reference a variable a bit down in the component hierarchy. Dymola can assist you. First you enter 0.5* and then you click on the small triangle to the right of the value field. Select **Insert Component Reference** and then **m**.

Using Component Reference to enter a formula.



You have now 0.5*m in the value field for J. Enter *. Use the menus to get a reference to r. Complete the expression with the square. Click **OK**. The model window now displays the expression for the load inertia.

The component's parameter definition is visible in the model.

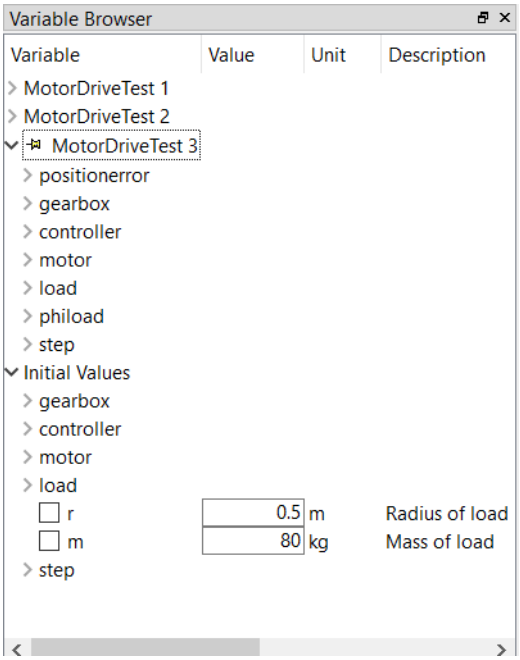


Open MotorDriveTest and switch to the **Simulation** tab.



Translate.

A bound parameter cannot be changed interactively.



The parameters r and m can be set interactively between simulation runs, but not the parameter load.J because it is no longer a free parameter; an expression is now binding it to r and m.

2.4.9 Documenting the simulation

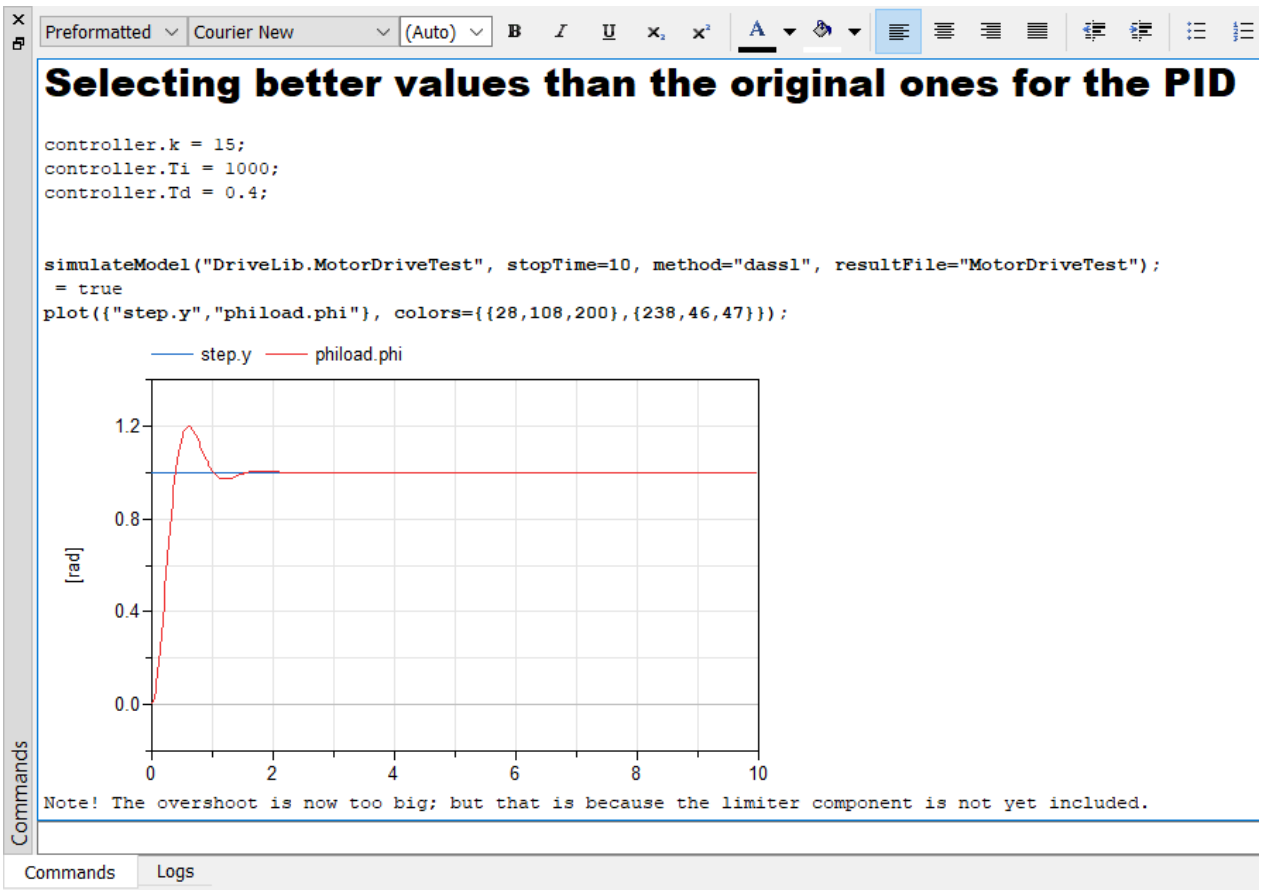
The command window can be used to document the simulation. Text (also headers), images, links, and mathematical expressions can be inserted and edited using the tools that are available in the toolbar in the top of the command window.

To use the command window this way, you must right-click in it and select **Show Toolbar**.

Plot results can also be inserted in the command log (together with the corresponding plot command) by using the command **Generate > Image in Log**. The command is available in the **Simulation > Plot: Options** tab. You can also insert the current plot by clicking the command input line in the bottom of the command window.

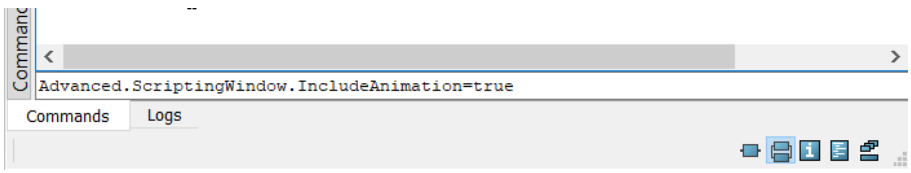
Now the content in the command log pane can be edited to document the simulation in a good way. Commands given and output results will automatically be included.

An example of documenting better values of the PID controller could be:



Some comments:

- This example is just a minor example of what can be done. More about using the command log pane to document simulations can be read in the chapter “Simulating a model”. Note that math rendering of equations etc. is supported, including indexing and Greek letters.
- The command log pane is enlarged; the command window can also be undocked instead using the **Undock** button.
- The header and the last line are entered by the user.
- The simulation command is output from the system: including “true” in the following line, indicating success (assuming the user has given a simulate command).
- The plot command and plot image are added the following way:
    - The variables are plotted in a plot window.
    - To get the plot command in the plot window (really not needed for documentation, but for scripting), the user clicked the command input line of the command window – that echoed the underlying plot command in the command log.
    - To add the image in the command log, the user used the command **Generate > Image in Log**. This command is available in the **Simulation > Plot: Options** tab.
    - The user removed the rather long createPlot command that was added together with the image.
- The content of an animation window can also be included using a flag; the below shows how to set this flag to true by typing in the command input line of the command window. This is one way of setting a flag; using flags is not unusual in Dymola.



- Sections of the command log pane can of course be copied to e.g. Microsoft Word for further work.
- The content of the command log pane can be saved as a command log in Dymola, in various formats (and including various contents).

2.4.10 Scripting

Scripting makes it possible to store actions, for various reasons. Examples might be:

- The script can be used to run a demo.
- The script can be used for testing of e.g. different parameter sets.
- The script can be used as an action that can be included in other models.

The first item can be solved creating a script file; while the last one is best solved creating a function that can be included in other models.

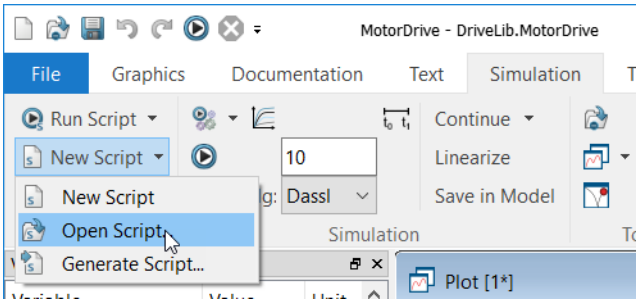
Creating functions is not treated here, please see, in the chapter “Simulating a model” for more about scripting, in particular using functions.

If the simple example here should be saved as a script file, the easiest way is to do the following (assuming that the models have been saved):

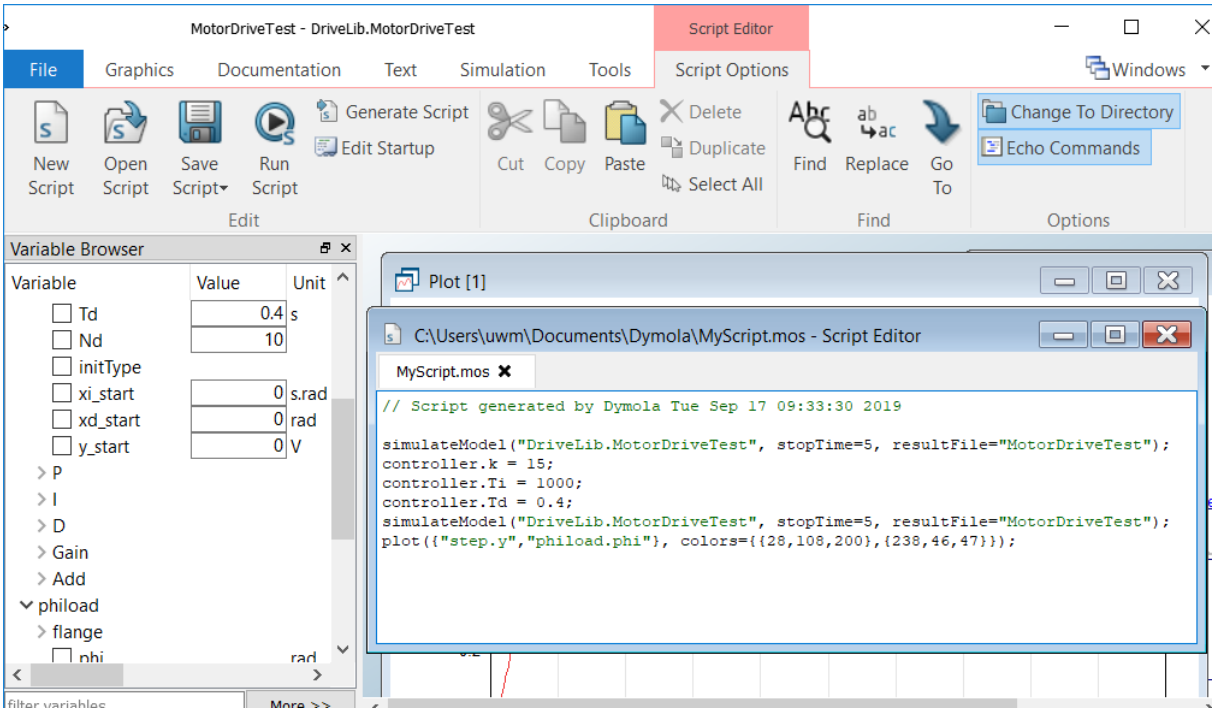
1. Clear everything using the command **File > Clear All**.
2. Clear the content of the command log pane using the command **File > Clear Log**.
3. Reopen the DriveLib package using **File > Recent Files**.
4. Open MotorDriveTest, and simulate it.
5. Set wanted values of controller. Simulate again.
6. Plot `phiload.phi` and `step.y`. (If those are already selected, you may have to deselect them and select them again to get the below item to work.)
7. Force a plot command in the log by clicking the command input line.
8. Save the command log using **File > Save Log…**. Select saving it as a Modelica Scripts (*.mos) file by selecting that alternative in the **Save as type** input field. Save the script where the model resides. Give it a name, e.g. MyScript.

The saved script can be opened (and edited) using the Dymola script editor. (A general text editor, e.g. Microsoft Notepad can also be used, but the script editor contains more facilities.)

To open the script in the Dymola Script editor, in the **Simulation** tab, in the **Scripting** group, click the arrow next to **Run Script** and select **Open Script**, then browse for the script file.



When the script file is opened in a Dymola script editor window, a new tab, the **Script Options** tab, appears for scripting (like the **Plot Options** tab for the plot window):



Note the difference between the saved log and the content in the command log pane. By saving as a .mos file *only* executable commands are saved.

Note also that the plot command echoed in the command log does not include features like plot header etc.; but for simple plot handling this is sufficient – more advanced commands can be used to fully restore a plot window using scripting.



To run the script, you have to be in the **Simulation** tab (or in the **Script Editor** tab). The command button **Run Script** works differently in the Simulation tab and in the **Script Editor** tab – in the Simulation tab you have to browse for the script to run, in the **Script Editor** tab you directly run current script.

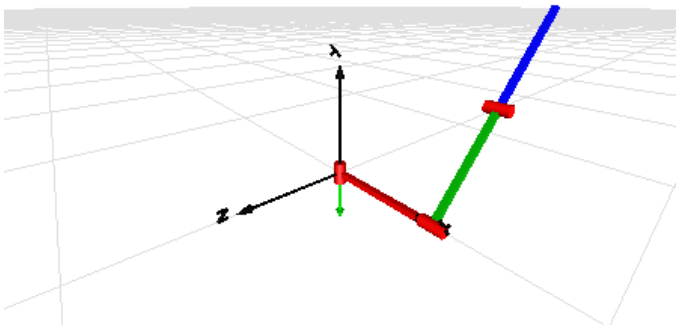
Some comments:

- This script file is very simple, just showing the idea of a script file rather than being a good example.
- Realizing how saving a script file works, it is not necessary to start all over to create the file, the total simulation can be saved, and afterwards the script file can be edited to keep only the wanted parts of the simulation. However, it is important to test it if created that way.
- More information about script files is available in the chapter “Simulating a model”.
- Working with scripting using functions is even smarter, for more information please see the reference in above item.

2.5 Building a mechanical model

We will now develop a more complex model, a 3D mechanical model of a pendulum called a Furuta pendulum. It has an arm rotating in a horizontal plane with a single or double pendulum attached to it, see below.

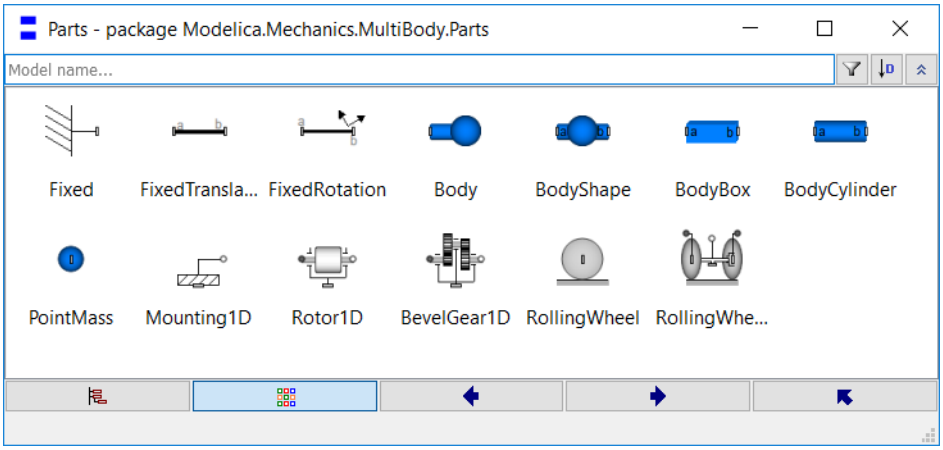
The Furuta pendulum.



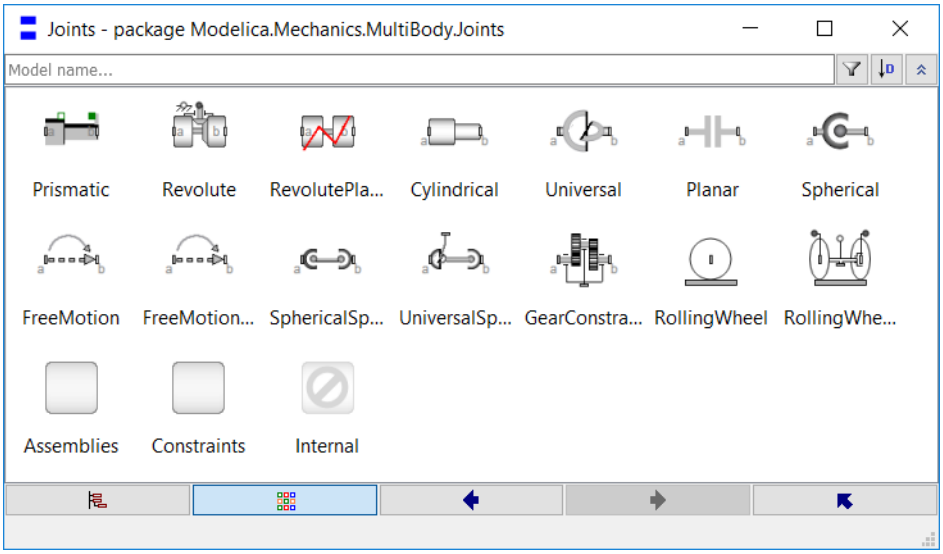
Start Dymola. Open Modelica Standard Library. In the package Mechanics open the sub-package MultiBody. This package includes 3D mechanical components such as joints and bodies, which can be used to build a model of the Furuta pendulum.

To build the Furuta pendulum, you will need to use the Parts and Joints sub-packages. If you open them in library windows by right-clicking on them in the package browser and using the command **Open Library Window** (and adapting the window) they will look the following:

The Parts sub-package library window.



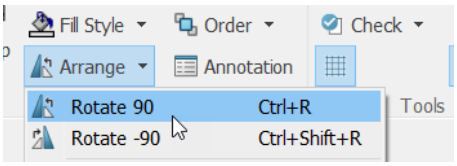
The Joints sub-package library window.



Create a model by using the command **File > New > Model** and give the name **Furuta**.

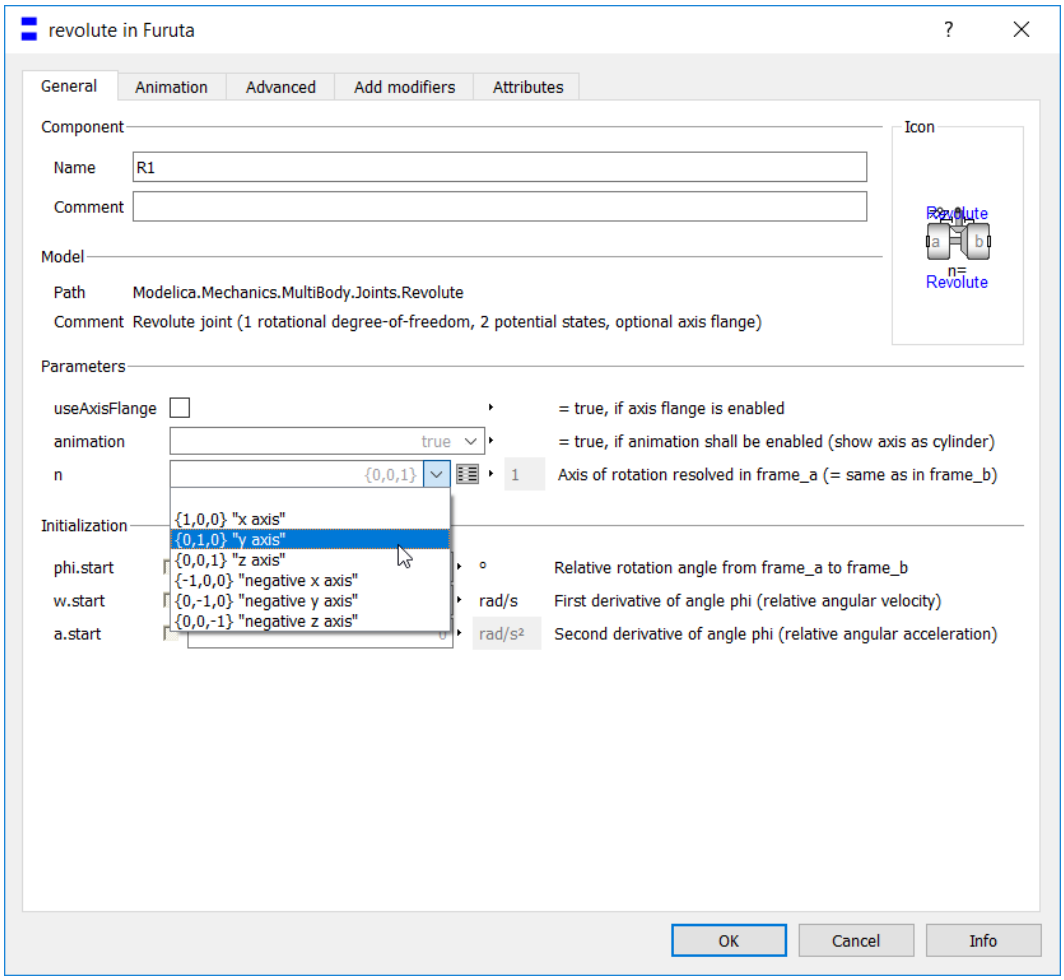
The first step in building a MBS (MultiBody System) model is to define an inertial system. Drag the component World from the package browser (Multibody package) into the Furuta diagram layer. The default parameters need not be changed. (The gravity of acceleration is set to 9.81 and pointing in the negative direction of the y axis).

We then need a revolute joint. Drag the model Joints.Revolute onto the Furuta window. You can either drag from the package browser or the library window. Select **Arrange > Rotate 90**.



This just changes the appearance of the icon. Double-click on the icon to open the parameter dialog, or or right-click it and select **Parameters**.

Change the name to R1. The axis of rotation is set as the parameter n. We would like to have a vertical axis of rotation; use the list of choices and select “y axis”.



Click **OK**. Connect the connector of world to the bottom connector of the revolute joint. Here it is very convenient to use smart connect, that is, drag the revolute joint so the bottom connector is on top of the connector on the World component. You will see a short green highlight, meaning that the components are now connected; then you can just drag them apart again, the connection will follow.

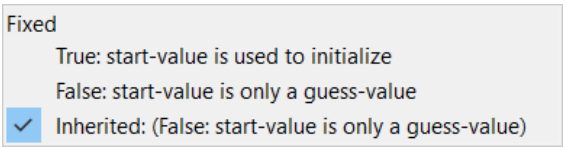
A bar is then connected to the revolute joint. There is one bar which has the visual appearance of a box during animation, called BodyBox in the Parts library. Drag such a component to the edit window. Double-click on the component to display the parameter dialog, or right-click it and select **Parameters**. Name it B1. We would like the bar to be 0.5 long and initially horizontal and pointing in the positive x direction. This is specified by setting the vector r between the two ends of the body to {0.5, 0, 0}. Click on the **Edit** icon just to the right of the value field of r and a vector editor pops up. Enter 0.5 in the first field, 0 in the following two fields (you must specify the values since no default values are shown, otherwise you later will get an error message). Click **OK**. The width and height will be 0.1 by default.

To get nicer animation, you can set different colors for the bars. For example, use the list of choices of color to make the bar red. Click **OK** and connect the bar to the revolute joint.

From the bar B1, we connect another revolute joint, R2, having a horizontal axis of rotation, n={1, 0, 0} and a BodyBox, B2, (rotated -90), with r={0, -0.5, 0} and a different color than the first bar.

When simulating, the start values of R2 are interesting. Looking at the parameter dialog for R2 the initial value of the angle (phi.start), the initial velocity (w.start) and the relative angular acceleration (a.start) can be specified in the dialog. The idea is of course to specify values when simulating, but we have to specify what type of start values we want to use. This is done by clicking on the box after each start value. The choices are:

Choices for start values.



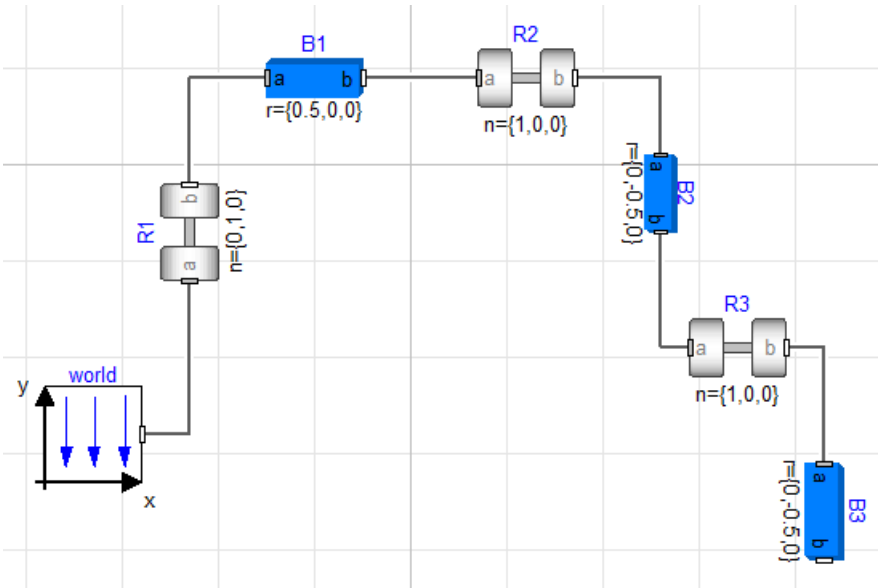
Actually we don't need to change anything, Dymola will itself take care of this, but we will have warnings when translating. In order to avoid these warnings, phi.start and w.start should be set to fixed=true using the menu above for all joints.

To get a double pendulum, create another similar joint and another BodyBox and connect them. This is accomplished easily by selecting the two components already present and choosing **Edit > Duplicate**. Notes:

- You may get an "extra" automatic connection when duplicating since you are close to existing connection lines (part of the "smart connect" feature, actually) that you will have to remove.
- The selection of the start value type for phi.start can be removed from R3 if wanted.
- Don´t forget to change the color of the third bar.

When you have connected the new components, you should now have arrived at a model that is similar to the following:

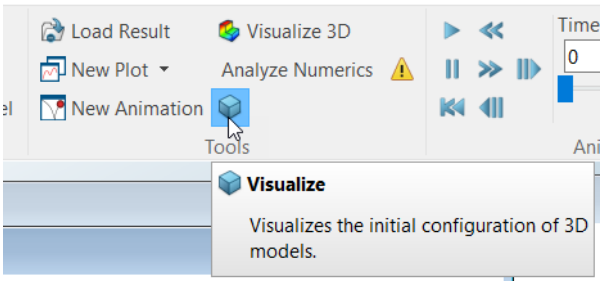
The diagram of the Furuta pendulum.



Now it is time to simulate. Start with clicking the **Simulation** tab and select **Translate**.

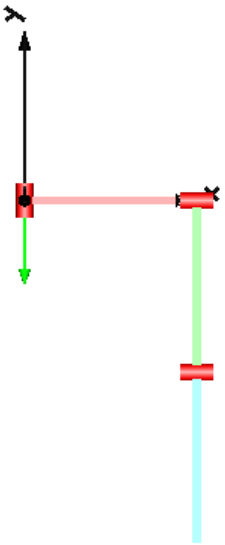
Since you have built a 3D mechanical model using components from the Modelica Standard Library, an **Animation** window is automatically opened, and you also have a new tab, **Animation Options**, that you can click on to have access to commands for animation.

When building 3D mechanical models, it is possible to get visual feedback to check that the mechanical system is put together as intended. Select the command **Visualize**.

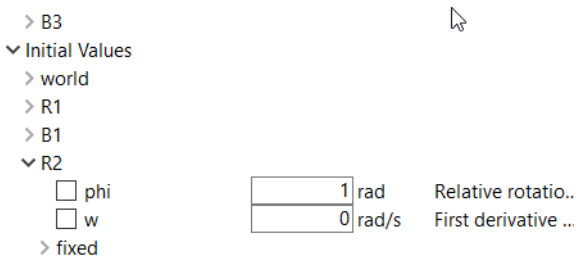


The animation window that appears and shows the initial configuration of the system:

Initial configuration of system.



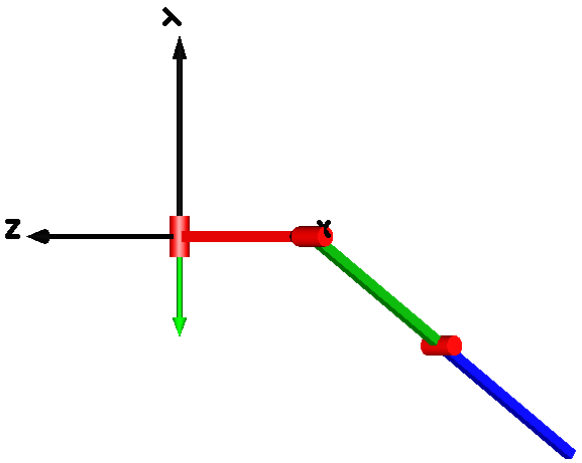
Translate the model. In the variable browser, open Initial Values > R2 by clicking on the expand icon ▷ in front and enter a value for phi_start, say 1 rad:



Set the stop time to 5 (you can set it directly in the Simulation tab) and select **Simulate** to simulate the model.

View the pendulum in the animation window; you may want to adjust the perspective when working with the robot (please see section “Simulation” on page 46 for tools used). It will be nicer to present it by e.g. moving it to the following position:

Initial configuration of system rotated around y axis.



(The representation of the revolute joints as cylinders can be changed using the parameter animation; if that is set to false the joint cylinders are not shown.)

Change parameters and study the different behavior.

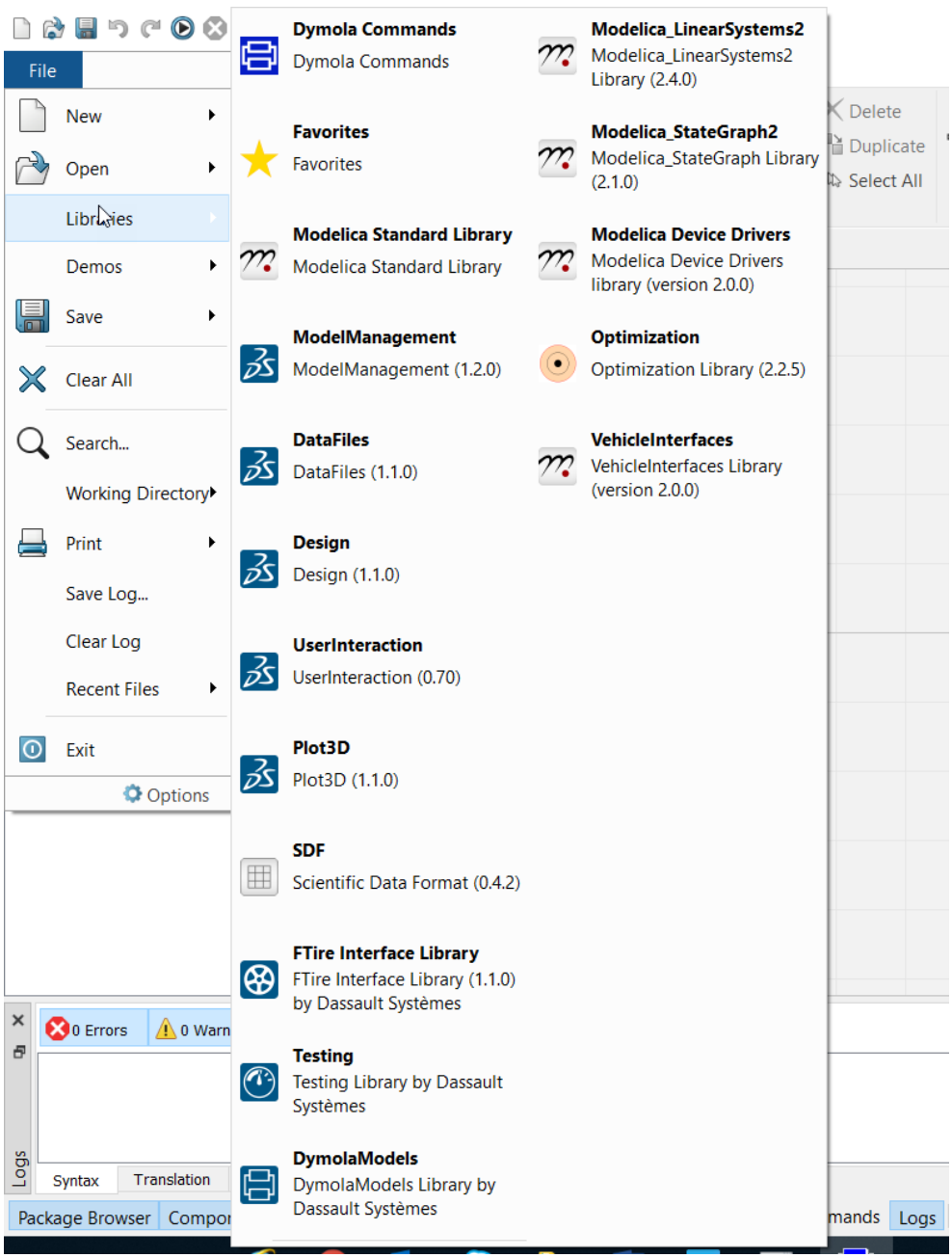
Try to control the pendulum in such a way as to get a stable upright position. (A trick is to use a “mechanical PD controller”, i.e. a spring and damper attached to the tip of the second bar and to a high position.)

2.6 Other libraries

2.6.1 Libraries available in the File menu by default

Using the command **File > Libraries** in a Dymola program with no extra libraries installed will at present display the following:

Libraries available when no optional library is installed.



DymolaCommands contains selected commands (built-in functions). They are grouped in the same way as they are documented in the chapter "Simulating a model" (section

"Scripting", sub-section "Built-in functions in Dymola"). Note that this library is automatically opened when opening Dymola.

Modelica Reference is the documentation of classes, operators etc. from the Modelica Language Specification. There are no components that can be dragged from this library to the edit window, it is purely documentation. This library is free.

Favorites is a library containing your favorite models, to simplify reuse of those models. It is empty from the beginning; you add yourself models in it.

Modelica Standard Library has been dealt with earlier in this chapter. This library is free.

Model Management deals with version management, model dependencies, encryption, model and library checking (including regression testing of libraries), model comparison and model structure. This library is free. For more information, please see the chapter "Model Management".

DataFiles contains functions for reading and writing data. For more information, please see the chapter "Developing a model", section "Advanced model editing", sub-section "Using data from files".

Design deals with four main areas:

- **Model calibration** makes it possible to calibrate and assess models. This package is free. For more information, please see the chapter "Model Calibration".
- **Model experimentation** gives the user possibility to vary parameters of the system to get an intuitive knowledge of the behavior of the model. Monte Carlo analysis is included. This part of the library is free. For more information, please see the chapter "Model Experimentation".
- **Design optimization** is used to determine improved values of model parameters by multi-criteria optimization based on simulation runs. The Optimization option is required if used on models that are more complex. For more information, please see the chapter "Design optimization". Note that a newer library is available also, with higher functionality; please see "Optimization" below.
- **Model validation** supports validation and regression testing of a specified model. The idea is to compare the simulation result against reference data to check if for example changes in model libraries change the result. The reference data is assumed to be stored as trajectory files, which is the data format used by Dymola to store simulation results. When developing a model, it is very natural and recommended to provide it with carefully checked reference simulation results. Please also compare the library "Model Management" where model validation and regression testing is supported on a larger scale.

UserInteraction enables changing parameters and inputs by numbers, sliders etc., as well as animation of simulation results. Please see the documentation in the package for more information.

Plot 3D is used to visualize models in 3D. This library is free.

SDF is a package for support of the Scientific Data Format (.sdf) files. The package is free.

FTire Interface Library enables interfacing Dymola with the FTire (Flexible Structure Tire Model) software from cosin scientific software.

Testing With this library, it is possible to build test cases, create reference results and run the tests – all within Dymola. It allows detecting unwanted side effects on model changes in an early stage by running regular tests. The library contains multiple blocks for the continuous comparison of signals to reference values and trajectories. Existing examples and test models can easily be converted to a test case by providing a reference and connecting the signal of interest to one of the various comparison blocks.

DymolaModels is the foundation library for the commercial libraries from Dassault Systèmes. It focuses on entry-level simulation. Therefore, it contains models with few parameters to enable quick first results. Many of the models offered here are available in much higher detail in the commercial libraries of Dassault Systèmes. It can also be seen as an extension to the Modelica Standard Library. It offers models, which are not available as such in MSL. The library is available with the standard license of Dymola, so no additional license is required for its usage.

Modelica_LinearSystems2 is a free library from Modelica Association providing different representations of linear, time invariant differential and difference equation systems, as well as typical operations on these system descriptions. See the documentation inside the package for details.

Modelica_StateGraph2 is a free library from Modelica Association providing components to model discrete event, reactive and hybrid systems in a convenient way with deterministic hierarchical state diagrams. It can be used in combination with any Modelica model. It has larger power than the package StateGraph in the Modelica Standard Library. See the documentation inside the library for details.

Modelica_DeviceDrivers is a free library that allows accessing some external devices in Modelica models. Such external devices are input devices, communication devices, shared memory, analog-digital converters and else. This is achieved by using the Modelica external C interface to call the appropriate C driver functions provided by the underlying operating system. Currently Microsoft Windows and Linux are supported.

Optimization is a newer version of the Design optimization package in the Design library (see above). It contains much more functionality and should be used for new applications rather than the old one. For more information, see the manual for the library, which is available using the command **Help > Documentation** or the shortcut **F1**.

VehicleInterfaces is a free library providing standard interface definitions for automotive subsystems and vehicle models to promote compatibility between the various automotive libraries. See the documentation inside the library for details.

2.6.2 Libraries that can be added

There are a number of libraries available, both free and commercial ones. For an overview of them, please see http://www.modelica.org/libraries.

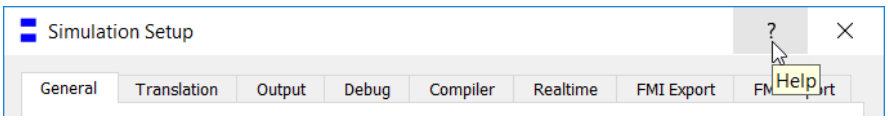
2.7 Help and information

2.7.1 Reaching compressed information

The Help command/button etc

The **Help** command is used to display information about many things. The command is activated in three different ways depending on where is should be used.

In any dialog



In any dialog the **Help** is reachable using the **?** in the upper corner of the window. Click on the **?** and then on the field of interest. Please note that sometimes the information concerns (only) exactly what has been clicked on, sometimes the information concerns a group of signals etc.

In any menu

When displaying any menu, help for a certain entry is available by resting the cursor on it and then pressing **Shift+F1**.

Tooltips

By resting the cursor on any component or button a tooltip text is shown. For buttons it will be the name of the button, and a description, for components it will be the name of the component + the path to it.

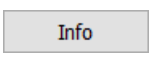
The Search command



You can search using the command **File > Search**. See the index entry "*file menu: search*" in the index in the end of the manual, for more information.

2.7.2 Reaching more extensive information

The Info command/button



If any component is selected in the diagram or in the package or component browser, right-clicking will take up a context menu where the command **Info** is available. Inside a dialog a button **Info** is available instead.

The Search command

The search functionality can be reached using the command **File > Search**. For reference, please see the corresponding section above.

The documentation layer

This layer can be used to display more extensive information about packages and components. Please see the chapter “Developing a model”, section “Basic Model editing”, sub-section “Documentation” for more information. This type of documentation can also be exported to HTML files etc.

Manuals for Dymola and a number of libraries



On the **Tools** tab, in the **Help** group, the **Help Documents** command can be used to display manuals and to go to the Dymola website.

The Dymola website

To access the Dymola website, use www.Dymola.com. Here you can download trial versions of Dymola, as well as see the latest information about supported compilers etc.

2.8 Shortcuts

Shortcuts are available for a number of commands and actions. The following is an overview of available shortcuts. Please note that the very common clipboard and history shortcuts, e.g. **Ctrl+C** for Copy, **Ctrl+V** for Paste, and **Ctrl Z** for Undo are listed in the "General clipboard and history commands" section below.

General (Windows command)

Command	Shortcut
Close [Closing of Dymola]	Alt+F4

File menu

Command	Shortcut
New > Model	Ctrl+Shift+M
New > Package	Ctrl+Shift+P
New > Duplicate Class	Ctrl+Shift+D
New > Extend From	Ctrl+Shift+E
Open	Ctrl+O
Load	Ctrl+Shift+O
Save	Ctrl+S
Save All	Ctrl+Shift+S
File > Search…	Ctrl+Shift+F
Print	Ctrl+P

Dymola main window: Model layers (Class layers)

The last shortcut in the list is not really a shortcut to a model layer, but here for completeness.

Note that you also go to a specific tab when going to a model/class layer. Please compare the shortcuts to tabs in the next section.

Command/Action	Shortcut
Display the **i**con layer	Ctrl+F1
Display the diagram layer	Ctrl+F2
Display the documentation layer	Ctrl+F3
Display Modelica text layer	Ctrl+F4
Display the Used Classes layer	Ctrl+F5

Display the "simulation layer" (the **Simulation** tab)	Ctrl+F6

Dymola main window: Tabs (Ribbon Tabs)

Note that you only can go to tabs currently visible/selectable, that is, you can, for example, never be in the **Graphics** tab and display the **Plot** tab using **Alt+P**; the **Plot** tab is not visible when working in the **Graphics** tab.

Please compare these shortcuts with the shortcuts to model layers, see the section above.

Command/Action	Shortcut
Display the **File** tab (the **File** tab is not a ribbon tab, but here for completeness)	Alt+F
Display the **Graphics** tab	Alt+G
Display the **Documentation** tab	Alt+D
Display the **Text** tab	Alt+T
Display the **Simulation** tab	Alt+S
Display the **Tools** tab	Alt+O
Display the **Plot** tab	Alt+P
Display the **Animation** tab	Alt+A
Display the **Script Editor** tab	Alt+C

General history and clipboard commands

Command	Shortcut
Undo	Ctrl+Z
Redo	Ctrl+Y
Cut	Ctrl+X
Copy	Ctrl+C
Paste	Ctrl+V
Delete	Del
Duplicate	Ctrl+D
Select All	Ctrl+A

Graphics tab commands

Command	Shortcut
Fit Window	Ctrl+T
Arrange > Rotate 90	Ctrl+R
Arrange > Rotate -90	Ctrl+Shift+R
Arrange > Flip Horizontal	H
Arrange > Flip Vertical	V
Arrange > Manhattanize	Ctrl+M
Align > To Grid	Ctrl+G
Order > Bring to Front	Ctrl+PageDown
Order > Send to Back	Ctrl+PageUp
Order > Bring Forward	Ctrl+Down
Order > Send Backward	Ctrl+Up
Toggle Grid	G
Check > Normal	F8
Check > Stop	F12
Connect Mode	C
Transition Mode	T
Find	Ctrl+F
Insert (> New Variable)	Ctrl+N

Documentation tab commands+ Command window (log pane) commands (command buttons)

Command	Shortcut
Find	Ctrl+F
Replace	Ctrl+H
Go To	Ctrl+G
Bold	Ctrl+B
Italic	Ctrl+I
Underline	Ctrl+U

Subscript[5]	Ctrl+=
Superscript	Ctrl++

Text tab commands

Command	Shortcut
Check > Normal	F8
Check > Stop	F12
Find	Ctrl+F
Replace	Ctrl+H
Go To	Ctrl+G
Insert (> New Variable)	Ctrl+N

Simulation tab commands

Command	Shortcut
Translate > Normal	F9
Translate > FMU	Ctrl+F9
Simulate	F10
Stop	F12
Continue > Continue	F11
Play (animation)	F3
Pause (animation)	F4
Rewind (animation)	F7
Reverse (animation)	Shift+F6
Forward (animation)	Shift+F5
Backstep (animation)	F6
Step Forward (animation)	F5

[5] Does not work on a Swedish keyboard.

Animation tab commands

Command	Shortcut
Play (animation)	F3
Pause (animation)	F4
Rewind (animation)	F7
Reverse (animation)	Shift+F6
Forward (animation)	Shift+F5
Backstep (animation)	F6
Step Forward (animation)	F5

Script Editor tab commands

Command	Shortcut
Find	Ctrl+F
Replace	Ctrl+H
Go To	Ctrl+G

Help commands

Command	Shortcut
What´s This?	Shift+F1
Documentation	F1

Model tabs commands (when in any tab except the Simulation tab)

Action	Shortcut
Activate next model tab in model tab list (go down in list)	Ctrl+Tab
Activate previous model tab in model tab list (go up in list)	Ctrl+Shift+Tab

Active subwindow selection (when in the Simulation tab)

Action	Shortcut
Activate next subwindow in recent window list (go down in list)	Ctrl+Tab
Activate previous subwindow in recent window list (go up in list)	Ctrl+Shift+Tab

Windows (handlings) commands

Action	Shortcut
Cascade Windows	Ctrl+Shift+C
Tile Windows	Ctrl+Shift+T

Package browser (context commands)

Command	Shortcut
New > Model	Ctrl+Shift+M
New > Package	Ctrl+Shift+P
New > Duplicate Class	Ctrl+Shift+D
Order > Move Up	Ctrl+Up
Order > Move Down	Ctrl+Down
Order > Move to Top	Ctrl+PageUp
Order > Move to Bottom	Ctrl+PageDown
Rename	F2
Copy Path	Ctrl+C

Modelica Text editor (context commands)

Command	Shortcut
Expand > Hide All Annotations	Ctrl+1
Expand > Show Components and Connections	Ctrl+2
Expand > Show Entire Text	Ctrl+3
Expand > Expand Local Packages	Ctrl+4
Highlight Syntax	Ctrl+L
Reformat Selection	Ctrl+Shift+L
Comment/Uncomment Selection	Ctrl+K
Insert Function Call	Ctrl+W
Edit Function Call	Ctrl+U
Variables > New Variable	Ctrl+N

[Code completion][6]	Ctrl+Space
[Code completion – SI units][7]	Ctrl+Shift+U

Plot window diagram (context menu of plot diagram, nothing selected, more than one diagram in the plot window)

Command	Shortcut
Move Up	Ctrl+Up
Move Down	Ctrl+Down

Command window (command log pane)

Command	Shortcut
Find / Find and Replace (depending if toolbar visible)	Ctrl+F

Command window (command input line)

Command	Shortcut
Command History	Ctrl+H
Insert Function Call	Ctrl+W

Default shortcuts for text editor in Qt

Action	Shortcut
Select all text	Ctrl+A
Delete the character to the left of the cursor	Backspace
Delete the character to the right of the cursor	Delete
Copy the selected text to the clipboard	Ctrl+C
Copy the selected text to the clipboard	Ctrl+Insert
Delete to the end of the line	Ctrl+K
Paste the clipboard text into the text edit	Ctrl+V
Paste the clipboard text into the text edit	Shift+Insert

[6] There is no such command, item in list to make the list of control key combinations complete.
[7] There is no such command, item in list to make the list of control key combinations complete.

Delete the selected text and copy it to the clipboard	Ctrl+X
Delete the selected text and copy it to the clipboard	Shift+Delete
Undo the last operation	Ctrl+Z
Redo the last operation	Ctrl+Y
Move the cursor one character to the left	Left
Move the cursor one word to the left	Ctrl+Left
Move the cursor one character to the right	Right
Move the cursor one word to the right	Ctrl+Right
Move the cursor one line up	Up
Move the cursor one line down	Down
Move the cursor one (viewpoint) page up	PageUp
Move the cursor one (viewpoint) page down	PageDown
Move the cursor to the beginning of the line	Home
Move the cursor to the beginning of the text	Ctrl+Home
Move the cursor to the end of the line	End
Move the cursor to the end of the text	Ctrl+End
Scroll the page horizontally	Alt+Wheel
Zoom the text	Ctrl+Wheel

To select (mark) text, hold down the **Shift** key whilst pressing one of the movement keystrokes, for example, **Shift+Right** will select the character to the right, and **Shift+Ctrl+Right** will select the word to the right, etc.

3 INTRODUCTION TO MODELICA

3 Introduction to Modelica

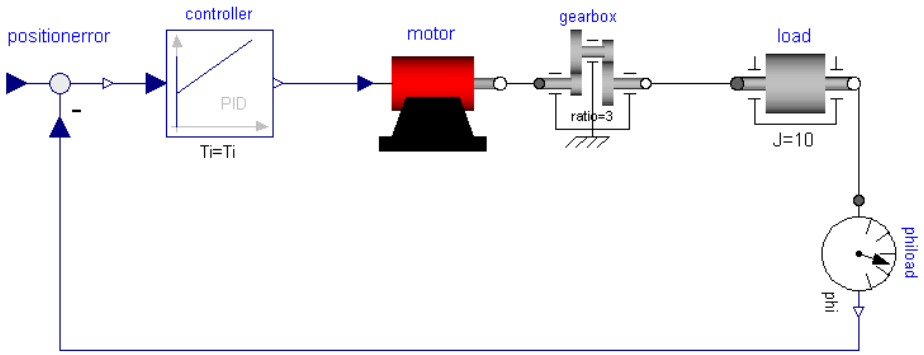
Model classes and their instantiation form the basis of hierarchical modeling. Connectors and connections correspond to physical connections of components. Inheritance supports easy adaptation of components. These concepts can be successfully employed to support hierarchical structuring, reuse and evolution of large and complex models independent from the application domain and specialized graphical formalisms.

The benefits of acausal modeling with DAE's will be demonstrated in this chapter and compared to traditional block diagram modeling. It will also be shown that tools can incorporate computer algebra methods to translate the high-level Modelica descriptions to efficient simulation code.

3.1 Modelica basics

Modelica supports both high level modeling by composition and detailed library component modeling by equations. Models of standard components are typically available in model libraries. Using a graphical model editor, a model can be defined by drawing a composition diagram by positioning icons that represent the models of the components, drawing connections and giving parameter values in dialogue boxes. Constructs for including graphical annotations in Modelica make icons and composition diagrams portable.

Schematic picture of a motor drive.



To describe how the details of a component are modeled, consider a simple motor drive system as defined above. The system can be broken up into a set of connected components: an electrical motor, a gearbox, a load and a control system. A Modelica model of the motor drive system is given below (excluding graphical annotations).

A Modelica model of the motor drive.

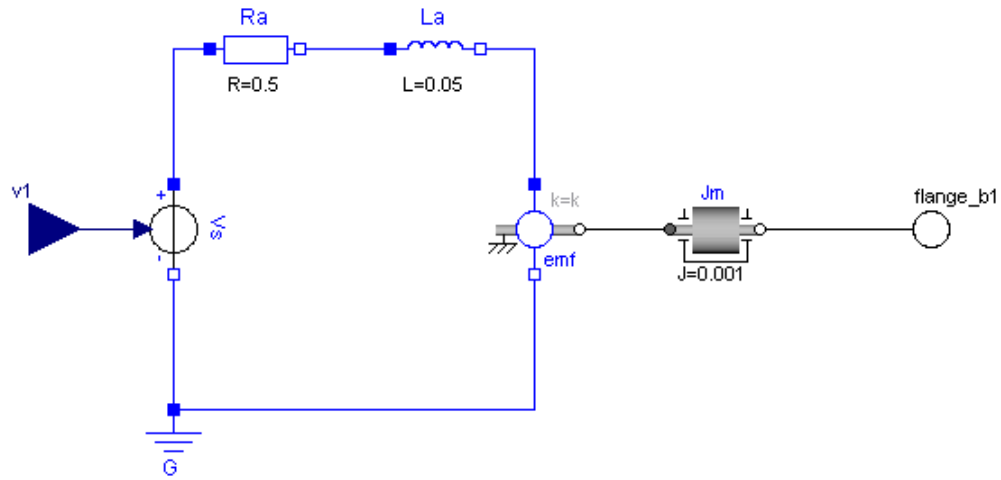
```
model MotorDrive
  Modelica.Blocks.Math.Feedback positionerror;
  Modelica.Blocks.Continuous.PID controller;
  Motor motor;
  Modelica.Mechanics.Rotational.Sensors.AngleSensor phiload;
  Modelica.Mechanics.Rotational.Components.Inertia load(J=10);
  Modelica.Mechanics.Rotational.Components.IdealGear
gearbox(ratio=3);
equation
  connect(phiload.phi, positionerror.u2);
  connect(positionerror.y, controller.u);
  connect(controller.y, motor.v1);
  connect(motor.flange_b1, gearbox.flange_a);
  connect(gearbox.flange_b, load.flange_a);
  connect(load.flange_b, phiload.flange);
end MotorDrive;
```

It is a composite model which specifies the topology of the system to be modeled in terms of components and connections between the components. The statement

```
Modelica.Mechanics.Rotational.Components.IdealGear gearbox(ratio=3);
```

declares a component `gearbox` of class `IdealGear` and sets the default value of the gear ratio, `n`, to 3.

A motor model.



A component model may be a composite model to support hierarchical modeling. The object diagram of the model class Motor is shown above. The meaning of connections will be discussed next as well as the description of behavior on the lowest level using real equations.

3.1.1 Variables

Physical modeling deals with the specification of relations between physical quantities. For the drive system, quantities such as angle and torque are of interest. Their types are declared in Modelica as

```
type Angle = Real(quantity = "Angle", unit = "rad",
                  displayUnit = "deg");

type Torque = Real(quantity = "Torque", unit = "N.m");
```

where `Real` is a predefined type, which has a set of attributes such as name of quantity, unit of measure, default display unit for input and output, minimum value, maximum value and initial value. The *Modelica Standard Library*, which is an intrinsic part of Modelica, includes these kinds of type definitions.

3.1.2 Connectors and connections

Connections specify interactions between components. A connector should contain all quantities needed to describe the interaction. Voltage and current are needed for electrical components. Angle and torque are needed for drive train elements.

```
connector Pin
  Voltage v;
  flow Current i;
end Pin;

connector Flange
  Angle r;
  flow Torque t;
end Flange;
```

A connection, `connect(Pin1, Pin2)`, with `Pin1` and `Pin2` of connector class `Pin`, connects the two pins such that they form one node. This implies two equations, namely Pin1.v = Pin2.v and Pin1.i + Pin2.i = 0. The first equation indicates that the voltages on both branches connected together are the same, and the second corresponds to Kirchhoff's current law saying that the current sums to zero at a node. Similar laws apply to flow rates in a piping network and to forces and torques in a mechanical system. The sum-to-zero equations are generated when the prefix **flow** is used in the connector declarations. The Modelica Standard Library includes also connector definitions.

3.1.3 Balanced models

A complete Modelica model that should be simulated must have the same number of unknowns and equations so that the unknowns can be solved from the equations. Assume you have built a model by dragging components from libraries and connected them. If you then would get an error message that one equation is missing and or that there is one equation too many, this is of course annoying. Moreover, it may be very difficult to pin-point what is wrong. It may be that a component is wrong of that you have connected them in a bad or unforeseen way.

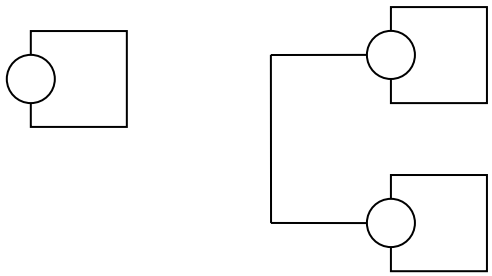
For a component having only connectors being defined as inputs and outputs it is possible to check that the component has the correct number of equations. We can here assume the inputs to be known and count that the component has the correct number of equations to calculate the output variables and the internal variables. It is also easy to check that such a component is used correctly. Each input must be connected to an output. For a connector specifying physical interaction between components we cannot specify causality to allow reuse as explained in the section further below. To solve this problem Modelica has the concept of balanced models.

Assume that we have a physical connector with n_f scalar flow connectors and n_p scalar non-causal, non-flow "potential" connectors. Modelica's concept of balanced models requires that the number of flow and non-causal non-flow variables should match: $n_f = n_p$.

It allows Dymola to check that library models have the correct number of equations and that when connecting such components the number of equations is correct. Experience has shown that this requirement is not restrictive for physical modeling.

A short motivation to the requirement will now be given. Consider the simplest case of a model where the only public part is one connector, and it is "physical", i.e. with n_f scalar flow connectors and n_p scalar non-causal, non-flow connectors. Connecting a physical connector to the outside world will give a certain number of equations. The two simplest uses of this

model are that the connector is unconnected (left) or connected to a connector of a similar component (right).



Consider first unconnected (left) case. If the connector is unconnected on its outside, it means that all flow connectors should be zero according to the Modelica specification. It means that the model must provide equations for all the non-flow physical connectors. If the model has n_l scalar local unknown variables, the model must provide $n_l + n_p$ equations. Thus for the right case, the upper component has $n_{1l} + n_p + n_f$ unknowns and similarly for the lower component. It gives in total $n_{1l} + n_{2l} + 2n_p + 2n_f$ unknown variables. The upper model has $n_{1l} + n_p$ equations and the lower has $n_{2l} + n_p$ equations. The connection gives $n_p + n_f$ equations. The total number of equations is $n_{1l} + n_{2l} + 3n_p + n_f$. Taking the difference between total number of unknown variables and the total number equations give $n_f - n_p$. To have this become zero, requires $n_f = n_p$.

3.1.4 Partial models and inheritance

A very important feature in order to build reusable descriptions is to define and reuse *partial models*. A common property of many electrical components is that they have two pins. This means that it is useful to define an interface model class `TwoPin`, which has two pins, `p` and `n`, and a quantity, `v`, which defines the voltage drop across the component.

```
partial model TwoPin
  Pin p, n;
  Voltage v;

equation
  v = p.v - n.v; p.i + n.i = 0;
end TwoPin;
```

The equations define common relations between quantities of a simple electrical component. The keyword **partial** indicates that the model class is incomplete. To be useful, a constitutive equation must be added. To define a model for a resistor, start from `TwoPin` and add a parameter for the resistance and Ohm's law to define the behavior.

```
model Resistor "Ideal resistor"
  extends TwoPin;
  parameter Resistance R;
```

```
equation
  R*p.i = v;
end Resistor;
```

A string between the name of a class and its body is treated as a comment attribute and is intended to describe the class. Tools may display this documentation in special ways. The keyword **parameter** specifies that the quantity is constant during a simulation experiment, but can change values between experiments. For the mechanical parts, it is also useful to define a shell model with two flange connectors,

```
partial model TwoFlange
  Flange a, b;
end TwoFlange;
```

A model of a rotating inertia is given by

```
model Shaft
  extends TwoFlange;
  parameter Inertia J = 1;
  AngularVelocity w;

equation
  a.r = b.r;
  der(a.r) = w;
  J*der(w) = a.t + b.t;
end Shaft;
```

where **der**(w) means the time derivative of w.

3.2 Acausal modeling

In order to allow reuse of component models, the equations should be stated in a neutral form without consideration of computational order, i.e., acausal modeling.

3.2.1 Background

Most of the general-purpose simulation software on the market such as ACSL, Simulink and SystemBuild assume that a system can be decomposed into block diagram structures with causal interactions (Åström *et al.* (1998)). This means that the models are expressed as an interconnection of sub models on explicit state-space form

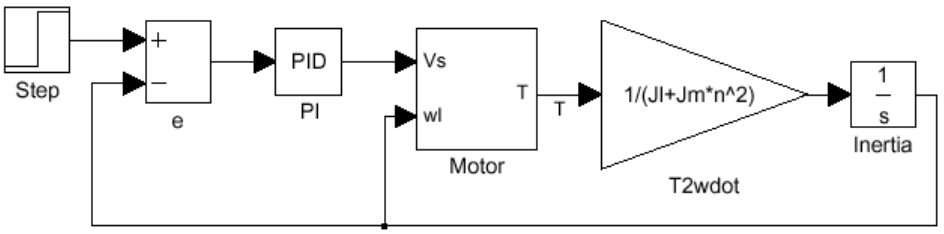
$$\frac{d\mathbf{x}}{dt} = \mathbf{f}(\mathbf{x}, \mathbf{u})$$

$$\mathbf{y} = \mathbf{g}(\mathbf{x}, \mathbf{u})$$

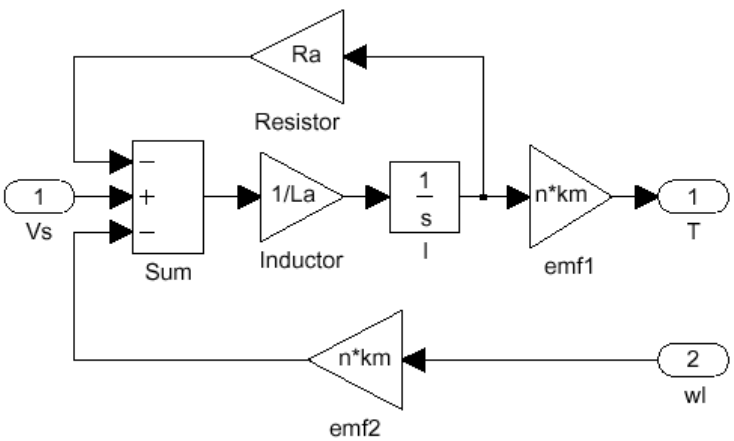
where **u** is input, **y** is output and **x** is the state. It is rare that a natural decomposition into subsystems leads to such a model. Often a significant effort in terms of analysis and analytical transformations is needed to obtain a problem in this form. It requires a lot of engineering skills and manpower and it is error-prone.

To illustrate the difficulties, a Simulink model for the simple motor drive (see page 150) is shown below. The structure of the block diagram does not reflect the topology of the physical system. It is easy to recognize the controller in the Simulink model for the motor drive, but the gearbox and the inertias of the motor and the load are no longer visible. They appear combined into a gain coefficient, $1/(J_l + J_m n^2)$

A Simulink model for the motor drive.



A Simulink model for the motor.



There is a fundamental limitation of block diagram modeling. The blocks have a unidirectional data flow from inputs to outputs. This is the reason why an object like a gearbox in the simple motor drive cannot be dealt with directly. It is also the reason why motor and load inertia appear in the mixed expression in the Simulink model. If it is attempted to simulate the basic equations directly there will be a loop which only contains algebraic equations. Several manual steps including differentiation are required to transform the equations to the form required by Simulink. The need for manual transformations imply that it is cumbersome to build physics based model libraries in the block diagram languages. A general solution to this problem requires a paradigm shift.

3.2.2 Differential-algebraic equations

In Modelica it is possible to write balance and other equations in their natural form as a system of differential-algebraic equations, DAE,

$$0 = \mathbf{f}(\frac{d\mathbf{x}}{dt}, \mathbf{x}, \mathbf{y}, \mathbf{u})$$

where **x** is the vector of unknowns that appear differentiated in the equation and **y** is the vector of unknowns that do not appear differentiated. Modelica has been carefully designed in such a way that computer algebra can be utilized to achieve as efficient simulation code as if the model would be converted to ordinary differential equations (ODE) form manually. For example, define a gearbox model as

```
model Gearbox "Ideal gearbox without inertia"
  extends TwoFlange;
  parameter Real n;

equation
  a.r = n*b.r;
  n*a.t = b.t;
end Gearbox;
```

without bothering about what are inputs from a computational point of view and use it as a component model, when modeling the drive system on page 150.

This use actually leads to a non-trivial simulation problem. The ideal gearbox is rigidly connected to a rotating inertia on each side. It means the model includes two rigidly connected inertias, since there is no flexibility in the ideal gearbox. The angular position as well as the velocity of the two inertias should be equal. All of these four differentiated variables cannot be state variables with their own independent initial values.

A DAE problem, which includes constraints between variables appearing differentiated, is sometimes called a "high index DAE". When converting it to ODE form, it is necessary to differentiate some equations and the set of state variables can be selected smaller than the set of differentiated variables. There is an efficient algorithm by Pantelides (1988) for the determination of what equations to differentiate and an algorithm for selection of state variables by Mattsson and Söderlind (1993).

In the drive example, the position constraint needs to be differentiated twice to calculate the reaction torque in the coupling, and it is sufficient to select the angle and velocity of either inertia as state variables. The constraint leads to a linear system of simultaneous equations involving angular accelerations and torques. A symbolic solution will contain a determinant of the form $J_l + J_m n^2$. The tool thus automatically deduces how inertia is transformed through a gearbox.

3.2.3 Stream connector support

Dymola supports in full the stream connector concept in Modelica Language Specification:

- The prefix `stream`.
- The operators `inStream` and `actualStream`.
- Generation of connection equations.
- Automatic adding of smooth/noEvent for `m_flow*actualStream(h)`.
- The proposed regularization.
- Symbolic check that models are balanced.

For more information, please see the Modelica Language Specification.

3.3 Advanced modeling features

The modeling power of Modelica is great. Some of the more powerful constructs are summarized below.

3.3.1 Vectors, matrices and arrays

Modeling of, for example, multi-body systems and control systems is done conveniently with *matrix equations*. Multi-dimensional arrays and the usual matrix operators and matrix functions are thus supported in Modelica.

The modeling of continuous time transfer function is given below as an example. It uses a restricted model called block having inputs and outputs with given causality. The polynomial coefficients in $a_0 + a_1 s + ... + a_n s^n$ are given as a vector $\{a_0, a_1, ..., a_n\}$.

```
partial block SISO "Single Input/Single Output block"
  input Real u "input";
  output Real y "output";
end SISO;

block TransferFunction
  extends SISO;
  parameter Real a[:]={1, 1} "Denominator";
  parameter Real b[:]={1} "Numerator";

protected
  constant Integer na=size(a, 1);
  constant Integer nb(max=na) = size(b, 1);
  constant Integer n=na-1 "System order";
  Real b0[na] =
    cat(1, b, zeros(na - nb)) "Zero expanded b vector.";
  Real x[n] "State vector";

equation
  // Controllable canonical form
  der(x[2:n]) = x[1:n-1];
  a[na]*der(x[1]) + a[1:n]*x = u;
  y = (b0[1:n] - b0[na]/a[na]*a[1:n])*x + b0[na]/a[na]*u;
end TransferFunction;
```

It is also possible to have arrays of components and to define regular connection patterns. A typical usage is the modeling of a distillation column which consists of a set of trays connected in series.

3.3.2 Class parameters

Component parameters such as resistance values have been discussed. Reuse of model library components is further supported by allowing model class parameters (local replaceable classes).

As an example assume that we would like to replace the PID controller in the motor drive model on page 150 by an auto tuning controller. It is of course possible to just replace the controller in a graphical user environment, i.e., to create a new model. The problem with this solution is that two models must be maintained. Modelica has the capability to instead substitute the model class of certain components using a language construct at the highest hierarchical level, so only one version of the rest of the model is needed. Based on the model MotorDrive on page 150 a model MotorDrive2 with re-declared controller is described as

```
model MotorDrive2 =
  MotorDrive(redeclare AutoTuningPID controller);
```

This is a strong modification of the motor drive model and there is the issue of possible invalidation of the model. The keyword **redeclare** clearly marks such modifications. Furthermore, the new component must be a subtype of PID, i.e., has compatible connectors and parameters. The type system of Modelica is greatly influenced by type theory, in particular the notion of sub typing (the structural relationship that determines type compatibility) which is different from sub classing (the mechanism for inheritance). The main benefit is added flexibility in the composition of types, while still maintaining a rigorous type system. Inheritance is not used for classification and type checking in Modelica.

The public components of a class are typically its connectors and parameters. A model of a PID controller has connectors for the reference signal, measured value and control output and parameters such as gain and integral time. So it is natural to require that also an autotuning controller has those components.

In many real applications there are many PID controllers. This makes it clumsy to use the approach described above to change controllers, because we need to know the names of all controllers. To avoid this problem and prepare for replacement of a set of models, one can define a replaceable class, `ControllerModel` in the drive model:

```
partial block SISOController
  Modelica.Blocks.Interfaces.RealInput ref;
  Modelica.Blocks.Interfaces.RealInput inp;
  Modelica.Blocks.Interfaces.RealOutput out;
end SISOController;

model MotorDrive3
  replaceable block ControllerModel = SISOController;

protected
  ControllerModel controller;
  // then same as MotorDrive.
end MotorDrive3;
```

where the replaceable model ControllerModel is declared to be of type `SISOController`, which means that it will be enforced that the actual class will have the inputs `ref` and `inp`

and the output `out`, but it may be parameterized in any way. Setting `ControllerModel` to for example `PID` is done as

```
model PIDDrive =
  MotorDrive3(redeclare block ControllerModel = PID);
```

3.3.3 Algorithms and functions

Algorithms and functions are supported in Modelica for modeling parts of a system in procedural programming style. Modelica functions have syntax similar to other Modelica classes and matrix expressions can be used. Assignment statements, if statements and loops are available in the usual way. A function for polynomial multiplication is given as an example. It takes two coefficient vectors as inputs and returns the coefficient vector for the product.

```
function polynomialMultiply
  input Real a[:], b[:];
  output Real c[:] = zeros(size(a,1) + size(b, 1) - 1);

algorithm
  for i in 1:size(a, 1) loop
    for j in 1:size(b, 1) loop
      c[i+j-1] := c[i+j-1] + a[i]*b[j];
    end for;
  end for;
end polynomialMultiply;
```

3.4 Hybrid modeling in Modelica

Realistic physical models often contain discontinuities, discrete events or changes of structure. Examples are relays, switches, friction, impact, sampled data systems etc. Modelica has introduced special language constructs allowing a simulator to introduce efficient handling of such events. Special design emphasis was given to synchronization and propagation of events and the possibility to find consistent restarting conditions after an event.

A hybrid Modelica model is described by a set of synchronous differential, algebraic and discrete equations leading to deterministic behavior and automatic synchronization of the continuous and discrete parts of a model. The consequences of this view are discussed and demonstrated at hand of a new method to model ideal switch elements such as ideal diodes ideal thyristors or friction. At event instants this leads to mixed continuous/discrete systems of equations that have to be solved by appropriate algorithms. For modeling of continuous time systems, Modelica provides DAEs to mathematically describe model components. For discrete event systems this is different, because there a single widely accepted description form does not exist. Instead, many formalisms are available, e.g., finite automata, Petri nets, state charts, sequential function charts, DEVS, logical circuits, difference equations, CSP, process-oriented languages that are all suited for particular application areas.

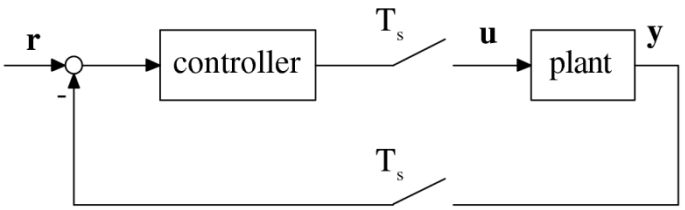
In Modelica the central property is the usage of synchronous differential, algebraic and discrete equations. The idea of using the well-known synchronous data flow principle in the

context of hybrid systems was introduced in Elmqvist (1993). For pure discrete event systems, the same principle is utilized in synchronous languages (Halbwachs, 1993) such as SattLine (Elmqvist, 1992), Lustre (Halbwachs, 1991) and Signal (Gautier *et al.*, 1994), in order to arrive at save implementations of real-time systems and for verification purposes.

3.4.1 Synchronous equations

A hybrid Modelica model basically consists of differential, algebraic and discrete equations.

Sampled data system.



A typical example is given in the figure above where a continuous plant

$$\frac{d\mathbf{x}_p}{dt} = \mathbf{f}(\mathbf{x}_p, \mathbf{u})$$
$$\mathbf{y} = \mathbf{g}(\mathbf{x}_p)$$

is controlled by a digital linear controller

$$\mathbf{x}_c(t_i) = \mathbf{A}\mathbf{x}_c(t_i - T_s) + \mathbf{B}(\mathbf{r}(t_i) - \mathbf{y}(t_i))$$
$$\mathbf{u}(t_i) = \mathbf{C}\mathbf{x}_c(t_i - T_s) + \mathbf{D}(\mathbf{r}(t_i) - \mathbf{y}(t_i))$$

using a zero-order hold to hold the control variable $\mathbf{u}$ between sample instants (i.e., $\mathbf{u}(t) = \mathbf{u}(t_i)$ for $t_i \le t \le t_i + T_s$), where T_s is the sample interval, $\mathbf{x}_p(t)$ is the state vector of the continuous-time plant, $\mathbf{y}(t)$ is the vector of measurement signals, $\mathbf{x}_c(t_i)$ is the state vector of the digital controller and $\mathbf{r}(t_i)$ is the reference input. In Modelica, the complete system can be easily described by connecting appropriate blocks. However, for simplicity of the following discussion, an overall description of the system in one model is used:

```
model SampledSystem
  parameter Real Ts=0.1 "sample interval";
  parameter Real A[:, size(A,1)],
                 B[size(A,1), :],
                 C[:, size(A,2)],
                 D[size(C,1), size(B,2)];
  constant Integer nx = 5;
  input Real r [size(B,2)] "reference";
  output Real y [size(B,2)] "measurement";
  Real u [size(C,1)] "control";
  Real xc[size(A,1)] "discrete state";
  Real xp[nx] "plant state";
equation
  der(xp) = f(xp, u); // plant
```

```
  y = g(xp);
  when sample(0,Ts) then // controller
    xc =A*pre(xc) + B*(r-y);
    u =C*pre(xc) + D*(r-y);
  end when;
end SampledSystem;
```

This Modelica model consists of the continuous equations of the plant and of the discrete equations of the controller within the when clause. Note that **der**(x) defines the time derivative of x. During continuous integration the equations within the when clause are deactivated. When the condition of the when clause *becomes* true an event is triggered, the integration is halted and the equations within the when clause are active at this event instant. The operator sample(...) triggers events at sample instants with sample time T_s and returns true at these event instants. At other time instants it returns false. The values of variables are kept until they are explicitly changed. For example, **u** is computed only at sample instants. Still, **u** is available at all time instants and consists of the value calculated at the last event instant.

Within the controller, the discrete states $\mathbf{x}_c$ are needed both at the actual sample instant $\mathbf{x}_c(t_i)$ and at the previous sample instant $\mathbf{x}_c(t_i - T_s)$, which is determined by using the **pre**(...) operator. Formally, the *left limit* $x(t^-)$ of a variable x at a time instant t is characterized by $\mathbf{pre}(x)$, whereas x itself characterizes the *right limit* $x(t^+)$. Since $\mathbf{x}_c$ is only discontinuous at sample instants, the left limit of $\mathbf{x}_c(t_i)$ at sample instant t_i is identical to the right limit of $\mathbf{x}_c(t_i - T_s)$ at the previous sample instant and therefore $\mathbf{pre}(\mathbf{x}_c)$ characterizes this value.

The synchronous principle basically states that at every time instant, the active equations express relations between variables which have to be fulfilled concurrently. As a consequence, during continuous integration the equations of the plant have to be fulfilled, whereas at sample instants the equations of the plant and of the digital controller hold concurrently. In order to efficiently solve such types of models, all equations are by block-lower-triangular partitioning, the standard algorithm of object-oriented modeling for continuous systems (now applied to a mixture of continuous and discrete equations), under the assumption that all equations are active. In other words, the order of the equations is determined by data flow analysis resulting in an automatic synchronization of continuous and discrete equations. For the example above, sorting results in an ordered set of assignment statements:

```
// "known" variables: r, xp, pre(xc)
y := g(xp);
when sample(0,Ts) then
  xc := A*pre(xc) + B*(r-y);
  u  := C*pre(xc) + D*(r-y);
end when;
der(xp) := f(xp, u);
```

Note, that the evaluation order of the equations is correct both when the controller equations are active (at sample instants) and when they are not active.

The synchronous principle has several consequences: First, the evaluation of the discrete equations is performed in zero (simulated) time. In other words, time is abstracted from the

computations and communications; see also Gautier *et al.* (1994). Second, in order that the unknown variables can be uniquely computed it is necessary that the number of active equations and the number of unknown variables in the active equations at every time instant are identical. This requirement is violated in the following example:

```
equation // incorrect model fragment!
  when h1 < 3 then
    close = true;
  end when;
  when h2 > 1 then
    close = false;
  end when;
```

If by accident or by purpose the relation `h1<3` **and** `h2>1` becomes **true** at the same event instant, we have two conflicting equations for close and it is not defined which equation should be used. In general, it is not possible to detect by source inspection whether conditions becomes true at the same event instant or not. Therefore, in Modelica the assumption is used that all equations in a model may potentially be active at the same time instant during simulation. Due to this assumption, the total number of (continuous and discrete) equations shall be identical to the number of unknown variables. It is possible to rewrite the model above by placing the when clauses in an **algorithm** section and changing the equations into assignment statements:

```
algorithm
  when h1 < 3 then
    close := true;
  end when;

  when h2 > 1 then
    close := false;
end when;
```

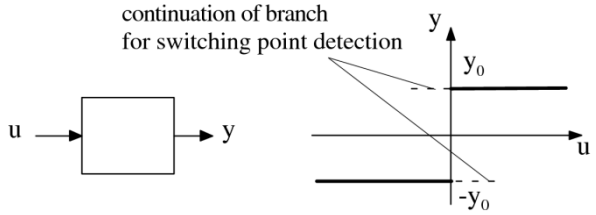
In this case the two when clauses are evaluated in the order of appearance and the second one get higher priority. All assignment statements within the same algorithm section are treated as a set of *n* equations, where *n* is the number of different left hand side variables (e.g., the model fragment above corresponds to one equation). An algorithm section is sorted as a whole together with the rest of the system. Note, that another assignment to close somewhere else in the model would still yield an error.

Handling hybrid systems in this way has the advantage that the synchronization between the continuous time and discrete event parts is automatic and leads to a deterministic behavior without conflicts. Furthermore, some difficulties to detect errors of other approaches, such as deadlock, can often be determined already during translation. Note, that some discrete event formalisms, such as finite automata or prioritized Petri nets, can be formulated in Modelica in a component-oriented way, see Elmqvist *et al.* (2000).

3.4.2 Relation triggered events

During continuous integration it is required that the model equations remain continuous and differentiable, since the numerical integration methods are based on this assumption. This requirement is often violated by if clauses.

A discontinuous component.



For example the simple block above with input *u* and output *y* may be described by the following model:

```
model TwoPoint
  parameter Real y0=1;
  input Real u;
  output Real y;
equation
  y = if u > 0 then y0 else -y0;
end TwoPoint;
```

At point u=0 this equation is discontinuous, if the if-expression would be taken literally. A discontinuity or a non-differentiable point can occur if a relation, such as $x_1 > x_2$ changes its value, because the branch of an if-statement may be changed. Such a situation can be handled in a numerical sound way by detecting the switching point within a prescribed bound, halting the integration, selecting the corresponding new branch, and restarting the integration, i.e., by triggering a state event.

In general, it is not possible to determine by source inspection whether a specific relation will lead to a discontinuity or not. Therefore, by default it is assumed that every relation potentially will introduce a discontinuity or a non-differentiable point in the model. Consequently, relations in Modelica automatically trigger state events (or time events for relations depending only on time) at the time instants where their value is changed. This means, e.g., that model `TwoPoint` is treated in a numerical sound way (the if-expression $u > 0$ is not taken literally but triggers a state event).

In some situations, relations do not introduce discontinuities or non-differentiable points. Even if such points are present, their effect may be small, and it may not affect the integration by just integrating over these points. Finally, there may be situations where a literal evaluation of a relation is required, since otherwise an "outside domain" error occurs, such as in the following example, where the argument of function `sqrt` to compute the square root of its argument is not allowed to be negative:

```
y = if u >=0 then sqrt(u) else 0;
```

This equation will lead to a run time error, because *u* has to become small and negative before the **then**-branch can be changed to the **else**-branch and the square root of a negative real number has no real result value. In such situations, the modeler may explicitly require a *literal* evaluation of a relation by using the operator **noEvent () :**

```
y = if noEvent(u>=0) then sqrt(u) else 0;
```

Modelica has a set of additional operators, such as **initial()** and **terminal()** to detect the initial and final call of the model equations, and **reinit(...)** to reinitialize a continuous state with a

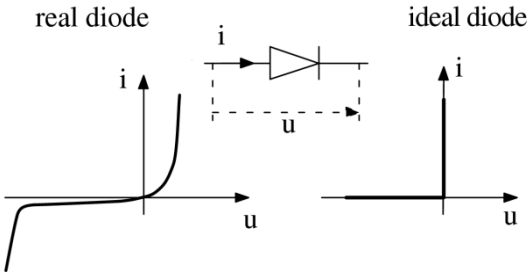
new value at an event instant. For space reasons, these language elements are not discussed. Instead, in the next section some non-trivial applications of the discussed language elements are explained.

3.4.3 Variable structure systems

Parameterized curve descriptions

If a physical component is modeled detailed enough, there are usually no discontinuities in the system. When neglecting some "fast" dynamics, in order to reduce simulation time and identification effort, discontinuities appear in a physical model.

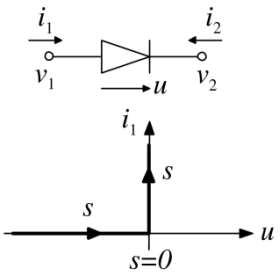
Real and ideal diode characteristic.



As a typical example, a diode is shown in the figure above, where *i* is the current through the diode and *u* is the voltage drop between the pins of the diode. The diode characteristic is shown in the left part of the figure. If the detailed switching behavior is negligible with regards to other modeling effects, it is often sufficient to use the ideal diode characteristic shown in the right part of the figure, which typically gives a simulation speedup of 1 to 2 orders of magnitude.

It is straightforward to model the real diode characteristic in the left part of the figure, because the current *i* have just to be given as a function (analytic or tabulated) of the voltage drop *u*. It is more difficult to model the ideal diode characteristic in the right part of the figure, because the current at $u = 0$ is no longer a function of *u*, i.e., a mathematical description in the form $i = i(u)$ is no longer possible. This problem can be solved by recognizing that a curve can also be described in a parameterized form $i = i(s)$, $u = u(s)$ by introducing a curve parameter *s*. This description form is more general and allows us to describe an ideal diode uniquely in a declarative way as shown in the figure below.

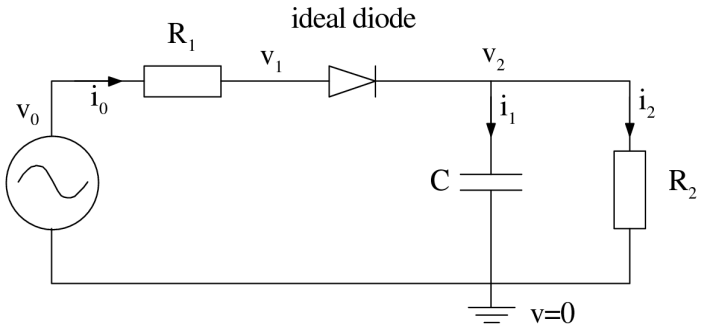
Ideal diode model.



$$0 = i_1 + i_2$$
$$u = v_1 - v_2$$
$$off = s < 0$$
$$u = \textbf{if } off \textbf{ then } s \textbf{ else } 0$$
$$i_1 = \textbf{if } off \textbf{ then } 0 \textbf{ else } s$$

In order to understand the consequences of parameterized curve descriptions, the ideal diode is used in the simple rectifier circuit below.

Simple rectifier circuit.



Collecting the equations of all components and connections, as well as sorting and simplifying the set of equations under the assumption that the input voltage $v_0(t)$ of the voltage source is a known time function and that the states (here: v_2) are assumed to be known, leads to

$$off = s < 0$$
$$u = v_1 - v_2$$
$$u = \textbf{if } off \textbf{ then } s \textbf{ else } 0$$
$$i_0 = \textbf{if } off \textbf{ then } 0 \textbf{ else } s$$
$$R_1 \cdot i_0 = v_0(t) - v_1$$

$$i_2 = \frac{v_2}{R_2}$$
$$i_1 = i_0 - i_2$$
$$\frac{dv_2}{dt} = \frac{i_1}{C}$$

The first 5 equations are coupled and build a system of equations in the 5 unknowns off, s, u, v_1 and i_0. The remaining equations are used to compute i_2, i_1 and the state derivative $\frac{dv_2}{dt}$. During continuous integration the Boolean variables, i.e., *off*, are fixed and the Boolean equations are not evaluated. In this situation, the first equation is not touched and the next 4 equations form a *linear* system of equations in the 4 unknowns s, u, v_1, i_0 which can be solved by Gaussian elimination. An event occurs if one of the relations (here: $s < 0$) changes its value.

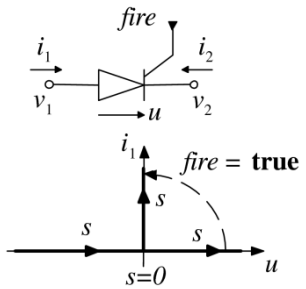
At an event instant, the first 5 equations are a mixed system of discrete and continuous equations which cannot be solved by, say, Gaussian elimination, since there are Real and Boolean unknowns. However, appropriate algorithms can be constructed: (1) Make an assumption about the values of the relations in the system of equations. (2) Compute the discrete variables. (3) Compute the continuous variables by Gaussian elimination (discrete variables are fixed). (4) Compute the relations based on the solution of (2) and (3). If the relation values agree with the assumptions in (1), the iteration is finished and the mixed set of equations is solved. Otherwise, new assumptions on the relations are necessary, and the iteration continues. Useful assumptions on relation values are for example:

- Use the relation values computed in the last iteration.
- Try all possible combinations of the values of the relations systematically (exhaustive search).

In the above example, both approaches can be simply applied, because there are only two possible values ($s < 0$ is false or true). However, if n switches are coupled, there are n relations and therefore 2^n possible combinations which have to be checked in the worst case.

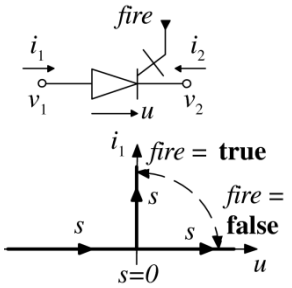
Below parameterized curve descriptions of the ideal thyristor and the ideal GTO thyristor are shown for further demonstration. Especially note that also non-unique curve parameters s can be used by introducing additional discrete variables (here: *fire*) to distinguish the branches with the same parameterization values.

Ideal Thyristor.



$$0 = i_1 + i_2$$
$$u = v_1 + v_2$$
$$off = s < 0 \textbf{ or } \textbf{pre}(off) \textbf{ and not } fire$$
$$u = \textbf{if } off \textbf{ then } s \textbf{ else } 0$$
$$i_1 = \textbf{if } off \textbf{ then } 0 \textbf{ else } s$$

Ideal GTO thyristor



$$0 = i_1 + i_2$$
$$u = v_1 + v_2$$
$$off = s < 0 \textbf{ or not } fire$$
$$u = \textbf{if } off \textbf{ then } s \textbf{ else } 0$$
$$i_1 = \textbf{if } off \textbf{ then } 0 \textbf{ else } s$$

The technique of parameterized curve descriptions was introduced in Clauß *et al.* (1995) and a series of related papers. However, no proposal was yet given how to actually implement such models in a numerically sound way. In Modelica the (new) solution method follows logically because the equation based system naturally leads to a system of mixed continuous/discrete equations which have to be solved at event instants.

In the past, ideal switching elements have been handled by (a) using variable structure equations which are controlled by finite automata to describe the switching behavior, see e.g., Barton (1992), Elmqvist *et al.* (1993), and Mosterman *et al.* (1996), or by (b) using a complementarily formulation, see e.g. Lötstedt (1982) and Pfeiffer and Glocker (1982). The approach (a) has the disadvantage that the continuous part is described in a declarative way but not the part describing the switching behavior. As a result, e.g., algorithms with better convergence properties for the determination of a consistent switching structure cannot be used. Furthermore, this involves a global iteration over all model equations whereas parameterized curve descriptions lead to local iterations over the equations of the involved elements. The approach (b) seems to be difficult to use in an object-oriented modeling language and seems to be applicable only in special cases (e.g. it seems not possible to describe ideal thyristors).

3.5 Initialization of models

A dynamic model describes how the states evolve with time. The states are the memory of the model, for example in mechanical systems positions and velocities. When starting a simulation, the states need to be initialized.

For an ordinary differential equation, ODE, in state space form, d**x**/dt = **f**(**x**, t), the state variables, **x**, are free to be given initial values. However, more flexibility in specifying initial conditions than setting state variables is needed. In many cases we would like to start at steady state implying that the user specifies d**x**/dt = 0 as initial condition to get the initial values of **x** calculated automatically by solving **f**(**x**, t) = 0. Besides the states, a model has also other variables and in many cases it is natural to specify initial conditions in terms of these variables.

Modelica provides powerful language constructs for specifying initial conditions. They permit flexible specification of initial conditions as well as the correct solution of difficult,

non-standard initialization problems occurring in industrial applications. Modelica provides a mathematically rigid specification of the initialization of hybrid differential algebraic equations.

Dymola manipulates symbolically the initialization problem and generates analytic Jacobians for nonlinear sub problems to make the solution of the initialization problem more robust and reliable. Moreover, the special analysis of the initialization problem allows Dymola to give diagnosis and user guidance when the initialization problem turns out not to be well posed.

3.5.1 Basics

Before any operation is carried out with a Modelica model, especially simulation, initialization takes place to assign consistent values for all variables present in the model. During this phase, also the derivatives, **der**(…), and the **pre**-variables, **pre**(…), are interpreted as unknown algebraic variables. To obtain consistent values, the initialization uses all equations and algorithms that are utilized during the simulation.

Additional constraints necessary to determine the initial values of all variables can be provided in two ways:

- Start values for variables
- Initial equations and initial algorithms

For clarity, we will first focus on the initialization of continuous time problems because there are some differences in the interpretation of the start values of continuous time variables and discrete variables. Also there are special rules for the usage of when clauses during initialization. All this makes it simpler to start discussing pure continuous time problems and after that discuss discrete and hybrid problems.

3.5.2 Continuous time problems

Initial equations and algorithms

Variables being subtypes of Real have an attribute *start* allowing specification of a start value for the variable

```
Real v(start = 2.0);
parameter Real x0 = 0.5;
Real x(start = x0);
```

The value for start shall be a parameter expression.

There is also another Boolean attribute *fixed* to indicate whether the value of start is a guess value (fixed = **false**) to be used in possible iterations to solve nonlinear algebraic loops or whether the variable is required to have this value at start (fixed = **true**). For constants and parameters, the attribute fixed is by default true, otherwise fixed is by default false. For a continuous time variable, the construct

```
Real x(start = x0, fixed = true);
```

implies the additional initialization equation

```
x = x0;
```

Thus, the problem

```
  parameter Real a = -1, b = 1;
  parameter Real x0 = 0.5;
  Real x(start = x0, fixed = true);
equation
  der(x) = a*x + b;
```

has the following solution at initialization

```
a      := -1;
b      := 1;
x0     := 0.5;
x      := x0;        // = 0.5
der(x):= a*x + b;   // = 0.5
```

Start and fixed attributes

A model may have the new sections **initial equation** and **initial algorithm** with additional equations and assignments that are used solely in the initialization phase. The equations and assignments in these initial sections are viewed as pure algebraic constraints between the initial values of variables and possibly their derivatives. It is not allowed to use when clauses in the initial sections.

Steady state

To specify that a variable `x` shall start in steady state, we can write

```
initial equation
  der(x) = 0;
```

A more advanced example is

```
  parameter Real x0;
  parameter Boolean steadyState;
  parameter Boolean fixed;
  Real x;

initial equation
  if steadyState then
    der(x) = 0;
  elseif fixed then
    x = x0;
  end if;
```

If the parameter steadyState is **true**, then `x` will be initialized at steady state, because the model specifies the initialization equation

```
initial equation
  der(x) = 0;
```

If the parameter steadyState is **false**, but fixed is **true** then there is an initialization equation

```
initial equation
  x = x0;
```

If both steadyState and fixed are **false**, then there is no initial equation.

The approach as outlined above, allows x0 to be any expression. When x0 is a parameter expression, the specification above can also be given shorter as

```
  parameter Real x0;
  parameter Boolean steadyState;
  parameter Boolean fixed;
  Real x(start = x0, fixed = fixed and not steadyState);
initial equation
  if steadyState then
    der(x) = 0;
  end if;
```

Mixed Conditions

Due to the flexibility in defining initialization equations in Modelica, it is possible to formulate more general initial conditions: For example, an aircraft needs a certain minimum velocity in order that it can fly. Since this velocity is a state, a useful initialization scheme is to provide an initial velocity, i.e., an initial value for a *state*, and to set all other *state derivatives* to zero. This means, that a mixture of initial states and initial state derivatives is defined.

How many initial conditions?

How many initial conditions are needed for a continuous time problem?

For an ordinary differential equation, ODE, in state space form, $d\mathbf{x}/dt = \mathbf{f}(\mathbf{x}, t)$, *exactly* dim(**x**) additional conditions are needed, in order to arrive at 2*dim(**x**) equations for the 2*dim(**x**) unknowns $\mathbf{x}(t_0)$ and $d\mathbf{x}/dt(t_0)$.

The situation is more complex for a system of differential algebraic equations, DAE,

$0 = \mathbf{g}(d\mathbf{x}/dt, \mathbf{x}, \mathbf{y}, t)$

where **x**(t) are variables appearing differentiated, **y**(t) are algebraic variables and dim(**g**) = dim(**x**) + dim(**y**). Here it can only be stated that at most dim(**x**) additional conditions **h**(..) are needed in order to arrive at 2*dim(**x**)+dim(**y**) equations for the same number of unknowns, $d\mathbf{x}/dt(t_0)$, $\mathbf{x}(t_0)$, $\mathbf{y}(t_0)$:

$$0 = \begin{bmatrix} \mathbf{g}(\dot{\mathbf{x}}(t_0), \mathbf{x}(t_0), \mathbf{y}(t_0), \mathbf{t}_0) \\ \mathbf{h}(\dot{\mathbf{x}}(t_0), \mathbf{x}(t_0), \mathbf{y}(t_0), \mathbf{t}_0) \end{bmatrix}$$

The reason is that the DAE problem may be a higher index DAE problem, implying that the number of continuous time states is less than dim(**x**).

It may be difficult for a user of a large model to figure out how many initial conditions have to be added, especially if the system has higher index. At translation Dymola performs an index reduction and selects state variables. Thus, Dymola establishes how many states there are. If there are too many initial conditions, Dymola outputs an error message indicating a set of initial equations or fixed start values from which initial equations must be removed or start values inactivated by setting fixed = **false**.

If initial conditions are missing, Dymola makes automatic default selection of initial conditions. The approach is to select continuous time states with inactive start values and make their start values active by turning their fixed attribute to **true** to get a structurally well posed initialization problem. A message informing about the result of such a selection can be obtained.

For additional information, see the index entry "*initialization of models : over-specified*" in the index in the end the manual.

Interactive setting of start values

The initial value dialogue of the Dymola window includes the continuous time variables having active start values (i.e., fixed=**true**) where the start values are also being literals. Setting parameters may of course influence an active start value bound to a parameter expression.

Dymola generates a warning when setting variables from scripts if setting the variable has no effect what so ever, e.g. if it is a structural parameter.

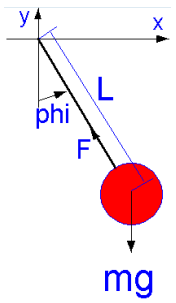
Non-linear algebraic loops

A non-linear algebraic problem may have several solutions. During simulation a numerical DAE solver tends to give the smoothest solution. A DAE solver is assumed to start at a consistent point and its task is to calculate a new point along the trajectory. By taking a sufficiently small step and assuming the existence of a Jacobian that is non-singular there is a local well-defined solution.

The initialization task is much harder and precautions must be taken to assure that the correct solution is obtained. The means to guide the solver are by including min and max values as well as start values for the unknowns.

As a simple example, consider a planar pendulum with fixed length L.

A planar pendulum.



The position of the pendulum can be given in polar coordinates. Introduce an angle phi that is zero, when the pendulum is hanging downward in its rest position.

The model can be given as:

```
parameter Real g = 9.81;
parameter Real m = 1;
parameter Real L = 1;
```

```
  Real phi, w;

equation
  der(phi) = w;
  m*der(w) = -(m*g/L)*sin(phi);
```

Assume that we want to specify the initial condition in Cartesian coordinates defined as

```
x =  L*sin(phi);
y = -L*cos(phi);
```

If we define

```
Real y(start = 0; fixed = true);
```

the pendulum will start in a horizontal position. However, there are two horizontal positions, namely

```
x = -L and x = L
```

To indicate preference for a positive value for x, we can define

```
Real x(start = L);
```

It means that we provide a guess value for numerical solvers to start from. In this case, the solver will hopefully find the positive solution for x, because it is closer to L than the negative solution.

For the angle phi there are many values giving the desired position, because adding or subtracting 2π gives the same Cartesian position. Also, here the start value can be used to indicate the desired solution. How critical it is to get a special solution depends of course on what phi will be used for in the model and the aim of the simulation. If no start value is given zero is used.

For additional information, see the the index entry "*start values : discriminating*" in the index in the end of the manual.

3.5.3 Parameter values

Parameters are typically given values in a model through definition equation or set interactively before a simulation. Modelica also allows parameter values to be given implicitly in terms of the initial values of all variables.

Recall the planar pendulum and assume that we would like to specify the initial position as

```
Real x(start = 0.3; fixed = true);
Real y(start = 0.4; fixed = true);
```

This means that we in fact also specify the length of the pendulum to be 0.5. To specify that the parameter L shall be calculated from the initial conditions, we define it as

```
parameter Real L(fixed = false);
```

Recall that the attribute fixed is by default **true** for constants and parameters, otherwise the attribute fixed is by default **false**.

The semantics of parameters in Modelica is a variable that is constant during simulation. The possibility to let the parameter value to depend on the initial values of time dependent (continuous-time or discrete) variables does not violate this semantics.

This feature has many useful applications. It allows powerful re-parameterizations of models. As an example, consider the model of an ideal resistor. It has one parameter, R, being the resistance. Assume that we would like to have use it as a resistive load with a given power dissipation at a steady state operating point. It is just to extend from the resistor model given in the Modelica Standard Library and do the following:

1. Add a parameter P0 to specify the power dissipation.
2. Set fixed=**false** for parameter R.
3. Add an initial equation section with v*i = P0.

In power systems, it is common practice to specify initial conditions in steady state and use different kind of load models including resistive load and specify their steady state operating conditions in terms of active and reactive power dissipation.

In some cases parameters may be provided outside of a Modelica model and the actual values may be read from file or parameter values may be inquired from a database system during initialization:

```
  parameter Real A(fixed=false);
  parameter Real w(fixed=false);
  Real x;

initial equation
  (A,w) = readSineData("init.txt");

equation
  der(x) = -A*sin(w*x);
```

When translating a model, it is checked that each parameter has a value. A parameter declared with the attribute fixed = false will be calculated (evaluated) at initialization from the initial equations or initial algorithms. In a general model library, it may be difficult to provide good values. Modelica allows the value of the attribute start to be used as a default value. Otherwise parameters must be given values though bindings.

3.5.4 Discrete and hybrid problems

The language constructs for specifying initial conditions for discrete variables are as for the continuous time variables: start values and initial equations and algorithms.

Variables being subtypes of Real, Integer, Boolean and String have an attribute start allowing specification of a start value for the variable.

For discrete variables declarations

```
Boolean  b(start  = false, fixed = true);
Integer  i(start  = 1,     fixed = true);
```

imply the additional initialization equations

```
pre(b) = false;
pre(i) = 1;
```

This means that a discrete variable v itself does not get an initial value $(= v(t_0 + \varepsilon))$, but the **pre**-value of v $(= v(t_0 - \varepsilon))$ does.

When clauses at initialization

For the initialization problem there are special semantic rules for **when** clauses appearing in the model. During simulation a **when** clause is only active when its condition becomes **true**. During initialization the equations of a **when** clause are only active during initialization, if the **initial**() operator explicitly enables it.

```
when {initial(), condition1, ...} then
  v = ...
end when;
```

Otherwise a **when** clause is in the initialization problem replaced by v = **pre**(v) for all its left hand side variables, because this is also the used equation during simulation, when the **when**-clause is not active.

Non-unique initialization

In certain situations an initialization problem may have an infinite number of solutions, even if the number of equations and unknown variables are the same during initialization. Examples are controlled systems with friction, or systems with backlash or dead-zones. Assume for example backlash is present. Then, all valid positions in this element are solutions of steady state initialization, although this position should be computed from initialization. It seems best to not rely on some heuristics of the initializer to pick one of the infinite numbers of solutions. Instead, the continuous time equations may be modified during initialization in order to arrive at a unique solution. Example:

```
y = if initial() then
      // smooth characteristics
    else
      // standard characteristics
```

Well-posed initialization

At translation Dymola analyses the initialization problem to check if it is well posed by splitting the problem into four equation types with respect to the basic scalar types Real, Integer, Boolean and String and decides whether each of them are well-posed.

As described for the pure continuous-time problem, Dymola outputs error diagnosis in case of over specified problems. In case of under specified problems Dymola makes automatic default selection of initial conditions.

How many initial conditions?

Basically, this is very simple: Every discrete variable v needs an initial condition, because $v(t_0 - \varepsilon)$ is otherwise not defined. Example:

```
  parameter Real t1 = 1;
  discrete Real u(start=0, fixed=true);
  Real x(start=0, fixed=true);

equation
  when time > t1 then
    u = ...
  end when;
  der(x) = -x + u;
```

During initialization and before the **when**-clause becomes active the first time, u has not yet been assigned a value by the **when**-clause although it is used in the continuous part of the model. Therefore, it would be an error, if **pre**(u) would not have been defined via the start value in the u declaration.

On the other hand, if u is used solely inside this **when**-clause and **pre**(u) is not utilized in the model, an initial value for u may be provided but *does not influence* the simulation, because the first access of u computes u in the **when**-clause and afterwards u is utilized in other equations inside the **when**-clause, i.e., the initial value is never used.

Since it may be tedious for a modeler to provide initial values for all discrete variables, Modelica only requires specifying initial values of discrete variables which influence the simulation result. Otherwise, a default value may be used.

3.5.5 Example: Initialization of discrete controllers

Below four variants to initialize a simple plant controlled by a discrete PI controller are discussed.

Variant 1: Initial values are given explicitly

```
  parameter Real k=10, T=1 "PI controller parameters";
  parameter Real Ts = 0.01 "Sample time";
  input     Real xref  "Reference input";
            Real x (fixed=true, start=2);
  discrete  Real xd(fixed=true, start=0);
  discrete  Real u (fixed=true, start=0);

equation
  // Plant model
  der(x) = -x + u;
  // Discrete PI controller

  when sample(0, Ts) then
    xd = pre(xd) + Ts/T*(xref - x);
     u = k*(xd + xref - x);
  end when;
```

The model specifies all the initial values for the states explicitly. The **when**-clause is not enabled at initialization but it is replaced by

```
xd      := pre(xd)
u       := pre(u)
```

The initialization problem is thus

```
x       := x.start  // = 2
pre(xd) := xd.start // = 0
pre(u)  := u.start  // = 0
xd      := pre(xd)  // = 0
u       := pre(u)   // = 0
der(x)  := -x + u   // = -2
```

Variant 2: Initial values are given explicitly and the controller equations are used during initialization. It is as Variant 1, but the **when**-clause is enabled

```
// Same declaration as variant 1
equation
  der(x) = -x + u;

  when {initial(), sample(0,Ts)} then
    xd = pre(xd) + Ts/T*(xref - x);
    u  = k*(xd + xref - x);
  end when;
```

It means that the **when**-clause appears as

```
xd = pre(xd) + Ts/T*(xref - x);
u  = k*(xd + xref - x);
```

in the initialization problem, which becomes

```
x      := x.start  // = 2
pre(xd):= xd.start // = 0
pre(u) := u.start  // = 0
xd     := pre(xd) + Ts/T*(xref - x);
u      := k*(xd + xref - x);
der(x) := -x + u;
```

Variant 3: As Variant 2 but initial conditions defined by initial equations

```
discrete  Real xd;
discrete  Real u;

// Remaining declarations as in variant 1

equation
  der(x) = -x + u;
  when {initial(), sample(0, TS)} then
    xd = pre(xd) + Ts/T*(xref - x);
    u  = k*(xd + xref - x);
  end when;
initial equation
  pre(xd) = 0;
  pre(u)  = 0;
```

leads to the following equations during initialization

```
x      := x.start  // = 2
pre(xd):= 0
pre(u) := 0
```

```
xd      := pre(xd) + Ts/T*(xref - x)
u       := k*(xd + xref - x)
der(x)  := -x + u;
```

Variant 4: Steady state initialization

Assume that the system is to start in steady state. For continuous time state, x, it means that its derivative shall be zero; **der**(x) = 0; While for the discrete state, xd, it means **pre**(xd) = xd; and the when clause shall be active during initialization

```
Real           x (start=2);
discrete  Real xd;
discrete  Real u;

// Remaining declarations as in Variant 1

equation
  // Plant model
  der(x) = -x + u;
  // Discrete PID controller
  when {initial(), sample(0, Ts)} then
    xd = pre(xd) + Ts/T*(x - xref);
    u  = k*(xd + x - xref);
  end when;

initial equation
  der(x)  = 0;
  pre(xd) = xd;
```

The initialization problem becomes

```
der(x) := 0

// Linear system of equations in the unknowns:
//  xd, pre(xd), u, x

pre(xd) = xd
xd      = pre(xd) + Ts/T*(x - xref)
u       = k*(xd + xref - x)
der(x)  = -x + u;
```

Solving the system of equations leads to **der**(x) := 0

```
x       := xref
u       := xref
xd      := xref/k
pre(xd) := xd
```

3.6 Standard libraries

3.6.1 Modelica Standard Library

In order that Modelica is useful for model exchange, it is important that libraries of the most commonly used components are available, ready to use, and sharable between applications. For this reason, an extensive base library is developed together with the Modelica language from the Modelica Association, see http://www.Modelica.org. It is called Modelica Standard Library and it is an intrinsic part of Modelica. It provides constants, types, connectors, partial models and model components in various disciplines. Predefined quantity types and connectors are useful for standardization of the interfaces between components and achieve model compatibility without having to resort to explicit co-ordination of modeling activities. Component libraries are mainly derived from already existing model libraries from various object-oriented modeling systems. They are realized by specialists in the respective area, taking advantage of the new features of Modelica not available in the original modeling system. The Modelica Standard Library consists currently of the following sub-libraries

Blocks	Basic input/output control blocks (continuous, discrete, logical, table blocks).
ComplexBlocks	Basic input/output control blocks with complex signals.
Clocked	Clock triggered blocks for describing synchronous behavior suited for implementation of control systems.
StateGraph	Library to model discrete event and reactive systems by hierarchical state machines. Please note that a more advanced library is available using the command **File > Libraries > State Graph**.
Electrical	Electrical models (analog, digital, machines, multi-phase).
Magnetic	Magnetic components to build especially electro-magnetic devices.
Mechanics	Library to model 1-dimentional and 3-dimentional mechanical systems (multi-body, rotational, translational).
Fluid	Components to model 1-dimensional thermo-fluid flow in network of vessels, pipes, fluid machines, valves and fittings. All media from Modelica.Media can be used.
Media	Property models of media.
Thermal	Library to model thermal systems (heat transfer, thermo-fluid pipe flow).
Math	Mathematical functions (e.g. sin, cos) and operations on matrices (e.g. norm, solve, eig, exp).
ComplexMath	Complex mathematical functions (e.g. sin, cos) and functions operating on complex vectors.
Utilities	Utility functions especially for scripting (operate on files, streams, strings, systems).
Constants	Mathematical and physical constants (pi, eps, h, R, sigma…)
Icons	Icon definitions of general interest.
Units	Library of type and unit definitions. SI units and non-SI units are included, as well as conversions between them.

3.6.2 Modelica Reference

This is a free library describing the element of the Modelica Language. The library is solely documentation, meaning that it is not possible to drag components from it.

3.6.3 Other libraries

For a more extensive list of libraries available, please see http://www.modelica.org/libraries. Please also see chapter "Getting started with Dymola", section "Other libraries".

3.7 References

Barton P.I. (1992): *The Modeling and Simulation of Combined Discrete/Continuous Processes.* Ph.D. Thesis, University of London, Imperial College.

Barton, P. and C.C. Pantelides (1994): "Modeling of combined discrete/continuous processes." AIChE J. , 40, pp. 966–979.

Benner, P., V. Mehrmann, V. Sima, S. Van Huffel, and A. Varga (1998): "SLICOT – A subroutine library in systems and control theory." In Datta, Ed., Applied and Computational Control, Signals and Circuits, vol. 1. Birkhäuser.

Breunese, A. P. and J. F. Broenink (1997): "Modeling mechatronic systems using the SIDOPS+ language."In Proceedings of ICBGM'97, 3rd International Conference on Bond Graph Modeling and Simulation, Simulation Series, Vol.29, No.1, pp. 301–306. The Society for Computer Simulation International.

Clauß C., J. Haase, G.]urth, and P. Schwarz (1995): "Extended Amittance Description of Nonlinear n-Poles.'' Archiv für Elektronik und Übertragungstechnik / International Journal of Electronics and Communications, 40, pp. 91-97.

Elmqvist, H. (1978): A Structured Model Language for Large Continuous Systems. PhD thesis TFRT-1015, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Elmqvist H. (1992): "An Object and Data-Flow based Visual Language for Process Control.'' ISA / 92-Canada Conference & Exhibit, Instrument Society of America, Toronto.

Elmqvist H., F. E. Cellier, and M. Otter (1993): "Object–Oriented Modeling of Hybrid Systems.'' Proceedings ESS'93, European Simulation Symposium, pp. xxxi-xli, Delft, The Netherlands.

Elmqvist H., B. Bachmann, F. Boudaud, J. Broenink, D. Brück, T. Ernst, R. Franke, P. Fritzson, A. Jeandel, P. Grozman, K. Juslin, D. Kagedahl, M. Klose, N. Loubere, S.E. Mattsson, P. Mosterman, H. Nilsson, M. Otter, P. Sahlin, A. Schneider, H. Tummescheit, and H. Vangheluwe (1998): Modelica TM – A Unified Object-Oriented Language for Physical Systems Modeling, Version 1.1, 1998. Modelica homepage: http: //www.modelica.org / .

Elmqvist H., S.E. Mattsson, and M. Otter (1999): “Modelica – A language for Physical System Modeling, Visualization and Interaction.’’ Proceedings of 1999 IEEE Symposium on Computer-Aided Control System Design, CACSD'99, Plenary talk, Hawaii, USA.

Elmqvist H., S. E. Mattsson, and M. Otter (2000): “Object–Oriented Modeling and Hybrid Modeling in Modelica.’’ Proceedings of the 4th International Conference on Automation of Mixed Processes: Hybrid Dynamic Systems, ADPM2000, pp. 7-16, DASA, Dortmund, Germany.

Engelson, V., H. Larsson, and P. Fritzon (1999): “Design, simulation and visualization environment for object-oriented mechanical and mult-domain models in Modelica.” In Proceedings of the IEEE International Conference on Information Visualisation. IEEE Computer Society, London, UK.

Fritzson, P., L. Viklund, D. Fritzson, and J. Herber (1995): “High-level mathematical modeling and programming.” IEEE Software, 12:3.

Gautier T., P. Le Guernic, and O. Maffeis (1994): “For a New Real-Time Methodology.’’ Publication Interne No. 870, Institut de Recherche en Informatique et Systemes Aleatoires, Campus de Beaulieu, 35042 Rennes Cedex, France.

Halbwachs N., P. Caspi, P. Raymond, and D. Pilaud (1991): “The synchronous data flow programming language LUSTRE.’’ Proc. of the IEEE, 79(9), pp. 1305–1321.

Halbwachs N. (1993): Synchronous Programming of Reactive Systems. Kluwer.

IEEE (1997): “Standard VHDL Analog and Mixed-Signal Extensions.” Technical Report IEEE 1076.1. IEEE.

Jeandel, A., F. Boudaud, P. Ravier, and A. Buhsing (1996): “U.L.M: Un Langage de Modélisation, a modelling language.” In Proceedings of the CESA’96 IMACS Multiconference. IMACS, Lille, France.

Joos, H.-D. (1999): “A methodology for multi-objective design assessment and flight control synthesis tuning.” Aerospace Science and Technology, 3.

Kloas, M., V. Friesen, and M. Simons (1995): “Smile — A simulation environment for energy systems.” In Sydow, Ed., Proceedings of the 5th International IMACS-Symposium on Systems Analysis and Simulation (SAS’95), vol. 18–19 of Systems Analysis Modelling Simulation, pp. 503–506. Gordon and Breach Publishers.

Lötstedt P. (1982): “Mechanical systems of rigid bodies subject to unilateral constraints.” SIAM J. Appl. Math., Vol. 42, No. 2, pp. 281-296.

Mattsson, S. E., M. Andersson, and K. J. Åström (1993): “Object-oriented modelling and simulation.” In Linkens, Ed., CAD for Control Systems, chapter 2, pp. 31–69. Marcel Dekker Inc, New York.

Mattsson, S. E., H. Elmqvist, and M. Otter (1998): “Physical system modeling with Modelica.” Control Engineering Practice, 6, pp. 501–510.

Mattsson, S. E. and G. Söderlind (1993): “Index reduction in differential-algebraic equations using dummy derivatives.” SIAM Journal of Scientific and Statistical Computing, 14:3, pp. 677–692.

Mosterman P. J., and G. Biswas (1996): "A Formal Hybrid Modeling Scheme for Handling Discontinuities in Physical System Models." Proceedings of AAAI-96, pp. 905-990, Portland, OR.

Mosterman, P. J., M. Otter, and H. Elmqvist (1998): "Modeling Petri nets as local constraint equations for hybrid systems using Modelica." In Proceedings of the 1998 Summer Simulation Conference, pp. 314–319. Society for Computer Simulation International, Reno, Nevada, USA.

Otter, M., H. Elmqvist, and S. E. Mattsson (1999) : "Hybrid modeling in Modelica based on the synchronous data flow principle." In Proceedings of the 1999 IEEE Symposium on Computer-Aided Control System Design, CACSD'99. IEEE Control Systems Society, Hawaii, USA.

Pantelides, C. (1988): "The consistent initialization of differential-algebraic systems." SIAM Journal of Scientific and Statistical Computing, 9, pp. 213–231.

Pfeiffer F., and C. Glocker (1996): Multibody Dynamics with Unilateral Contacts. John Wiley.

Piela, P., T. Epperly, K. Westerberg, and A. Westerberg (1991): "ASCEND: An object-oriented computer environment for modeling and analysis: the modeling language." Computers and Chemical Engineering, 15:1, pp. 53–72.

Sahlin, P., A. Bring, and E. F. Sowell (1996): "The Neutral Model Format for building simulation, Version 3.02." Technical Report. Department of Building Sciences, The Royal Institute of Technology, Stockholm, Sweden.

Tummescheit, H. and J. Eborn (1998): "Design of a thermo-hydraulic model library in Modelica." In Zobel and Moeller, Eds., Proceedings of the 12th European Simulation Multiconference, ESM'98, pp. 132–136. Society for Computer Simulation International, Manchester, UK.

Åström, K. J., H. Elmqvist, and S. E. Mattsson (1998): "Evolution of continuous-time modeling and simulation." In Zobel and Moeller, Eds., Proceedings of the 12th European Simulation Multiconference, ESM'98, pp. 9–18. Society for Computer Simulation International, Manchester, UK.

4 DEVELOPING A MODEL

4 Developing a model

This chapter describes the Dymola environment for developing models. (The next chapter describes how to use Dymola for simulation of models.) Please observe that this chapter is closely related to the chapter "Getting started with Dymola".

The content is the following:

In section 4.1 **"Windows, tabs and browsers available when developing a model"** starting on page 186 the windows, tabs, and browsers available in Dymola when developing a model are described as an overview. More information about the use of those is given in the following sections.

Section 4.2 **"Basic model editing"** starting on page 217 describes the basics of how to edit a model (creation/editing of components, connectors, parameters/variables, graphical objects, equations and documentation). Finally, the selection of active simulation model and the support for national characters is also described here.

Section 4.3 **"Advanced model editing"** starting on page 329 presents powerful tools for a more advanced model editing.

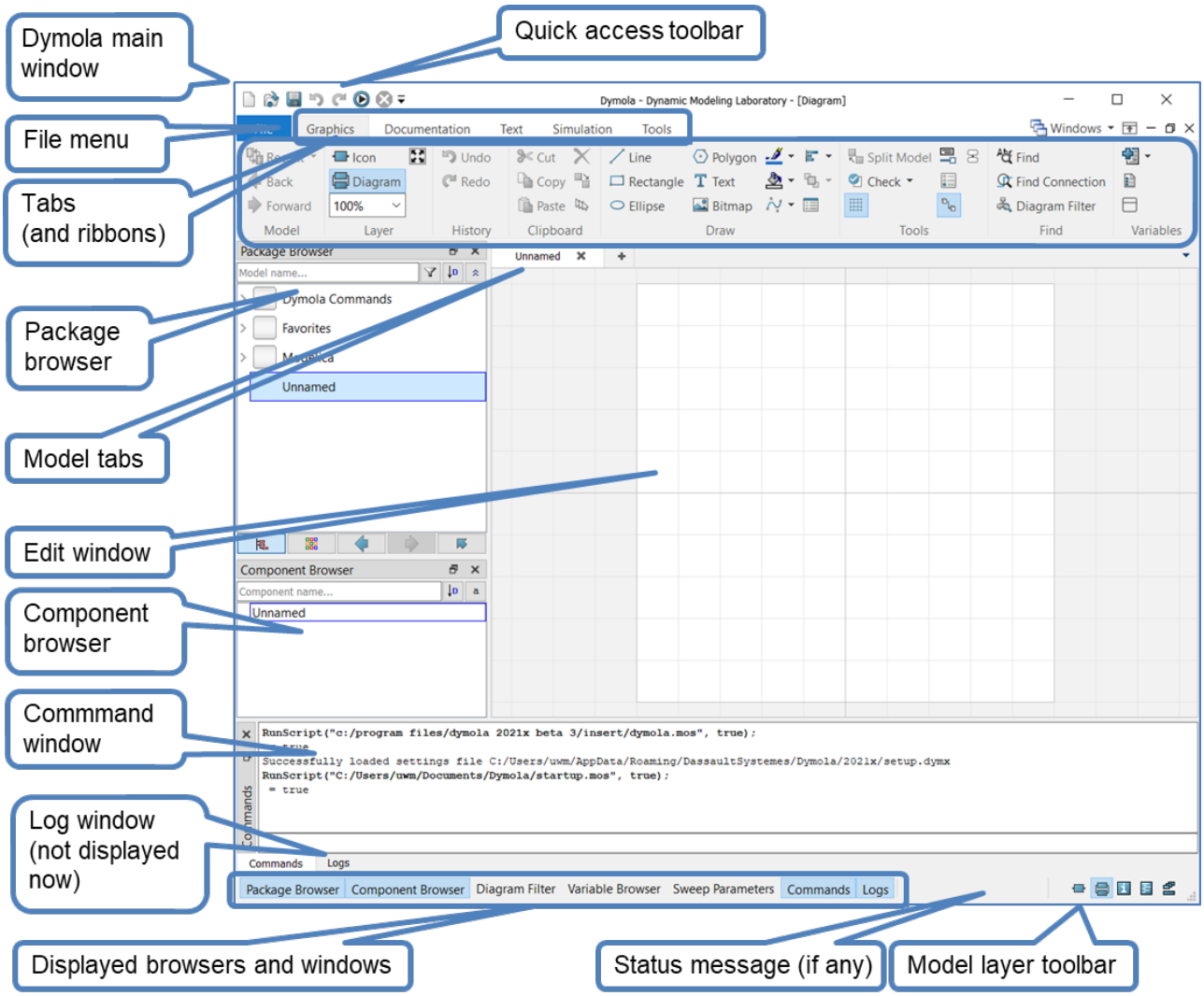
Section 4.4 "**Checking the model**" starting on page 437 describes the testing of the model. Unit checking and unit deduction is dealt with here.

Section 4.5 "**Editing model reference**" starting on page 443 presents some items of interest for engineers building their own libraries.

Section 4.6 "**Editor command reference when developing a model**" starting on page 446 is a reference section, describing the menus available when modeling. The menus are described in the order they appear in certain windows, starting with Dymola Main window. The menus of other windows follow, followed by context menus for the windows. Context menus for components, graphical objects etc. are described, as well as specific menus for graphical objects (line and fill style). The section is concluded by some windows not often used when developing a model.

4.1 Windows, tabs and browsers available when developing a model

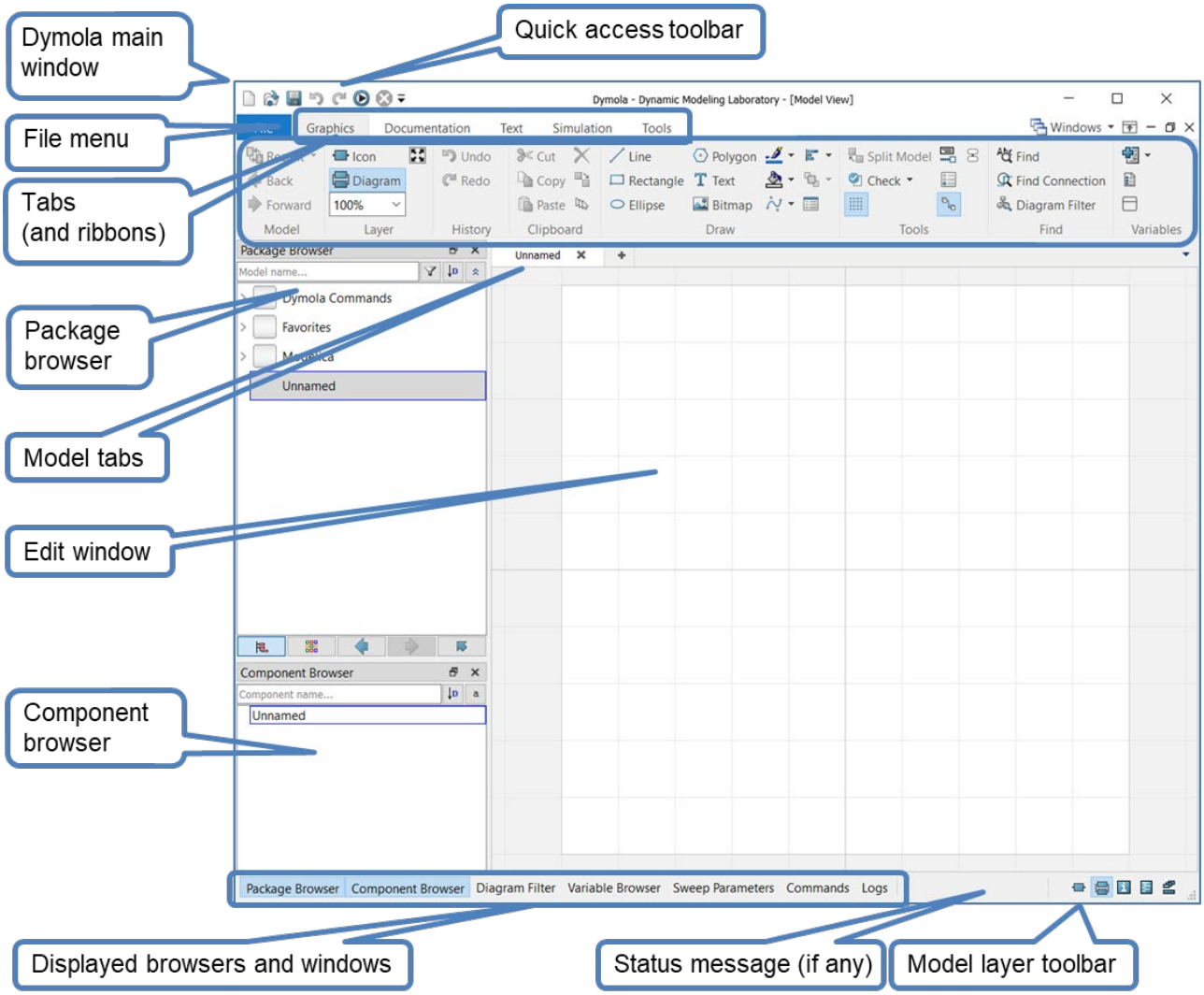
A number of window, tabs, and browsers are available in Dymola when working with modeling. The first time Dymola is started, it looks the following:



What is seen is the Dymola main window that contains a number of browsers and windows.

The user can decide what browsers and windows should be seen, and some general settings concerning windows. Please see section “Window settings” on page 444 for more information.

In many of the images of this chapter, to have more space for the model, we have removed the command window and the log window, by clicking the **Commands** and **Logs** buttons in the status bar at the bottom of the window. We then get:



4.1.1 Dymola Main window

What is available in the Dymola Main window by default depends partly on if you build a model or simulate it. When building or editing a model, you can use any of the following tabs – they will be more described in detail later:

- The **Graphics** tab. That tab is shown in the figure above, and is used for working with the graphical representation of the model.

- The **Documentation** tab, for working with documenting the model.
- The **Text** tab, for working with the code of the model, using the Modelica language.
- The **Tools** tab, for general tools and settings.

This chapter deals with those tabs. There is one more tab, the **Simulation** tab, used when simulating the model. That tab is described in the next chapter.

Default content in Dymola Main window when developing a model

Dymola Main window by default contains the following when using any of the tabs above:

- A number of commands for each tab (in the tab ribbon). All these commands are presented in the section "Editor command reference when developing a model" starting on page 446.
- Model tabs, making it simple to switch between models.
- Three sub-windows – package browser, component browser, edit window, command window and log window.

Additional content available

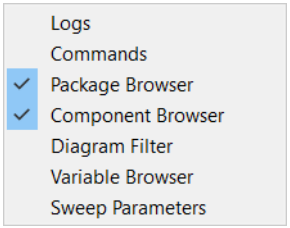
Additional content available when modeling depends on:

- User selection of content; by clicking the buttons in the lower part to of the status bar at the bottom of the Dymola window, the content of Dymola window can be changed. Features by default only available in the Simulation tab can be added. The selection is individual for each ribbon tab and is saved between sessions. For more information, see section "Window content – Dymola Main window" on page 444.
- Errors detected (e.g. syntax errors when programming). If errors are detected a log window will appear.
- When using any **Info** button or **Info** command an information browser pops.
- A diagram filter browser can be displayed by the command **Diagram Filter** from the **Graphics** tab.

Context menu

Right-clicking in an empty space in any of the riboon tabs of the Dymola main window will display a menu where the available sub-windows can be changed:

Context menu of main window when modeling.



This menu is the same as the buttons in the lower part to of the status bar at the bottom of the Dymola window, see previous section. You can also right-click in an empty space in the gray header of any sub-window to get this menu.

4.1.2 Edit window

The edit window is a sub-window of Dymola Main window. The edit window is used to build/edit models, but also for model inspection. Models are built using existing *classes* (e.g. models and connectors), connections between connectors and graphical information.

The edit window has a toolbar with buttons to select layer and to start drawing operations. The edit window is closely related to the package browser and component browser along its left side. The user may e.g. insert components into a model by dragging components from the package browser.

Class layers

An edit window by default shows a single layer of a class; the diagram layer. Four more layers are available and can be open at the same time. Each layer represents a different aspect of the class. The first two layers are graphical, the rest text representation. The five layers are:

- Icon layer
- Diagram layer
- Documentation layer
- Modelica Text layer
- Used Classes layer

What layer that should be displayed as top layer is usually selected using command buttons to the lower right in the main window - the order of the buttons corresponds to the list above.

Selection of class layer by buttons.



Selection of class layer by shortcuts.

There are shortcuts available as well (the last shortcut is really not a shortcut to a class layer, but here for completeness):

Shortcut keys	Action
Ctrl+F1	Displays icon layer
Ctrl+F2	Displays diagram layer
Ctrl+F3	Displays the documentation layer
Ctrl+F4	Displays the Modelica text layer
Ctrl+F5	Displays the Used Classes layer
Ctrl+F6	Displays the "simulation layer" (the **Simulation** tab) This tab is described in the next chapter.

Selection of class layer by tabs.

The class layers are associated to the tabs; the icon layer and diagram layer are presented in the **Graphics** tab, the Modelica text and Used Classes layer are presented in the **Text** tab, and the documentation layer has a dedicated tab, the **Documentation** tab. So by selecting a certain tab, you get access to certain class layers.

There are also shortcuts to the tabs (for completeness the shortcuts to the tabs available when simulating the model are listed as well):

Shortcut keys	Action
Alt+F	Displays the **File** menu
Alt+G	Displays the **Graphics** tab
Alt+D	Displays the **Documentation** tab
Alt+T	Displays the **Text** tab
Alt+S	Displays the **Simulation** tab
Alt+O	Displays the **Tools** tab
Alt+P	Displays the **Plot** tab
Alt+A	Displays the **Animation** tab
Alt+C	Displays the **Script Editor** tab

For information about the tabs not corresponding to class layers, see the next chapter. **Note** that you can only go to tabs visible, that is, you can, for example, never be in the **Graphics** tab and go to the **Plot** tab by using **Alt+P**; the **Plot** tab is not visible when in the **Graphics** tab.

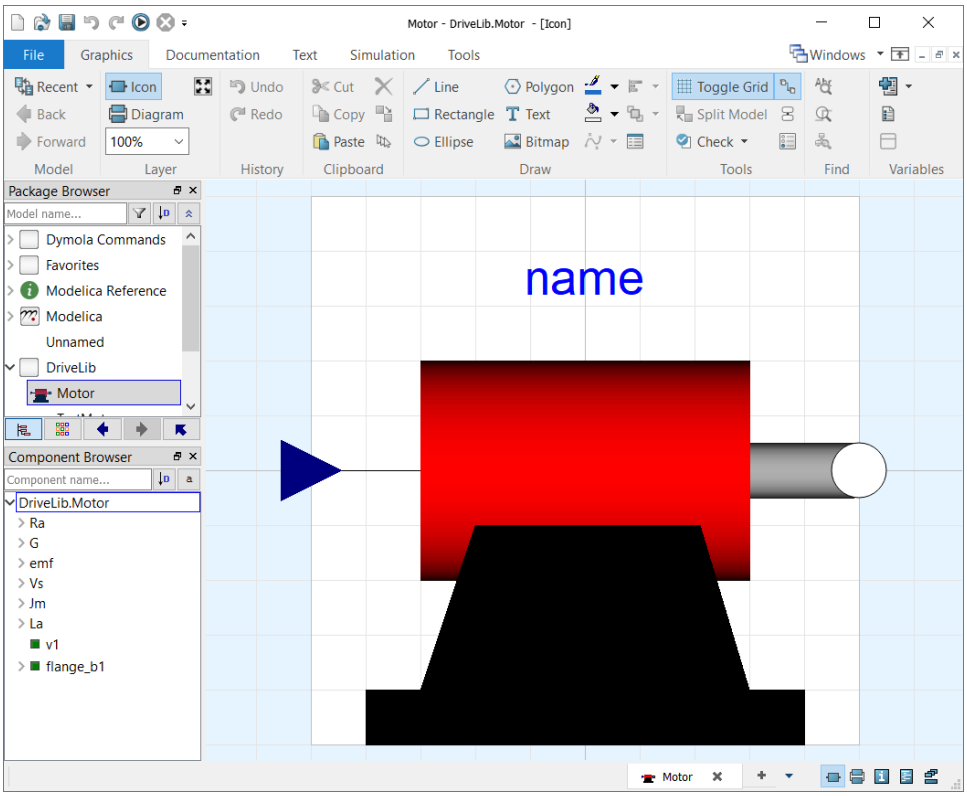
Indication of class layer selected.

What class layer *is* displayed as top layer is indicated in two ways: by the button line (in this case the diagram layer is on top) and by a text in brackets in the end of the Dymola main window header.

Icon layer

The icon layer of the edit window.



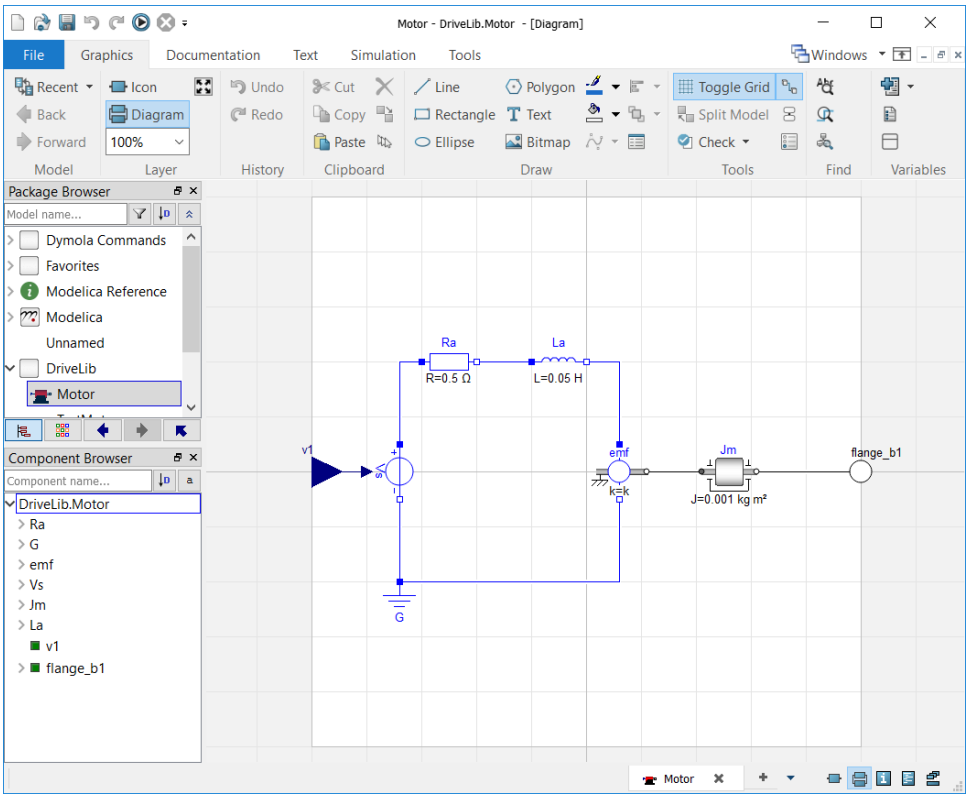


The icon layer represents the class when it is used as a component or connector in another model, that is, the diagram layer displays the icon layer of its components. The surrounding background of the icon layer is light blue when editing is possible. The features available for this layer are similar to the ones available for the diagram layer presented below.

Diagram layer

The diagram layer of the edit window.





The diagram layer shows the major contents of the class, i.e., components, connectors and connections, decorated with additional graphical primitives. The surrounding background of the diagram layer is light grey when editing is possible.

The diagram layer is used when building the model by inserting and manipulating components represented by graphics. Connectors are typically placed in the diagram layer of a model. Public connectors are also shown in the icon layer to assist the graphical layout of the icon and to make it possible to connect to the component. Protected connectors are only shown in the diagram layer.

The name of a component can be shown as a tooltip by resting the cursor on it.

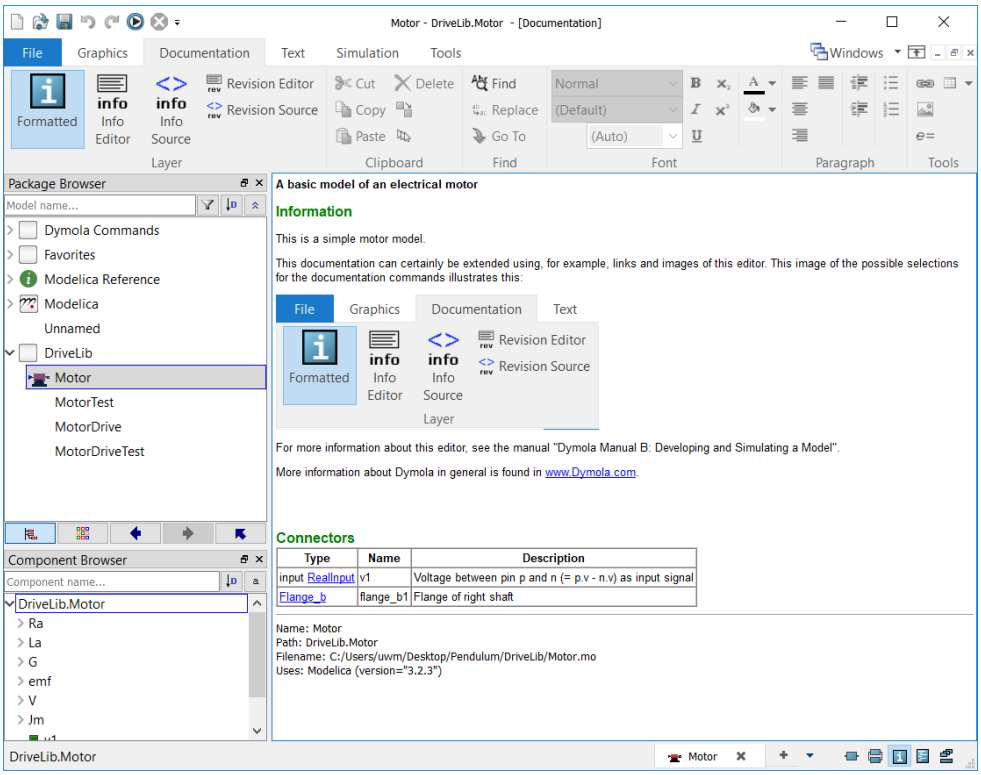
Moving the view and zooming is possible. Please see the section “Moving” and the section “Zooming” on page 217.

The diagram layer is a central layer when it comes to editing, it will follow us in most of this chapter.

Documentation layer

Documentation layer of edit window.





The documentation layer shows the one-line description of the class, plus the longer info text that gives a complete explanation of its behavior. Information about parameters, connectors, inputs/outputs, package content and revision is displayed as well. These texts are available when browsing classes and components. It is also used for automatically generated HTML documentation.

A documentation editor makes it possible to include pictures, links, headers etc. in a convenient way.

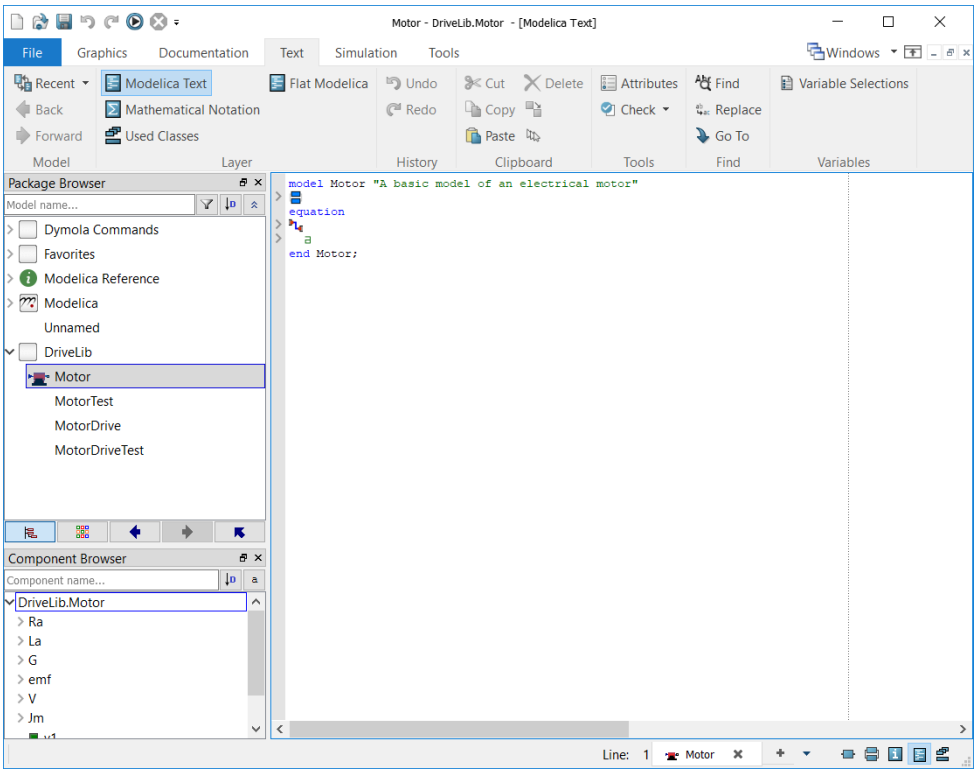
Metadata and markdown are supported as well.

More about documentation is presented in the section “Documentation” on page 300.

Modelica Text layer

Modelica Text layer of the edit window.

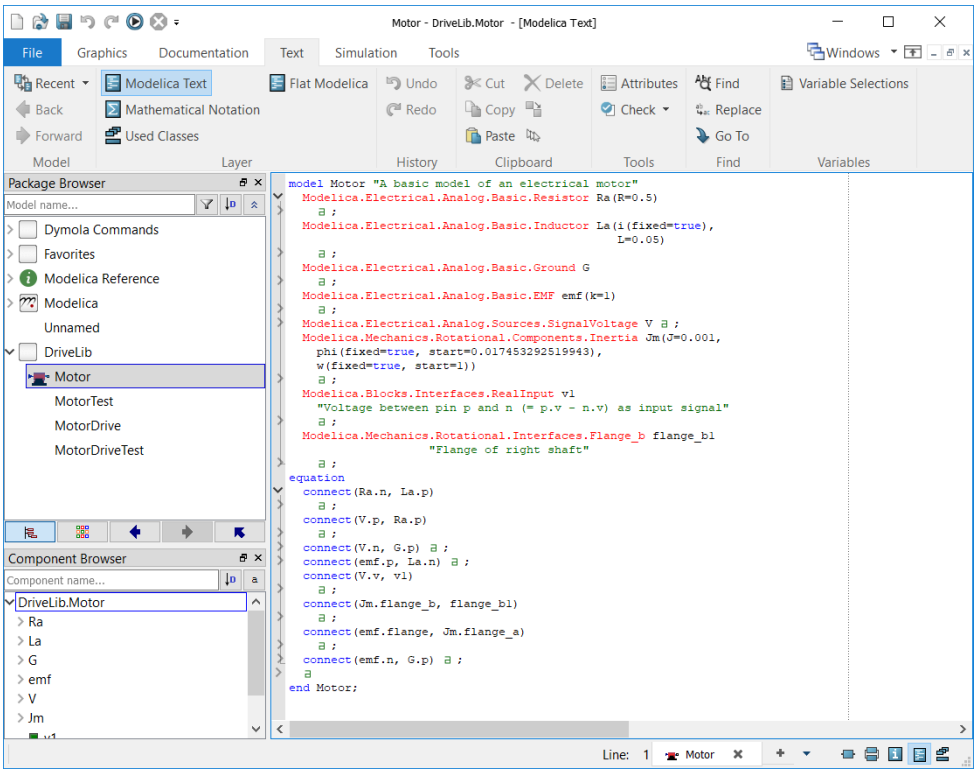




The Modelica text layer shows simple declarations of local constants, parameters and variables, and the equations of the class. Components with graphical representation are by default hidden.

In this specific example it is a good idea to expand what is shown clicking on the expand icons ▷ or by right-clicking and selecting **Expand > Show Components and Connections**. That will show:

Modelica Text layer of the edit window. – expanded.



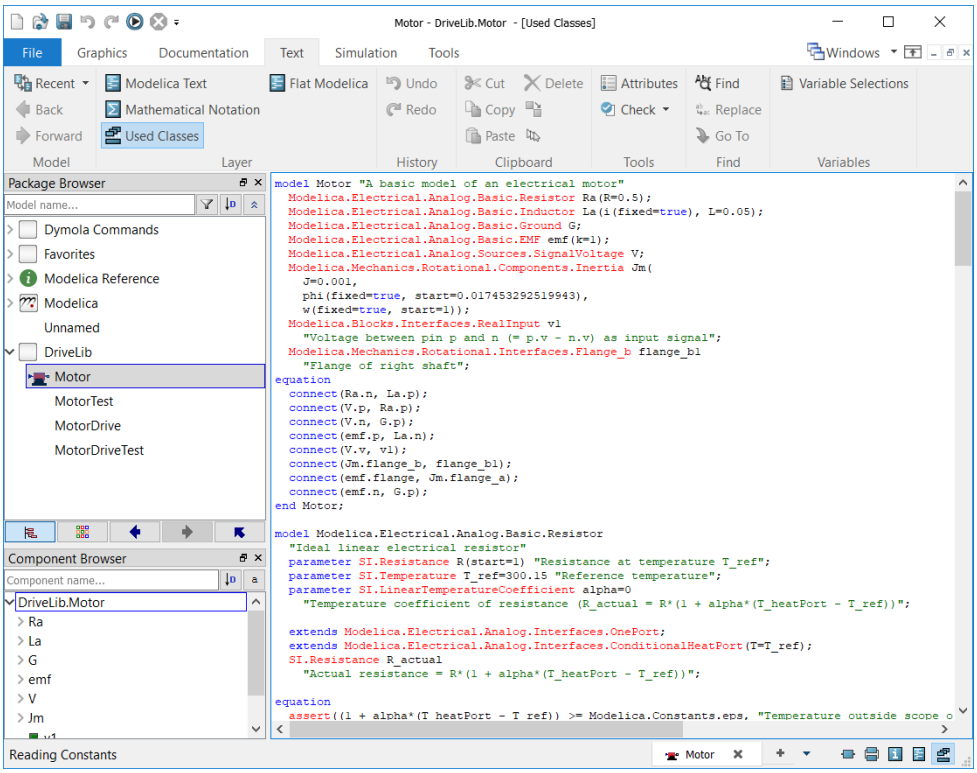
The Modelica Text layer can be used to edit equations etc.

More information about how to use this layer for programming Modelica text is given in section “Programming in Modelica” on page 271.

Used Classes layer

Used Classes layer of edit window.





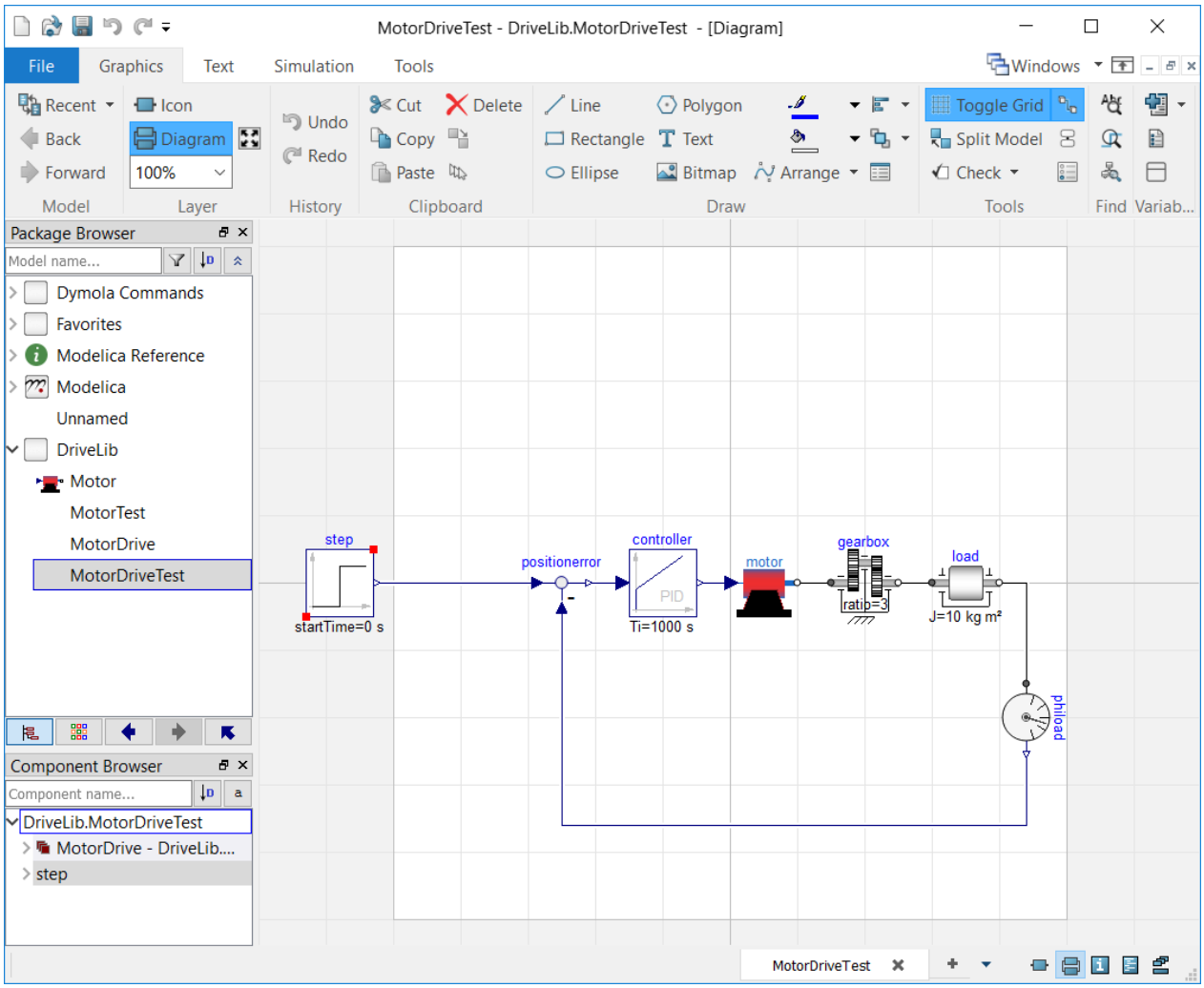
The used classes' layer shows the base classes, classes used for components and connectors, and the Modelica text of the class. This facilitates searching for the location of e.g. a specific variable.

By selecting, on the **Text** tab, in the **Layer** group, **Flat Modelica**, the flat Modelica text is showed instead. There are some limitations what can be displayed in this mode, see reference below. The flat Modelica text is the result when component structure and inheritance has been evaluated; it corresponds to the result when translating the item. The result is a more readable representation; the feature is valuable when investigating more complex models.

Note that the text in the Used Classes layer cannot be edited.

For more information about the Used Classes layer, e.g. about the flat Modelica text, please see section "Inspecting code in the Used Classes layer" starting on page 297.

Coordinate system



Every class has a coordinate system that is used for all graphical information in the class. The view of a class in a window is normally scaled so the entire coordinate system is visible. The extension of the coordinate system is marked by a boundary. The origin of the coordinate system is also marked. The user can change the zoom factor to show more or less of the model (see “Graphics > Zooming” on page 469). The coordinate system is defined with real numbers and does not depend on screen or window size.

A class may also have a grid specification, which is used to make drawing operations easier. Points defined by the user will “jump” to the nearest grid point. Points read from a file are not snapped to the current grid.

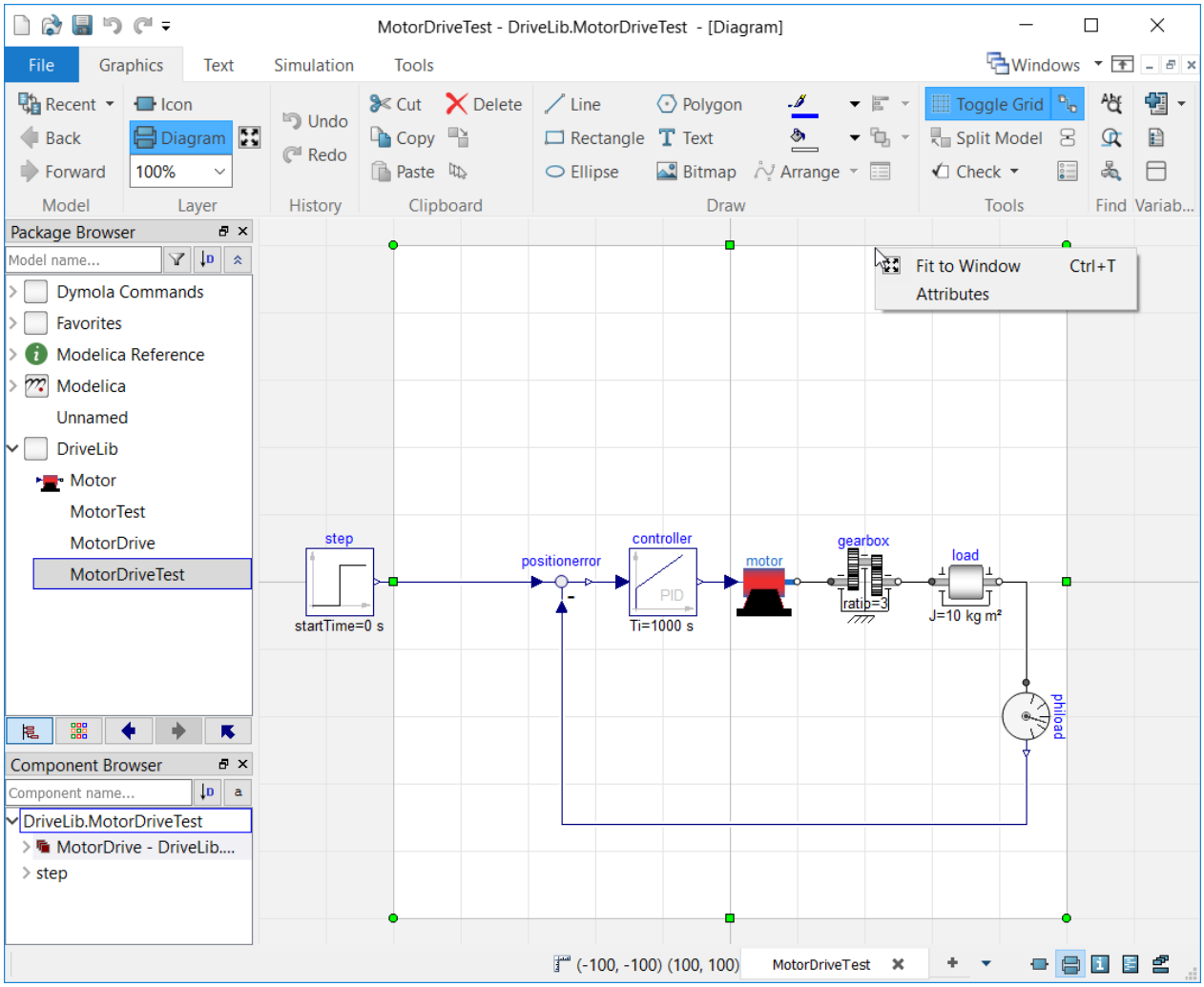
Associated with every model class is also a default component size. Components and connectors dragged into the model class will initially have the default size, but can be reshaped after insertion.

Specification

The coordinate system can be defined by the user either by dragging of the borders of the boundary, or in the **Graphics** tab of the **Graphics > Attributes** menu (see “Graphics tab” on page 480). The latter can also be reached by right-clicking when the coordinate system boundary is selected.

To facilitate selecting the boundary, right-clicking in the diagram or icon layer (having nothing previously selected) will display a context menu where **Select Boundary** will select the boundary.

The selected boundary is highlighted with *green* handles (to be able to differ it from e. g. a graphical object on top of it in the icon layer), and the extent is shown in the status bar. The following example shows selecting the boundary by right-clicking on it:



A typical user case to enlarge the coordinate system is to zoom out pressing **Ctrl** and using the mouse scroll button, then selecting the boundary and extending it by dragging the handles, and finally set the zoom level to 100 % by selecting 100 % in the zoom level box in the upper right.

The default coordinate system, grid and component size are either

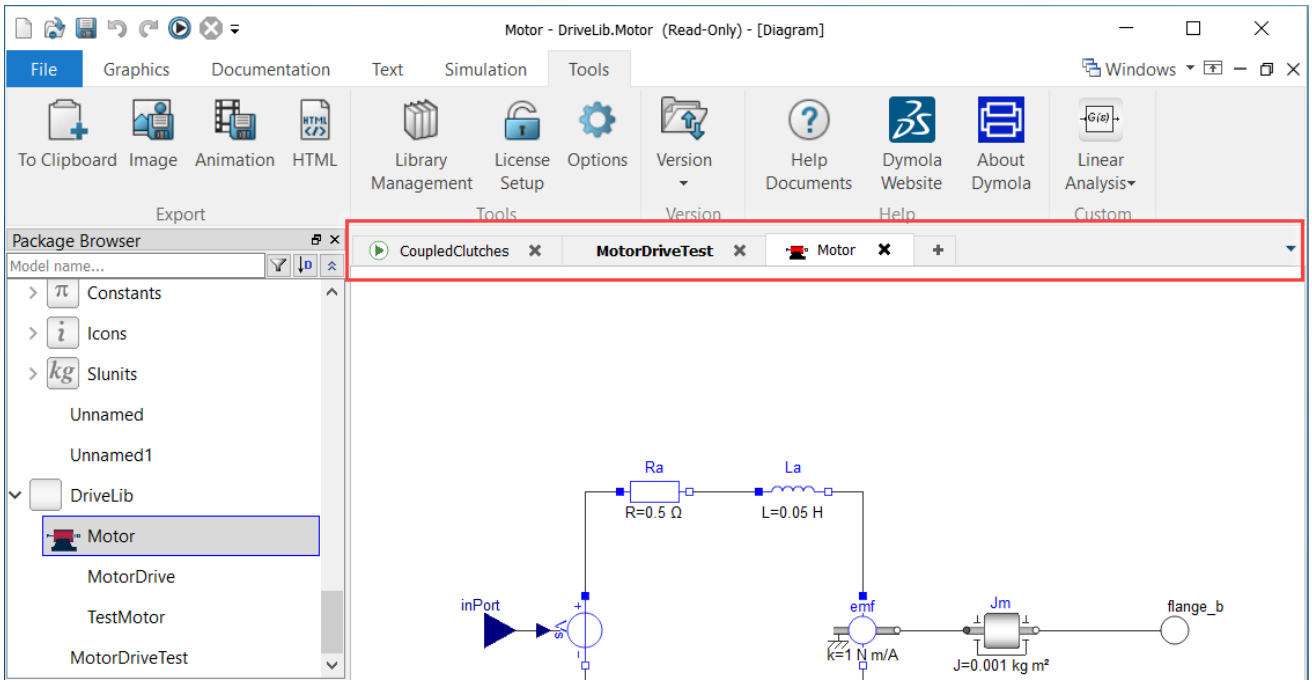
- Inherited from a base class, or
- Copied from the program defaults (–100, –100) to (100, 100) with grid (2, 2). The default component size is (20, 20).

Alignment of components in a diagram is facilitated by gridlines. Gridlines are drawn for every 10th of the class' grid points.

The command **Fit to Window** adapts the zoom factor to display the whole model, independently if e.g. some parts of the model are located outside the coordinate system. The command can be reached as for example a button beside the zoom factor box or from the context menu when nothing is selected.

4.1.3 Model tabs

More than one model can be open in the main window in Dymola.



Opening of model tabs

When Dymola is opened, one model tab is opened as well, plus a "+" tab. Clicking that tab adds a new tab with a model `Unnamed`. If such a model is already present, the name will be `Unnamed1` etc. This "+" (New tab) tab is always available.

When creating a new class (except a new package) using the command **File > New** (or corresponding button), the new class is opened in a new model tab by default. (This can however be changed, see section "File > New > Model etc." on page 448.)

A number of context menus that can open new classes have a command for opening a class in a new model tab, for example the context menu for a class in the package browser or component browser, the context menu for Modelica Text layer and the context menu for a component in the diagram layer.

The model tabs can be rearranged by dragging them.

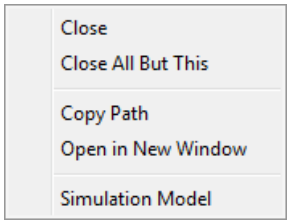
If a model is selected as active simulation model, the model name in the tab is in boldface. For information about active simulation model, see section “Selection of active simulation model” starting on page 325.

Closing of model tabs

Clicking the red cross of a model tab closes it. If a model that is displayed in a tab is unloaded, the tab is also closed. It the tab of the active simulation model is closed; no model is selected as active simulation model.

Context menu

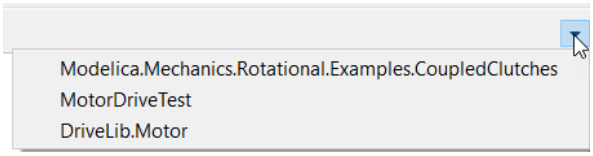
Each model tab has a context menu:



For more information about the context menu, see section “Context menu: Model tabs” on page 542.

List of model tabs

A list of tabs is available to the right in the tabs toolbar:



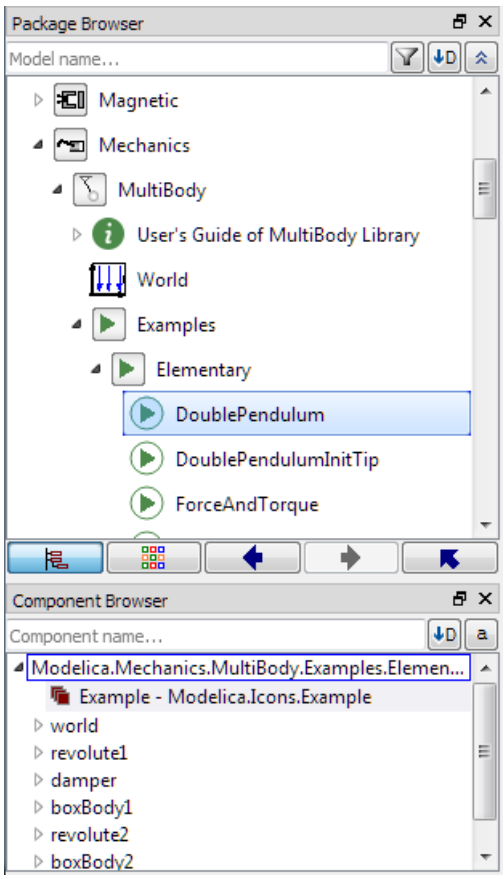
Navigating model tabs using shortcuts

Having a number of model tabs, **Ctrl+Tab** activates the tab to the right of the currently active model tab, corresponding to moving one step down in the model tab list. **Ctrl+Shift+Tab** activates the tab to the left of the currently active model tab, corresponding to moving one step up in the model tab list.

4.1.4 Package and component browsers

General

Package and component browser.



The Dymola main window contains by default two browsers along the left edge of the window. The package browser (top) displays the hierarchy of several packages and it is possible to drag a component model from the tree into the diagram layer of the edit window (the graphical editor) in order to add a component to a model. The component browser (bottom) provides a tree representation of the current model's component structure.

Browser handling



The package browser and component browser can be undocked using the **Undock** button or by double-clicking on the browser header or dragging it.

To dock the browser again, the best way is to double-click on the browser header. It can also be dragged to its original location (you will then see how space for it is created automatically before releasing it).

It is possible to dock also to the right side of the main window.

It is also possible to have the package browser and component browser as two tabs by e.g. undocking the package browser and dragging it on top of the component browser.

If you have displayed a diagram filter browser, it is treated like the other browsers.

Handling of components

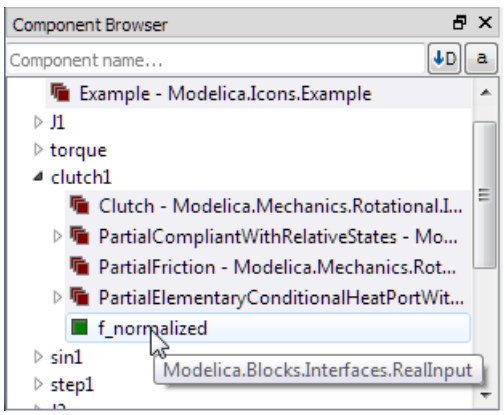
Please note that selecting a package in the package browser by just clicking on it does not mean that it is the one that is displayed in the edit window (and component browser). Yes, it is indicated in the package browser by blue (or red if not saved), but the one displayed stays the same (and is indicated by a blue frame in the package browser, the name in the window header, the top name in the component browser and the name in the bottom left in the window). By *double-clicking* on the package in the package browser the displayed package is changed, however. (An alternative is to right-click on the package in the package browser and select any **Open Class…** command.) Note that this will not be the case if the active tab contains the active simulation model; in such a case the selected class will be opened in a new tab.

The edit window and the component browser are synchronized to give a consistent view. When you select a component in the edit window (displaying the diagram layer), it is also highlighted in the component browser and vice versa. The diagram layer of the edit window shows the top-level component structure of a component, while the component browser shows the entire hierarchical component structure.

When a model is chosen in the package browser, it becomes the root model of the graphical editor. The check, translate and simulate commands operate on the root model. Navigation into its component hierarchy allows inspection of model details, but does not change the root model or permit editing.

Tooltip information.

The type of a component etc. is shown as a tooltip if resting the cursor over it.



The contents of the package browser and the component browser are by default sorted according to order of declaration in the enclosing class. To change the order, you can click the sorting button to sort in three different ways:

-  Sort in declaration order

-  Sort alphabetically A - Z
-  Sort alphabetically Z - A

Note that the button displays the current sorting.



To collapse the tree in the package browser, you can click the button **Collapse all packages**.

For filtering, see “Filtering” starting on page 208.

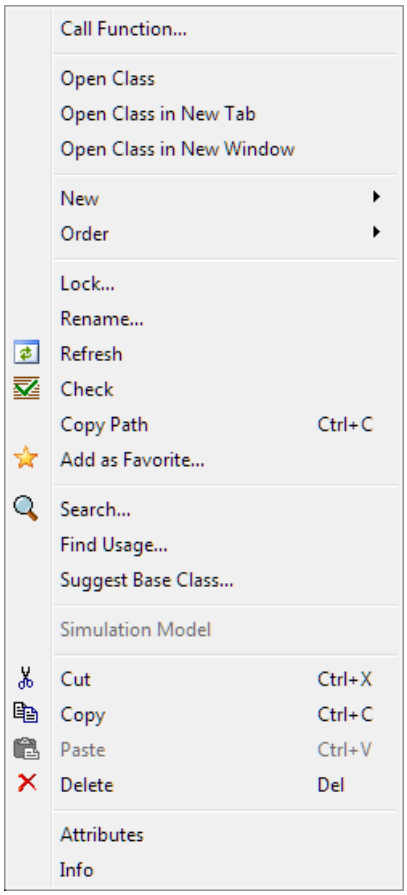
By dragging components from the package browser to the component browser certain actions can be obtained (changing of type, variable declaration etc.). Please see section “Inserting a component or a connector” on page 235.

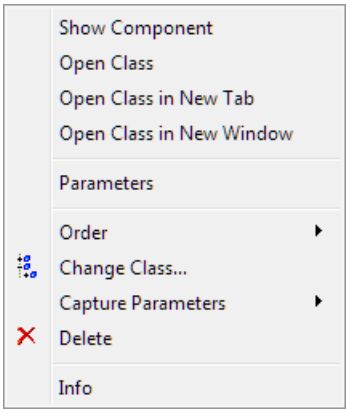
Context menus

Right-clicking **on a component** in the **package browser** presents a context menu (the figure to the left below). **Edit** will display a number of editing operations; in particular, to create/remove/modify classes of a package. For more information on the alternatives, please see section “Context menu for components in the package browser” on page 543.

Right-clicking **on a component** in the **component browser** presents a context menu with the choices in the figure to the right in the figure below. For more information on the alternatives, please see section “Context menu for components in the diagram layer and the component browser” on page 551.

Examples of context menu from a component in package and component browser.





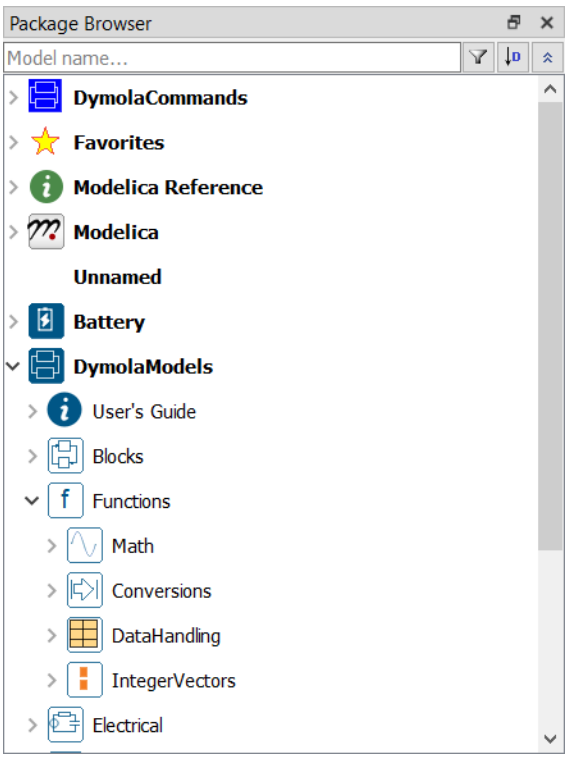
Presentation in package browser

What is not shown

Protected classes are by default not shown in package browser.

Highlighting top-level classes

To facilitate easier navigation in the package browser, in particular when you have opened several sub-packages, you can select to have all top-level classes highlighted:



You activate this option by setting the flag

```
Advanced.UI.HighlightPackageInBrowser = true
```

The flag is by default `false`.

Components in the package browser

A number of symbols can be seen in the package browser when any package is open. This is not strange because the user can change the symbol (icon) of all entities except constants (and that does not mean that all texts without a symbol are constants!). However, some symbols are often kept. The table below gives a few examples of such symbols:

Examples of some symbols.

Symbol	Meaning	Comments
	Package	Packages often have different symbols
	Information	This symbolizes a package containing only information. Selecting such a package will always show the documentation layer and dragging of such package is inhibited
	Function	
	Record	
	Constant (pi)	Constants are symbolized by their names, without any other symbol. Resting the cursor over it the value will be shown as a tooltip. (Please note that also other things might be symbolized with a text without a symbol)

To see what the symbol symbolizes, the easiest way is to look in the corresponding documentation layer of the edit window.

Please note that in some case the same symbol can be used for different categories of components, e.g. the “information” symbol.

Specific indication on the symbols

A selected component in the package browser has blue background if saved, red background if modified but not saved. Please note that a component modified but not saved is not indicated by red if not selected!

The component in the package browser that is displayed in the edit window and component browser is indicated by a blue frame if it is a component not selected (otherwise it is the one selected).

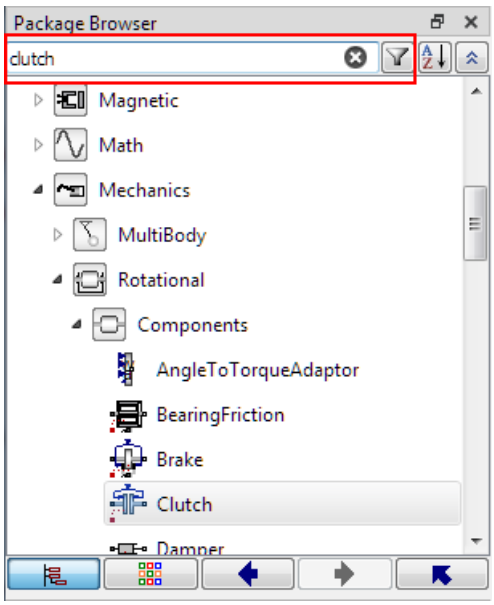
Please note that selecting a package in the package browser by just clicking on it does not mean that it is the one that is displayed in the edit window (and component browser). Yes, it is indicated in the package browser by blue (or red if not saved), but the one displayed stays the same (and is indicated by a blue frame in the package browser, the name in the window header, the top name in the component browser and the name in the bottom left in the window). By *double-clicking* or *right-clicking* on the package in the package browser the displayed package is changed, however.

Filtering

Filtering in the package browser

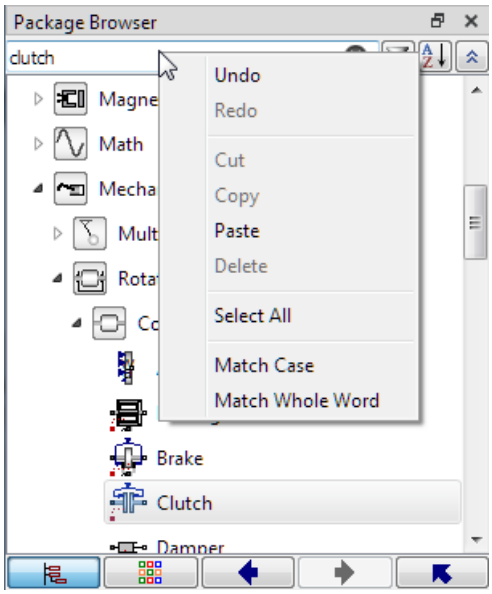
It is possible to search and filter the content of the package browser. Also, filtering may search and load unloaded libraries.

To search, start typing in the filter input box, like the example below where clutch has been typed:



The matches are dynamically updated while typing, however with a short delay to prevent immediate search while still typing. The filter marks the first matching class. **Note** that exact match is picked first. To go to next match, press arrow down when in the textbox; to go to previous match, press arrow up when in the textbox. To open the class displayed in the textbox, press **Ctrl+Enter** when in the textbox.

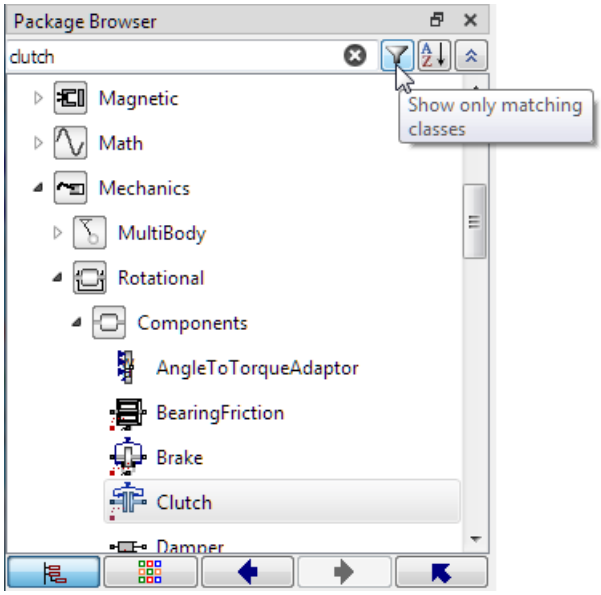
The matching is by default case-sensitive. This can be changed by the second last command **Match Case** in the context menu of the filtering line. The last command **Match Whole Word** makes it possible to match only whole words (if selected, parts of words will not match). By default these commands are not activated:



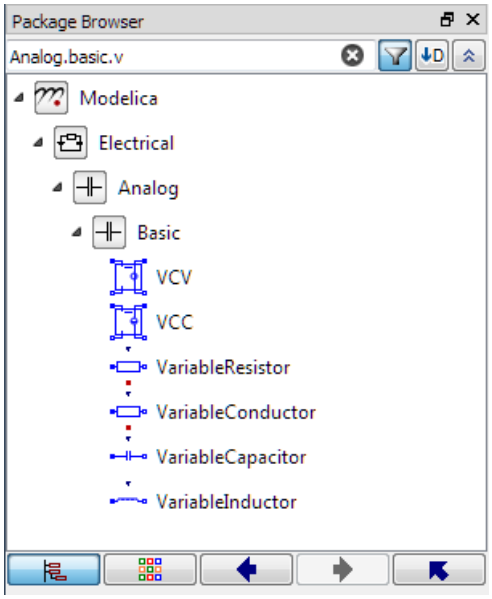
Packages extends nodes (whose name start with `extends`) are excluded from the filter.



Classes not matching can either be shown or hidden. The default is to show them. To change this, click the **Show only matching classes** button. The button works in toggle mode. Clicking this button in our example gives:



The filter searches for matches in the whole Modelica path of the model. This means that an expression such as `Analog.Basic.V` and be used:

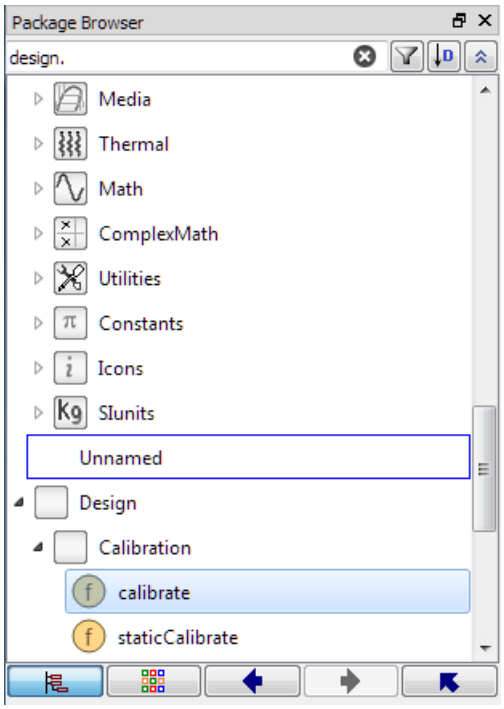


Note that when a string is detected as a Modelica path, the corresponding class is navigated to, otherwise filtering is performed.

The filter can search and, if matches are found, load unloaded libraries, if the following conditions are fulfilled:

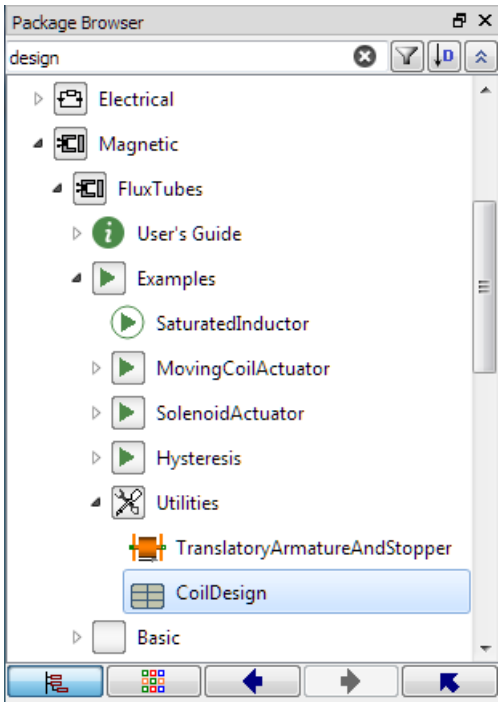
- The filter text must be a valid Modelica path
- The filter text must contain at least one dot “.”
- The option **Filter can automatically load libraries** must be active. This option can be reached by the command **Tools > Options...**, the **Package Browser** tab. See “Package Browser options tab” on page 512.

An example of filtering loading an unloaded library could be:



Note. *Loading* libraries does not change the working directory, *opening* libraries does. (Compare the command **File > Open > Load…** and **File > Open > Open…**.)

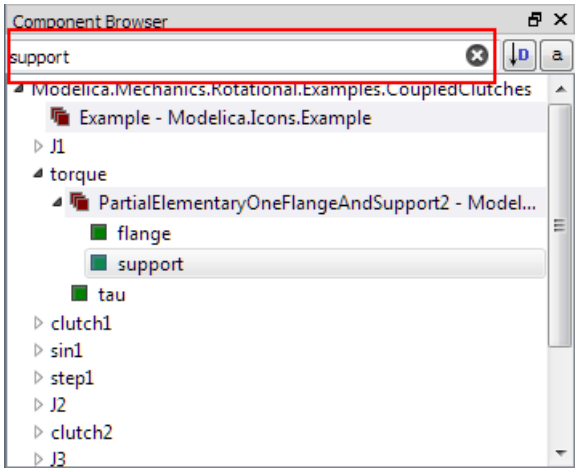
If any of the above conditions are not fulfilled, the filter will only search in the libraries that are present in the package browser, for example, not typing the dot after `design` will display (given the Design library is not previously loaded for some other reason):



Filtering in the component browser

It is possible to search and filter the content of the component browser.

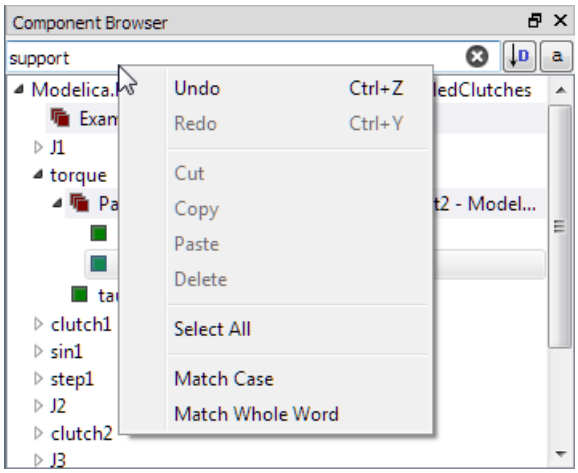
To filter, start typing in the filter input box, like the example below where `support` has been typed:



The filtering is dynamically updated while typing, however with a short delay to prevent immediate search while still typing. The filter marks the first matching component. To go to

next match, press arrow down when in the textbox; to go to the previous match, press arrow up when in the textbox. To apply the command **Show Component** for the component selected, press **Ctrl+Enter** when in the textbox.

The filter is by default not case-sensitive. This can be changed by the second last command **Match Case** in the context menu of the filtering line. The last command **Match Whole Word** makes it possible to match only whole words (if selected, parts of words will not match). By default, these commands are not activated:



Notes:

- The filtering never matches on the model name (the top-level node)
- The filtering never matches extends classes.
- Non-matching components are always shown.

Including top-level non-graphical components in the component browser



To include top-level components containing no graphics in the component browser, click the button **Include non-graphical** after the filtering and sorting buttons. The button works in toggle mode, and corresponds to the setting with the same name, reached by the command **Tools > Options** the **General** tab. See “General options tab” on page 511.

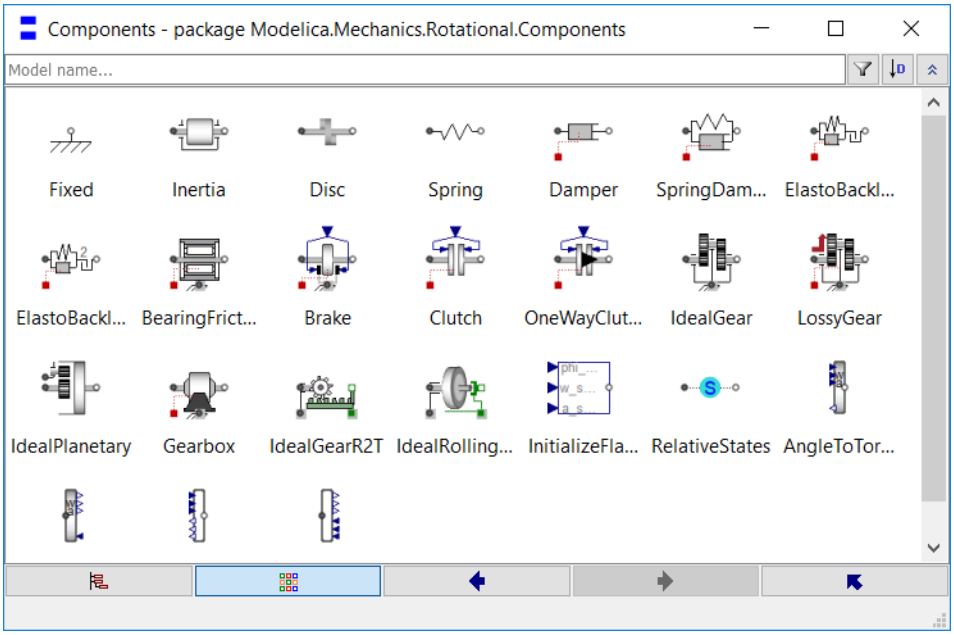
4.1.5 Library window

A library window shows the contents of a Modelica package, typically models, connectors or other packages. The user may insert components into a model by dragging components from library windows to an edit window. Double-clicking on a model or a nested package opens it.

The library window can show a tree view of the package, an icon view of the package or both. For more information on Library window, please see the chapter “Getting Started with Dymola”. Library windows do not allow editing and there is no toolbar.

The library window is not displayed by default; the command **Windows > New Library Window** (or **Open Library Window** in the package browser context menu) has to be used to display such a window.

A library window with icon view only.



4.1.6 Command window

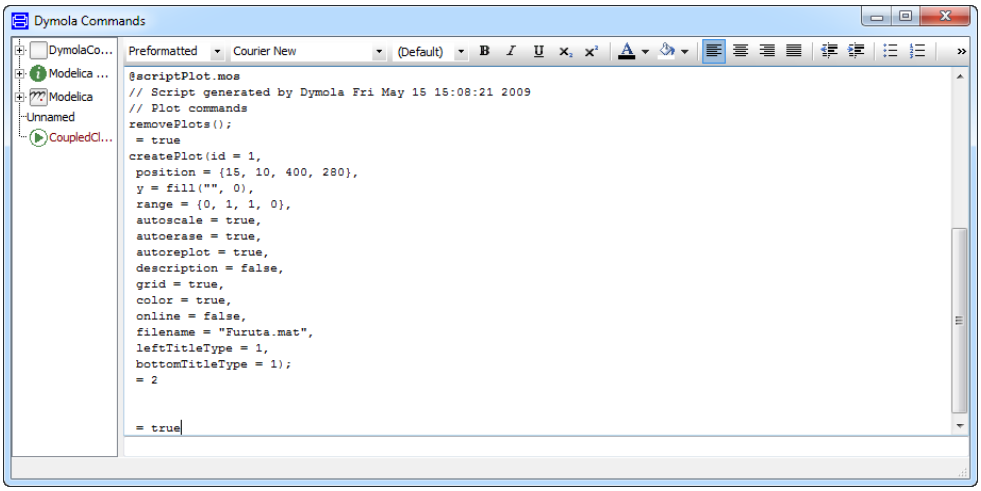
The Command window is used for entering commands to Dymola.

It is shown by default when you have selected any tab that is used when simulating a model, but can be made visible in for every tab selected in two ways:

- As a sub-window of Dymola Main window (this is default in simulation tabs) by right-clicking in the header part of the window and selecting **Commands**.
- As a free-standing window by using the command **Windows > New Command Window**.

For a look on how the command window looks like as a sub-window of Dymola Main window, please look in the next chapter. As a free-standing window it looks the following:

The Command window as a free-standing window.



The window contains a package browser, a toolbar, a pane showing the Command Log and a command input line. For more information about the window, including the context menus, please see next chapter.

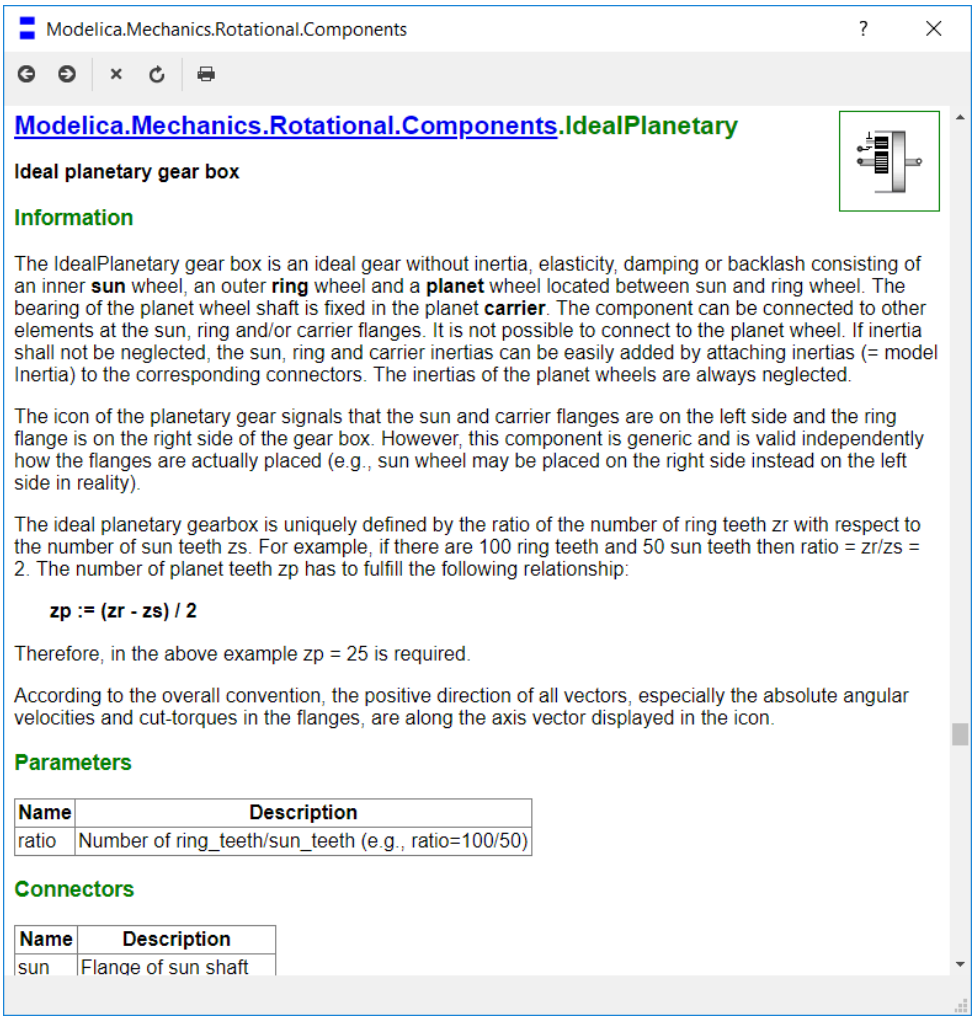
4.1.7 Log window

The log window is by default shown when you have selected any tab that is used when simulating a model. It appears also in other tabs when an error (e.g. a syntax error) is detected. Since the window is much more used when simulating, please next chapter for more information.

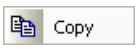
4.1.8 Information browser

An information browser is used to display model documentation, e.g. when using any **Info** button or **Info** command in any context menu.

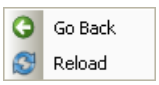
Information browser.



The toolbar contains the usual **Go back**, **Go forward**, **Stop**, **Reload, Copy** and **Print** facilities.

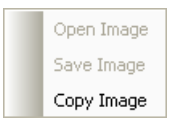


Right-clicking on a selected text will pop a menu that enables copying of that text. **Ctrl+C** can of course also be used, as can the corresponding icon in the toolbar.



Right-clicking on an empty area will give a context menu that allows going back or reloading present view.

These alternatives are also present in the toolbar.



Right-clicking on an image will give a context menu that allows copying the image to the clipboard.

For details on what is shown in the browser, see section “Looking at the documentation layer (and more) using the Info button/menu entry” starting at page 306.

4.2 Basic model editing

This section describes basic editing of models. For more advanced features, please see the section "Advanced model editing" starting on page 329.

4.2.1 Basic operations

The default interaction mode, known as "select mode" or "pick mode" is used both to select and move objects, and also to connect connectors. Other modes are temporary; for example, after inserting a component the program goes back to select mode.

Moving and zooming in the diagram/icon layer

Moving

To move the view, press and hold the **Ctrl** key as well as your left mouse button and then move your mouse. The diagram will be dragged along with your mouse cursor. To stop, simply let go of the left mouse button.

As an alternative, you can use the mouse wheel to move the diagram/icon layer in the y and x direction the following way:

- **Mouse wheel** moves the diagram/icon layer in the y direction only
- **Shift+Mouse wheel** moves the diagram/icon layer in the x direction only

Zooming

To zoom, there are four different options:

- Press and hold the **Alt** key and span a rectangle[8]. When the mouse button is released, the spanned area is zoomed to fit the window.
- Mouse wheel: press and hold the **Ctrl** key and scroll the mouse wheel.
- Mouse move: press and hold the **Ctrl** key and the right mouse button, then move the mouse forwards or backwards.
- Change the zoom factor by editing the zoom factor value in the toolbar.



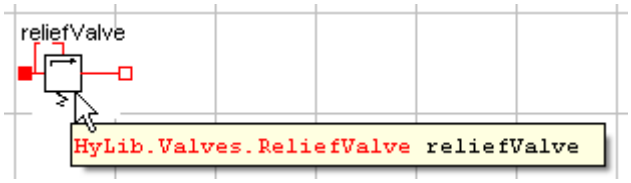
A convenient way displaying the whole model independently if some objects are located outside the coordinate system is to right-click an empty area, having nothing selected, and select **Fit to Window** from the context menu. This command is also available as a button in the upper toolbar. (If components in the model are located outside the coordinate system, the resulting zoom factor will differ from 100 %.)

[8] On Linux, **Ctrl+Shift** have to be used instead of **Alt**.

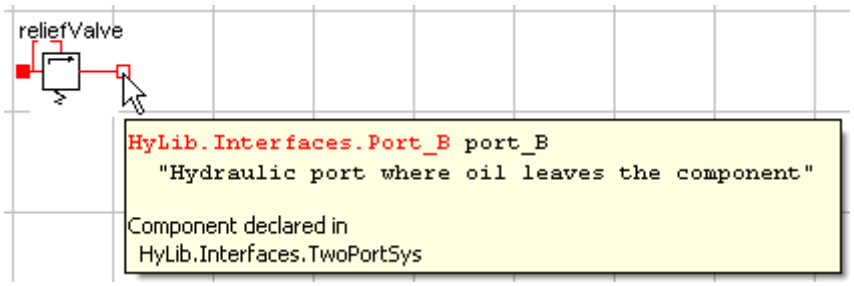
Dynamic tooltips

Resting the cursor over a model component or connector displays a tooltip with type and name. Over a connection line the tooltip contains the names of the connectors.

Component tooltip.



Connector tooltip.



Selecting objects

Left mouse button selects, moves and reshapes.

Visible objects, i.e., components, connectors and graphical primitives, are selected with the left mouse button. Many commands operate on the current selection, one or more objects marked with small red squares at the corners (called handles). The component browser also indicates selected components.

The procedure for selecting objects in the diagram or icon layers is as follows:

- Clicking on an unselected object makes this object selected, and the previous selection is unselected. Please note that graphical primitives must be selected clicking on the contour of the object, the inside is insensitive!
- Clicking on an unselected object while holding down the **Shift** key toggles the select status of this object, without unselecting the previous selection. Please note that graphical primitives must be selected clicking on the contour of the object, the inside is insensitive!
- Multiple objects can be selected by pressing the left mouse button with the cursor between objects, and then moving the mouse (while holding the left button down) to span a selection rectangle. All objects inside the selection rectangle are selected, and the **Shift** key has the same effect as described above.
- Clicking on a selected object does not change the current selection (if **Shift** is not pressed).
- Components can also be selected by clicking in the component browser.

In an edit window, double-clicking on a component by default opens a dialog, the parameter dialog with additional information. For components and connectors several attributes may be changed, see relevant sections in “Components and connectors” starting on page 235.

See also “Grapics > Select All” on page 470.

Context menus

Right mouse button presents a context menu.

Pressing the right mouse button (right-clicking) usually presents a context menu with operations suitable for the selected object. Context menus are presented for components and connectors, for lines and connections, for graphical objects, and also for the model itself when no object is selected.

Context menus are also available in the package and component browsers, and in the library window.

The content of all available context menus when developing a model are presented as sub-sections in the section "Editor command reference when developing a model" starting on page 446.

Moving objects

Objects are moved by pressing down the left mouse button with the cursor over one of the selected objects, and then moving the mouse (while holding the left button down) to the desired position. All selected objects are moved the same distance, rounded to a multiple of the grid.

Note that when connect mode is enabled connectors must be selected first, and then moved. Clicking on a connector followed by an immediate move will draw a connection.

Moving objects with arrow keys.

Selected graphical objects can be moved by pressing the arrow keys. The default is to move one grid unit (as specified in **Graphics > Attributes** – see "Graphics tab" on page 480). If the **Ctrl+Shift** keys are held down, the objects are moved half a grid unit. If the **Shift** key is held down, the objects move half a gridline (five grid units).

Note that you can space and align components using commands. See section "Spacing and aligninig components automatically" on page 246.

Connections are automatically adjusted after a move operation to make sure that the end points still reach the corresponding connectors. **Manhattanize** can be used to clean up skewed connections (see about Manhattanize in the section "Creating a connection without using Smart Connect" starting on page 252). Automatic routing is by default also applied after a move operation, see section "Routing of present connections" on page 253.

Reshaping objects

First select an object. The object can then be reshaped by moving the handles, for example, making a component larger or changing the shape of a polygon. The object is redrawn while the handle is moved.

For some objects, you can press **Shift** when moving a handle, which will lead to a uniform scaling.

Common reshaping operations are also available in the context menu of the object or on the **Graphics** tab, in the **Draw** group; for example, rotation and flipping horizontally and vertically. For more information, see "Graphics > Arrange" on page 474.

The commands **Insert point** and **Remove point** are available for inserting/removing points in certain graphical objects, please see section "Context menu: Graphical objects" on page 555.

Displaying subcomponents

To display the subcomponents in a component, select the component in the diagram, right-click it to display the context menu and select **Show Component**. (If you can select the component by just clicking, you can actually right-click it directly without first selecting it to get the context menu.)

You can also, by default, use **Shift+double-click** on the component in the diagram to perform the **Show Component** command. Alternatively, you can define double-click to perform this command. For details on this, see section "Graphical Editor options tab" starting on page 516.



Use the **Back** button in the **Model** group of the **Graphics** tab to get back. (You can also use the mouse button Back if you have such a mouse.)

As an alternative to working with the diagram layer, you can work with the component browser. Double-clicking a component in the component browser always perform the **Show Component** command. You can also right-click the component in the component browser and select **Show Component**.

Deleting objects

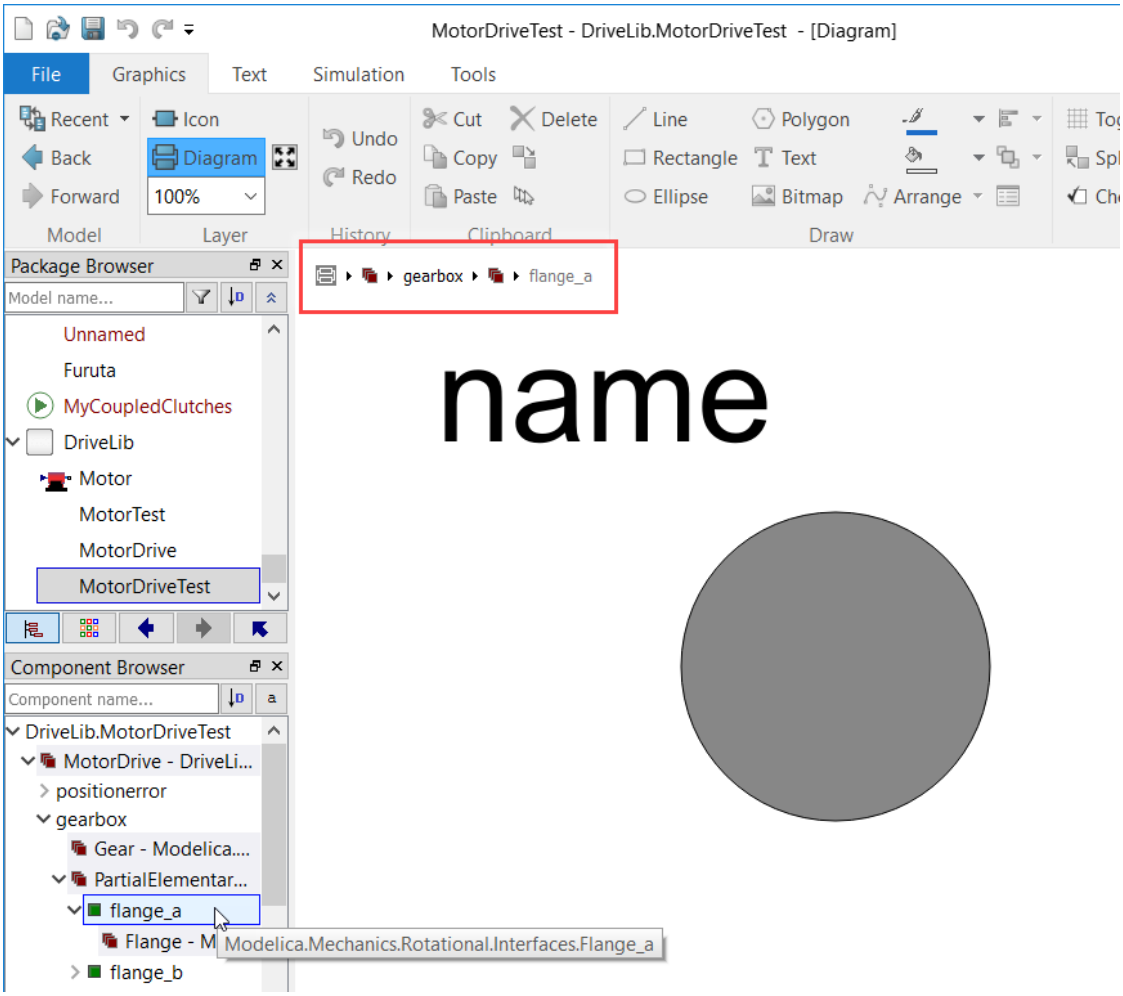
Pressing the **Delete** key deletes the current selection. Connections attached to any deleted components or connectors are also by default deleted, in order to maintain model consistency.

Only objects defined in the class being edited can be deleted. Inherited objects, while also shown as selected, are not deleted. Objects in read-only models cannot be deleted.

Navigation path information

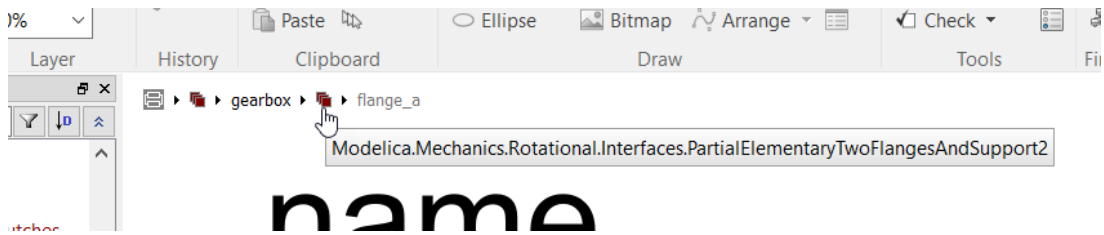
Except from information from dynamic tooltips (see above), information about location is given from the following:

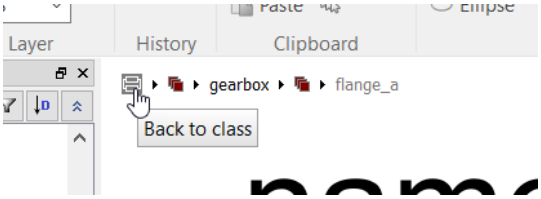
- The path to the opened class can be seen in the header of the main window. (This information is also displayed shortly in the lower left of the window when opening a new class.)
- The path to the opened component (instance hierarchy) can be seen in a toolbar above the package browser.



Note that base classes are shown in the instance hierarchy, with the same icon as in the component browser.

You can navigate by clicking in the instance hierarchy toolbar, for example:



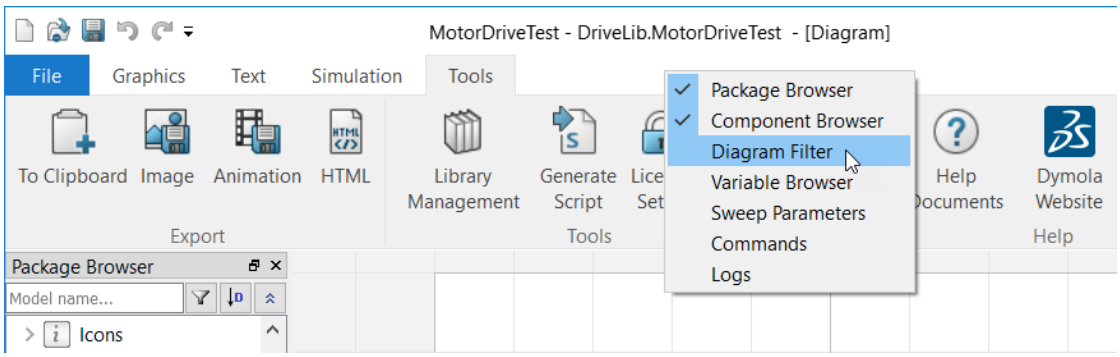


Filtering objects

To be able to get an overview of where components come from, you can filter the diagram layer to dim components and connections from selected packages and/or components not matching a name. You can also filter on inherited components and connections, if the model is editable.

The filtering can be activated in two ways:

- Right-clicking the header part of the main window and selecting **Diagram Filter**:



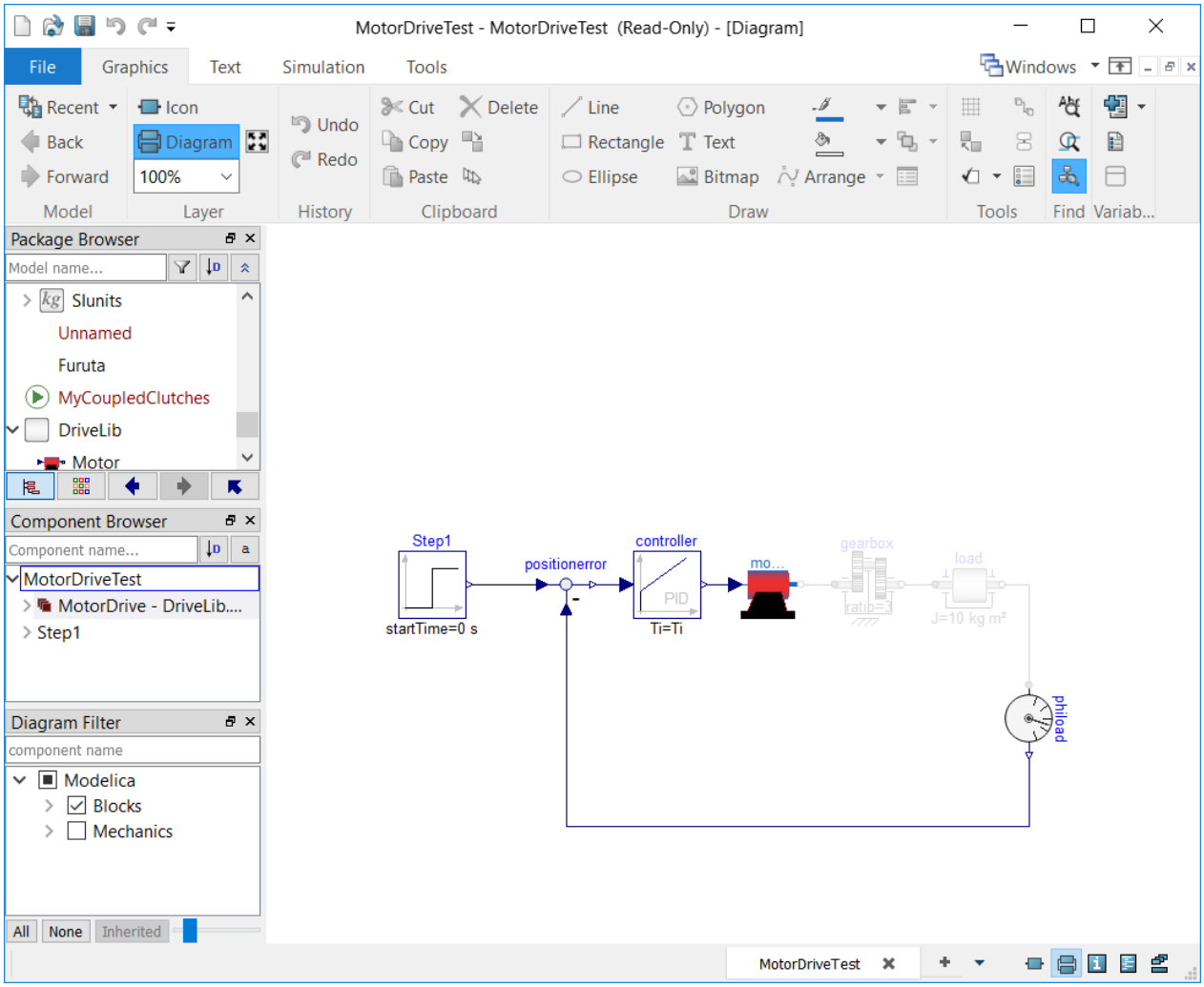
- On the **Graphics** tab, in the **Find** group, clicking **Diagram Filter**.

Finally, you can right-click a component and select **Filter Diagram**, which will have a specific effect, see below.

The diagram filter browser is by default docked; it appears in the lower left part of the main window (under the component browser if that is not undocked). It can be undocked as well as made to a tab of for example the package browser or component browser. For browser handling, see “Browser handling” on page 202.

Filtering on domains

An example of filtering on domains, in the Motor Drive demo. (This demo is opened by the command **File > Demos > Motor Drive**.):



In this case, the components belonging to Modelica.Blocks are not dimmed, while other components are.

Except for selecting to filter/display components from various packages/domains, you can also select **All**, or **None** to select/deselect all. You can control the opacity of filtered components by the bar. The opacity goes from 0.0 which is no opacity, to 1.0 which means fully opaque. The opacity setting corresponds to the flag `Advanced.DiagramFilterOpacity`. The default value of this flag is `0.15`. The opacity is saved between sessions.

The feature is important when for example trying to track which components have been inserted that are not from e.g. the Modelica Standard Library.

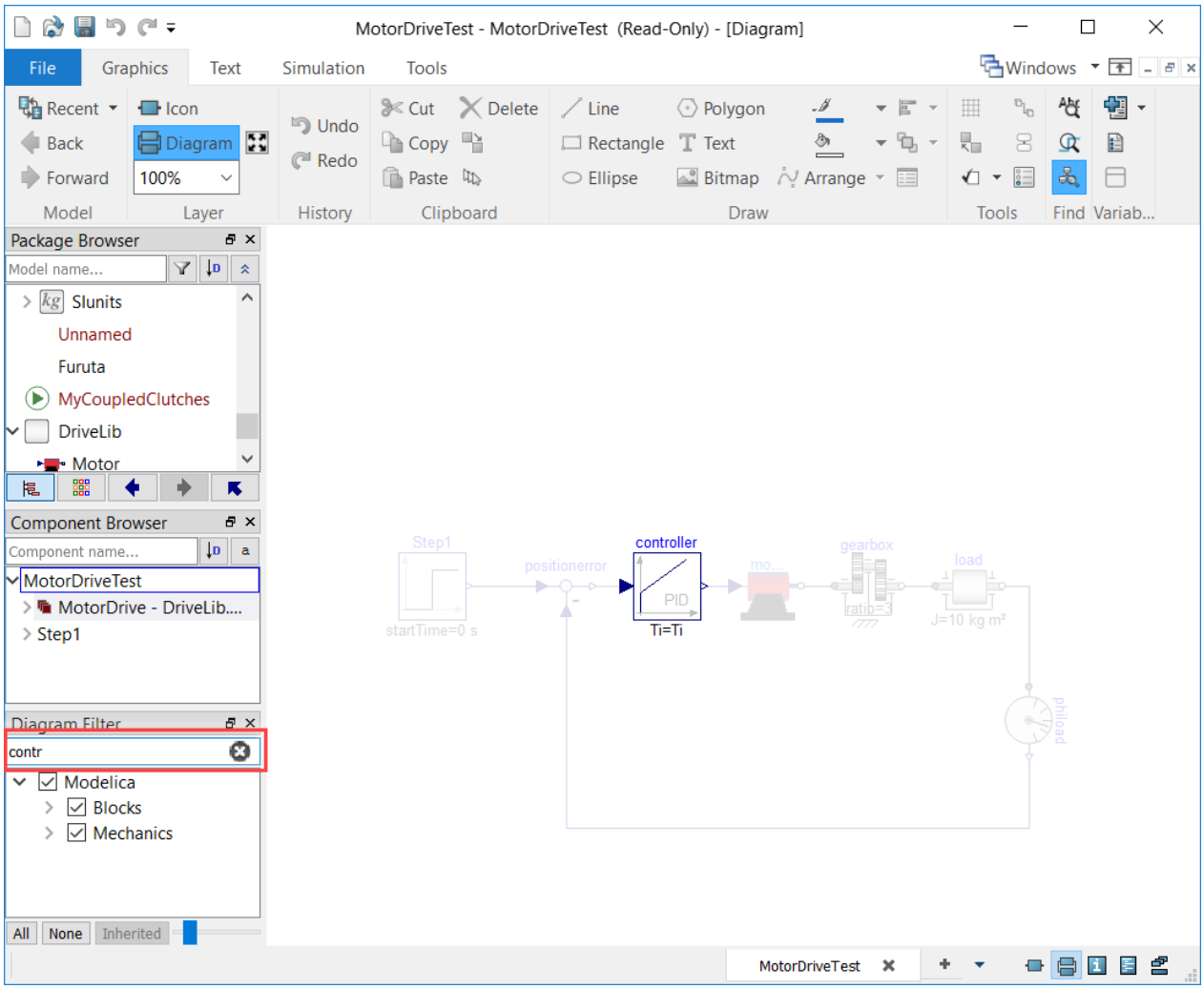
Note that the filtering is connector based:

- If any of the connectors of a component is extended from a base class that is part of a deselected package, the connector is seen as “filtered”.
- If *all* connectors of a component are seen as “filtered”, the component is dimmed.

Note also in some cases the domain “other” is present; this domain enables you to filer components not belonging to any domain.

Filtering on component name and domains

You can filter on component name using input field in the top of the Diagram Filter browser (see framed area below):



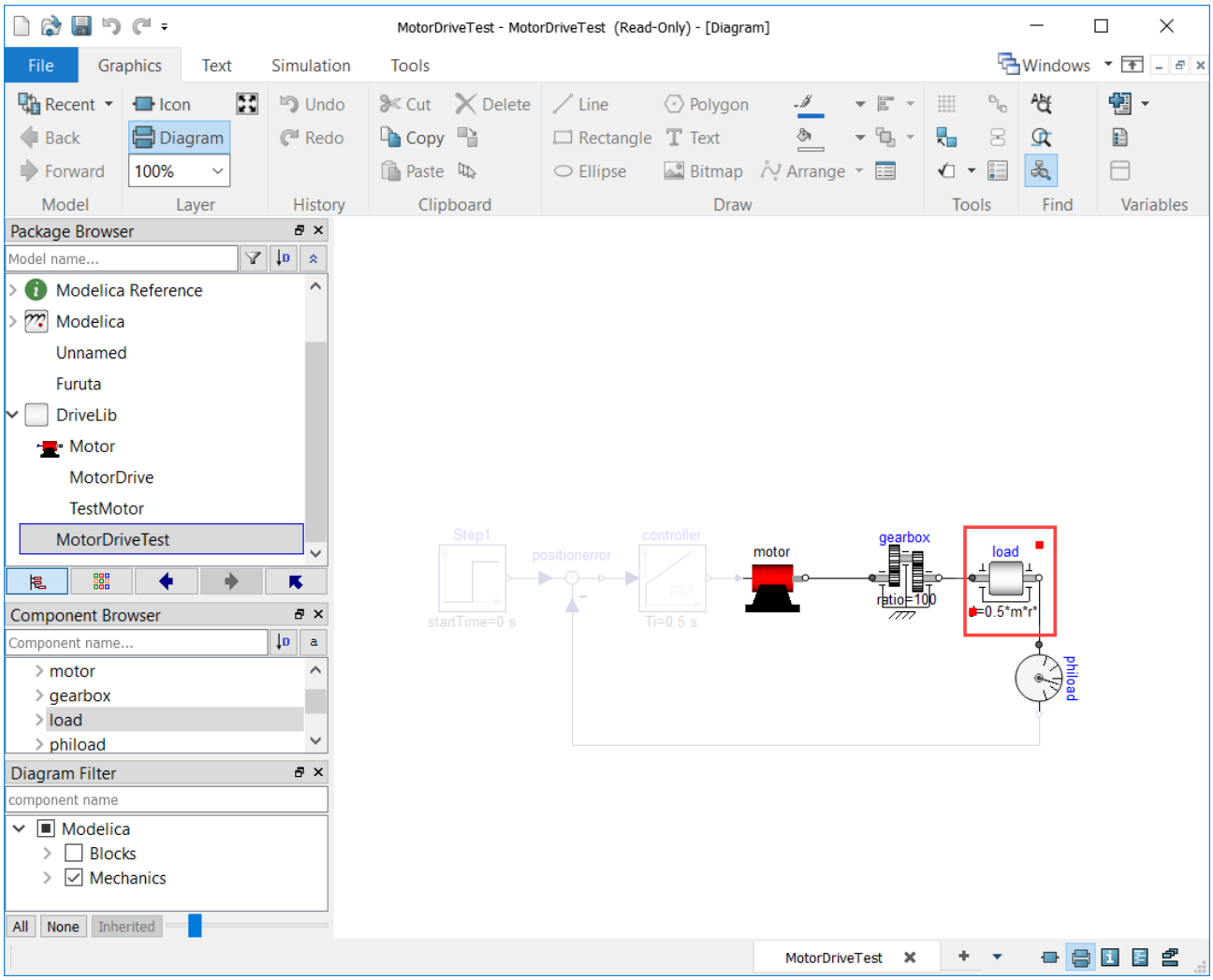
Typing in that field dynamically filters out components not matching this text.

Note that components filtered out by the domains will still be filtered out even if they match the component name being input.

Clicking the cross in the end of input line removes the typed name.

Activating or modifying filtering from a selected component

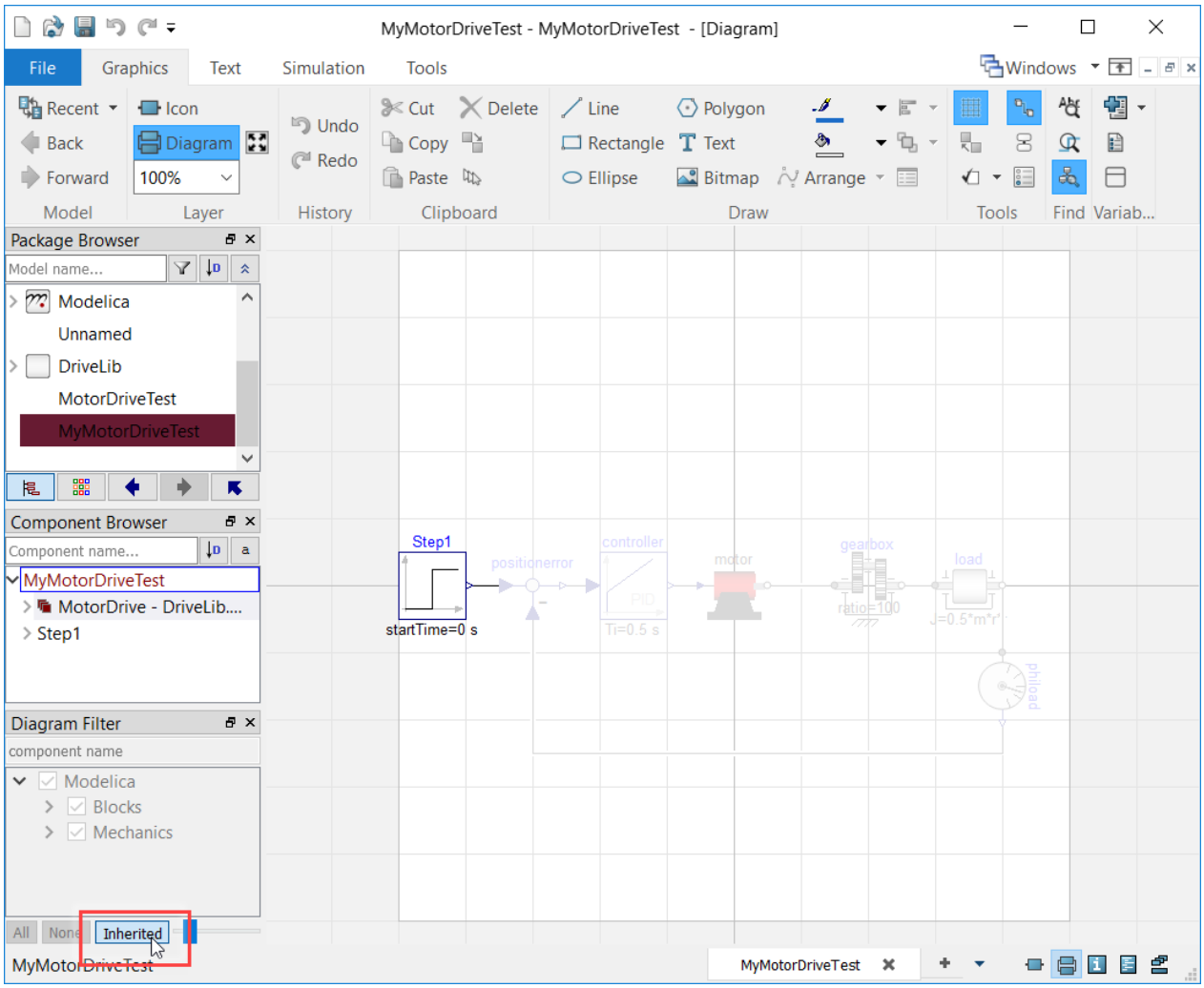
You can activate diagram filtering from a selected a component by right-clicking the component and selecting **Filter Diagram**. (that this specific command is named **Filter Diagram** and not **Diagram Filter** is to emphasize that it is an active filtering operation) The selection will then control what is filtered, the domain that contains the selected object is filtered. An example of a component belonging to the “Mechanics” domain (the component is marked in the image):



Filtering on inherited components and connectors

If the model is editable, it is possible to filter on inherited components and connectors, using the button **Inherited** in the diagram filter. Such components cannot be deleted, moved or replaced.

As an example, for the opened Motor Drive demo, right-click MotorDriveTest in the package browser and select **New > Duplicate Class...** and give it a new name MyMotorDriveTest. This model is editable. Now we can open the diagram filter and active the filtering of inherited components by clicking the button **Inherited** in the diagram filter. The result is:



Notes:

- The **Inherited** button works in toggle mode.

- When the inherit filter is activated, the other diagram filtering is disabled and showed as dimmed. When deactivating the inherit filtering, the other diagram filtering is activated again.
- The opacity can be changed also for inherit filtering.

4.2.2 Packages, models, and other classes

Creating packages

Creating a library with subpackages

Although you can create models at top level, it is recommended to collect models in libraries. If you have many models, the library can be divided into subpackages.

To create any package, the command New > Package is used. The command is available as the command **File > New > Package** or as **New > Package** in the context menu of a class in the package browser. The former is used to create libraries, the latter to create sublibraries (the location is prefilled).

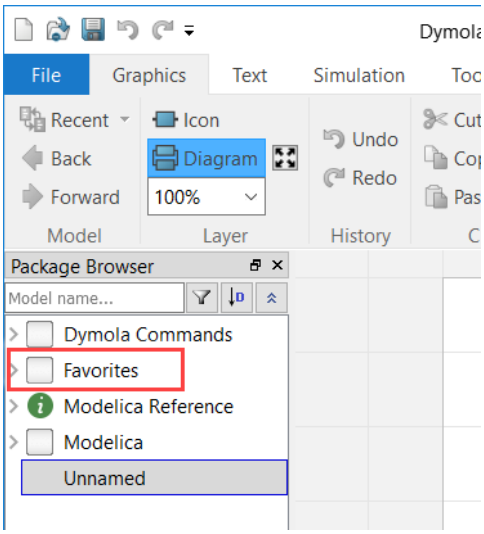
For a simple example of creating a library, see the chapter "Getting started with Dymola", section "Using the Modelica Standard Library", subsection "Creating a library for components".

The command has options to create partial packages ("template packages") and to extend ("reuse") packages. See "Advanced model editing" starting on page 329 for information on such more advanced features.

The Favorites packages

When working with inserting a number of components from different libraries, a useful feature is to use a "Favorites" package, where you can organize commonly used classes to get easy access to them.

By default, if no Favorites package is available, an empty Favorites package is created when starting Dymola:



If a Favorites package is already available from previous work, it is automatically loaded when starting Dymola.

Favorite from class

You can add classes directly to a favorite package with the command **Add as Favorite...** This command is available from the context menu when right-clicking a class in the package browser, as well as from the context menu of an open class, right-clicking an empty area in the diagram of the open class.

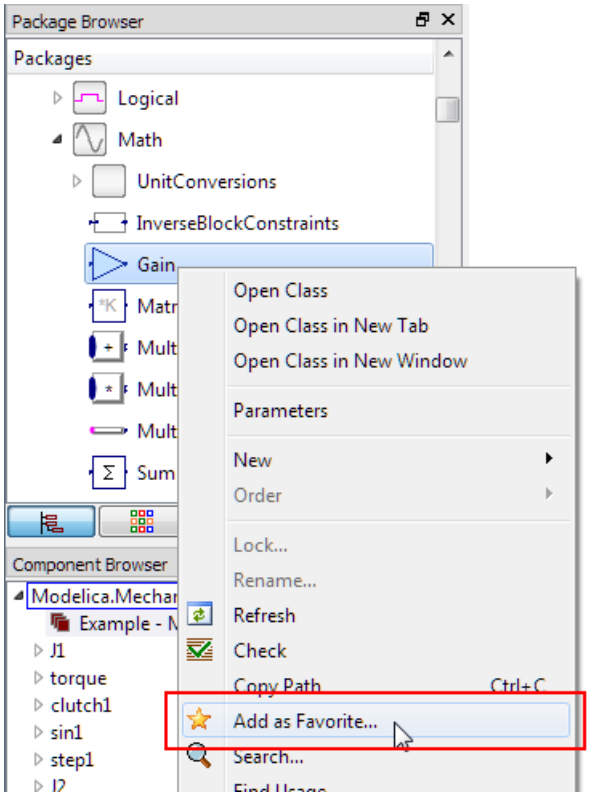
Favorite from component´s model, with parameter values included

You can however also add favorite models to a favorite package by adding the models of selected components, using the command **Capture Parameters > As Favorite**. This command is available both when right-clicking a component in the diagram layer and right-clicking the component in the component browser. An important feature when using this command is that the parameter values given to the component by the parameter dialog, as well as other modifiers, are included in the favorite model.

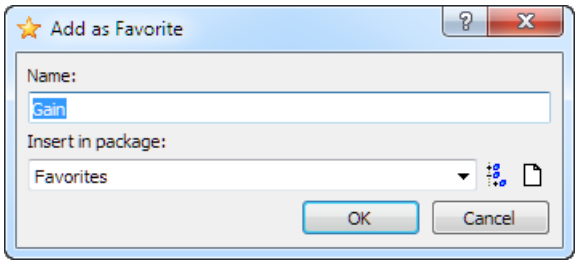
Assume that you have not created a Favorites package before. Whatever of the above command you use, you will by default have the automatically created favorite package to use for storage in the resulting dialog.

Working with favorites from class

As an example, right-click on a class in the package browser, and select **Add as Favorite...**,

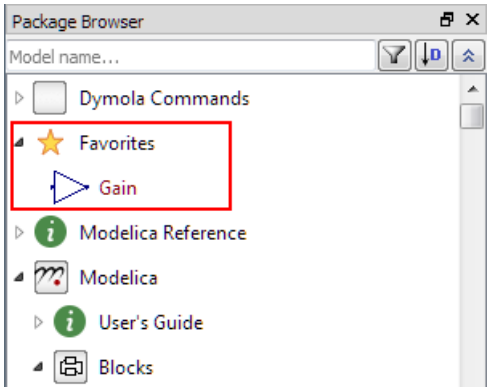


This opens a simplified class creation dialog, where the Favorites package is selected by default for insertion of the new favorite:

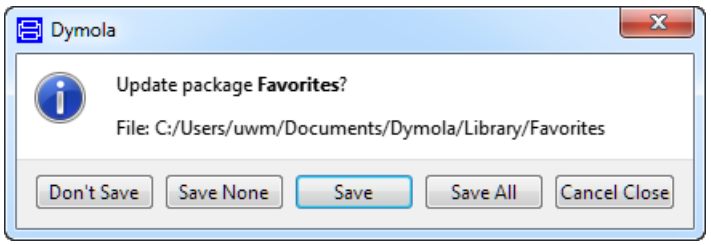


In the menu you can select the name of the new favorite class, and in which favorites package to insert it in. (To create a new favorites package to store the new favorite class in, click the **New Package** icon . This opens the normal Create New Package dialog. Note however that the Favorites package always is the default selection, even if you have created packages with other names.)

If you now click **OK** keeping the default selection you will get:



If you now exit Dymola you will get:



The package Favorites is by default created in a folder `Library\Favorites` in the `…Documents\Dymola` folder. The folder structure is automatically created if not present. (If the environment variable DYMOLAPATH contains another path, that path is used instead of …Documents\Dymola.)

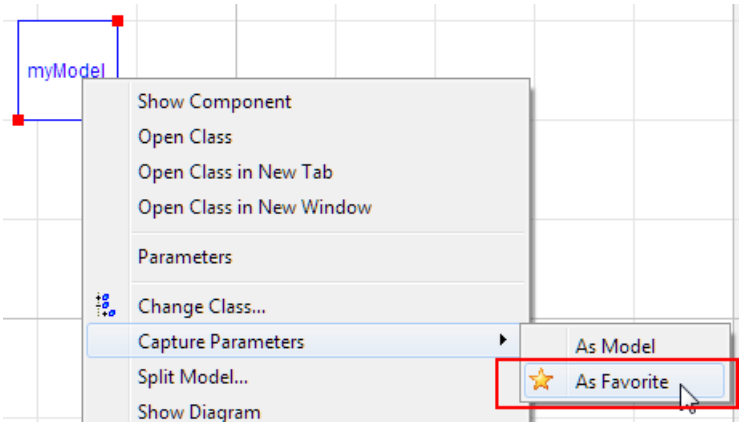
The `Favorites` package looks like an ordinary package, but behaves differently:

- When the `Favorites` package is opened, the favorite classes in it will not be loaded (except that Modelica Standard Library will be opened).
- If you have a favorite (made from `Modelica.Blocks.Math.Gain`), drag-and-drop of this class will insert a `Modelica.Blocks.Math.Gain` component, not a `Favorites.Gain` component.
- A favorite package is similar, if you have as favorite the package `Math` (made from `Modelica.Blocks.Math`), drag-and-drop of its `Gain` will insert a `Modelica.Blocks.Math.Gain` component, not a `Favorites.Math.Gain` component.

Working with favorites from components

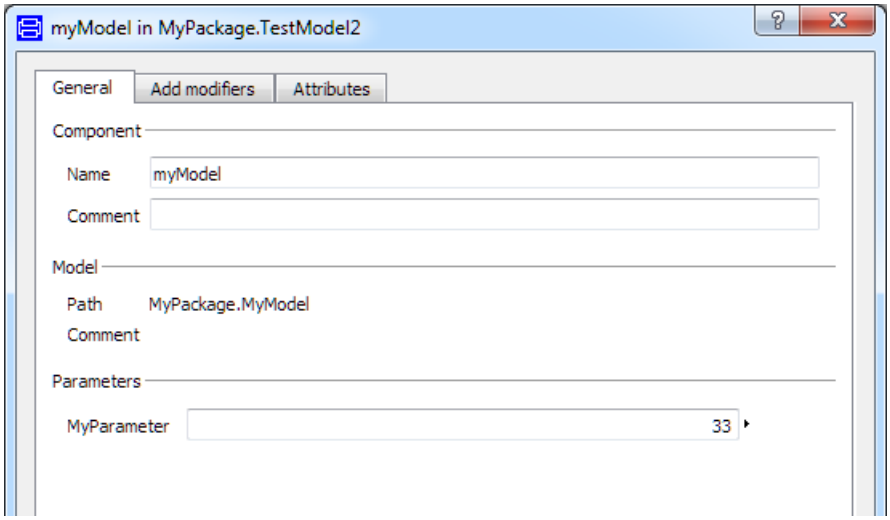
The command **Capture Parameters > As Favorite** works the same as the **Add as Favorite…** command above, except that it adds a component´s model with parameter values and other modifiers included as a favorite model in a favorites package.

As an example, consider a component myModel with a parameter MyParameter that has been given the value 33 by using the parameter dialog. Now, right-clicking this component and making it a favorite model:



A similar Add to Favorite submenu as for the **Add as Favorite…** command in the example above appears. The component will be added as a model in the selected favorites package when clicking **OK** in this menu.

Now, dragging this model into another package to test it, and right-clicking it and selecting **Parameters** gives the dialog (if we kept the name when adding it as favorite):



The given parameter value 33 is kept; the dialog is actually the same as for the original component.

Creating favorite packages using code.

Note that there is a more advanced way of creating favorite packages, possibly with prefilled parameter values when creating components from the classes. See chapter “User-defined GUI”, section “Extendable user interface – menus, toolbars and favorites” for more information.

Creating a model

You can create a model in a number of ways. For the case of warnings about potentially conflicting Modelica names when creating a model, see "Creating other classes" below.

Creating an empty model

To create an empty model, the command New > Model is used. The command is available as the command **File > New > Model** or as **New > Model** in the context menu of a class in the package browser. Using the latter alternative, the location of the new module is prefilled, as the **Insert in** field.

For a simple example of creating an empty model, see, in the chapter "Getting started with Dymola", section "Using the Modelica Standard Library", subsection "Creating a model for an electrical DC motor".

The new model can be selected as "partial" in the creation dialog, meaning it should be used as a template. See "Advanced model editing" starting on page 329 for information on such more advanced features.

Creating a model by extending from an existing model

Extending a model allows you to reuse an existing model with inheritance. For example, you can create a new class from an existing one and add more objects to it. This new class inherits any modification from the original class it is extended from. Note that is also includes any experiment annotation defined in the corresponding base class. (The experiment annotation corresponds to the settings in the **General** tab of the simulation setup, for example, start time, stop time and algorithm. For more information about this tab, see index entry "*simulation setup : general tab*" in the index in the end of this manual.)

For a simple example how to extend a model, see the chapter "Getting started with Dymola", section "Using the Modelica Standard Library"; subsection "Creating a model for the motor drive".

Note the difference between extending and copying; in most cases extending is more favorable.

Note also the possibility to extend from an existing component with partameter values included, see below.

For more advanced use, see "Advanced model editing" starting on page 329 for more information.

You can extend a model using the command New > Extend From. The command is available as the command **File > New > Extend From…** or as **New > Extend From…** in the context menu of a class in the package browser. It is also available when using any **New > Model** command, as **Extends (Optional)**, where you can browse for the class you want to extend from. The advantage using the "Extends From" command is that a number of fields are prefilled.

Creating a model by extending from an existing component, with parameter values included

You can extend a component, with parameter values and other modifiers included, to a new model by the command **Capture Parameters > As Model**. The command is available for a component in the diagram or in the component browser. The advantage of this command is that you can make a component with a number of valuable parameter values and other modifiers reusable as a model.

For details, see "Creating a new model from a component with parameter values included" starting on page 336.

Creating a model using the "Split Model" command

The basic idea with "split model" is to use a selection of already existing components (including connections) in a model and from them create any of the following:

- A submodel, to which the selected components are moved, to simplify the diagram layer.
- A new model, to which the selected components are moved, and then this new module is extended to the original one, for being able to reuse the collection of components some elsewhere.
- A new model to which copies of the components are added.

However, these are more advanced features; please see "Advanced model editing" starting on page 329 for more information.

Creating other classes

When you want to create other classes than packages or model, you can still use the command New. The command is available as the command **File > New** or as **New** in the context menu of classes in the package browser. Giving the command, you select what class you want to create.

In some cases, you can also use the "Split Model" command for creating other classes. See above.

Warning about potentially conflicting Modelica names when creating or renaming classes

When creating a class (or renaming it) you may in some cases risk conflicts since some operating systems, for example, Windows, do not have case-sensitive file names, while Modelica names are. To prevent such problems, a warning is given when creating or renaming a class, if the new class differs only in upper/lower case from an existing class (in the same scope, for example, in the same package).

There are two cases, one is when creating a model/class that is similar to an existing model/class in the same package, for example, creating `MYMODEL` when `MyModel` is already present in the same package. You will in this case be given a warning and a suggestion to use another name.

The other case is when you create a model/class that has similar name as a model/class name of an instance used in the package, but coming from another package. As an example, dragging, from the package `AnotherPackage`, an instance `MyTest` to a model in the package `ThisPackage`, and then, in `ThisPackage`, create a model `ANOTHERPACKAGE`. In this case

you will also be given the alternative to update the references according to the new model you create.

Duplicating a class

You can duplicate a class in three ways:

- Using the command **New > Duplicate Class**. The command is available as the command **File > New > Duplicate Class…** or as **New > Duplicate Class…** in the context menu of a class in the package browser.
- By pressing **Ctrl** while dragging the class in the package browser. See section "Drag and drop of classes in the package browser" on page 234 below.
- By using **Copy** and **Paste** in the context menu of the class in the package browser. They work the same as dragging.

Note. If you want to reuse a class, it is usually better to extend from it than duplicating it.

Renaming a class

You can rename a class in four ways:

- Using the context command **Rename…** for the class in the package browser.
- In the package browser, select a class; then click it again, or press **F2**. An inline editor opens where you can enter a new name. Finalize by pressing **Enter** (or **Escape** to cancel).
- By dragging the class in the package browser. See section "Drag and drop of classes in the package browser" on page 234 below.
- By using **Cut** and **Paste** in the context menu of the class in the package browser. They work the same as dragging.

Note that although the references to the class are updated in most cases, renaming a used class might be a complex operation worth considering before performing. For more information about consequences when renaming, please see the Rename command in the section "Context menu for components in the package browser" starting on page 543.

Extend from a class

Extending models is described above. The commands can be used for other classes as well, and the principle is the same.

Drag and drop of classes in the package browser

Classes in the package browser can be dragged and dropped; this makes for example rearranging of the classes in a package more convenient.

When dragging a class to a location in another top level package than where the class resides, the **Move** dialog is by default opened, to let the user use this dialog to conclude the moving of the class.

If you press **Ctrl** while dragging you will copy the class. If the class is dragged to a location in another top level package than where the class resides, the **Copy** dialog is by default opened, to let the user conclude the duplication of the class.

Note that **Cut**, **Copy**, and **Paste** are available in the context menu of a class in the package browser; these commands work the same as drag and drop.

Note also that if you want to reuse a class in any way, extending it is often to prefer rather than for example duplicating it.

The feature of drag and drop of classes in the package browser can be disabled in two ways:

- By unchecking the setting **Reorder using drag and drop** in the **General** tab of the options dialog. This dialog is reached by the command **Tools > Options…**. See section "General options tab" starting on page 511.
- By setting the flag `Advanced.DragDropPackageBrowser = false;`.

You can decide when the **Move**/**Copy** dialog is to be displayed when you move/copy classes. This can be done by any of the alternatives in the **Show dialog when rearranging classes in Package browser** group setting, available by the command **Tools > Options…**, in the **General** tab. See section "General options tab" starting on page 511.

4.2.3 Components and connectors

Inserting a component or a connector

Components and connectors are usually inserted by dragging a class from the package browser, or a library window, into the diagram layer of an edit window. The dragging procedure is:

- Select a class in the package browser or in a library window.
- While pressing the left mouse button, move the cursor to an edit window and position the icon at the desired position.
- Release the left mouse button to confirm the insertion.

The size of the inserted component or connector is according to the specified component size defined in the enclosing class.

Name and modifiers are set after inserting.

The newly inserted component or connector is automatically given a name composed from the name of its class and a sequence number (to avoid name conflicts). The first inserted component of e.g. class Brake will be brake, the next brake1, followed by brake2 etc. (A class name begins with an upper-case character, a component name by a lower-case character.)

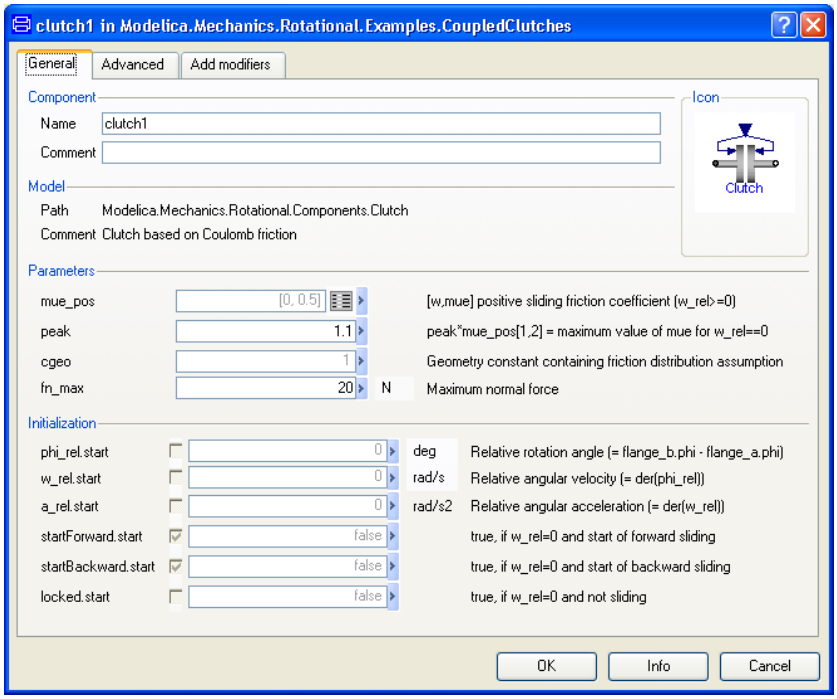
Note that components can be added anywhere in the diagram layer; the coordinate system can be conveniently rescaled using mouse dragging of the coordinate system boundary. For more information, see section "Coordinate system" on page 197.

Editing name and comment

Name and comment can be edited in the parameter dialog. This dialog is reached either by double-clicking on the component or by selecting **Parameters…** in the context menu of the

component. (Opening the parameters dialog by double-clicking a component is default but can be changed. If double-clicking does not work, **Shift+double-click** will do. See “Graphical Editor options tab” starting on page 516 for details.) The parameter dialog can look like:

Parameter dialog.



In the **General** tab the **Component** group gives the possibility to edit the name of the component and insert a comment (if the component or connector is not read-only and provided the component or connector is found at the top-level in the component browser). The **Icon** group shows the icon of the component, while the **Model** group shows the path to it and the comment to the base class.

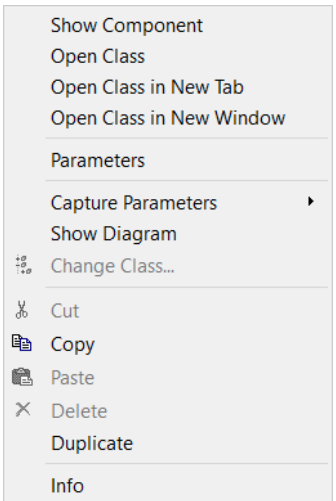
Notes:

- Changing the name of a component this way by default also updates references to it. This corresponds to the default value of the setting “Updates other classes when renaming component”, reached by the command **Tools > Options...** For details about this setting, see this setting in section “Package Browser options tab” starting on page 512.
- Due to the item above, it is not recommended to change a component name by editing the component name in the Modelica text layer; no automatic update will occur.

The context menu

Right-clicking when the cursor is located on an object (or when one or several objects are selected) presents a context menu:

The context menu of an object.



A similar menu is presented when right-clicking in the component browser. For more information about the choices, please see section “Context menu: Components” on page 543.

Editing parameters and variables

Below the use of the parameter dialog is described. Note that initial conditions for certain parameters and variables also can be changed by saving values from the simulation directly into the current model, or even extend the model and save to the new model. See section “Programming when simulating – saving start values in model” on page 296 for more information.

The parameter dialog presented above gives also possibility to edit parameters and other modifiers.

The modifiers that appear and can be set are:

- Normal parameters declared with the keyword parameter in the model. These can be set to a value that is either numeric or computed from other parameters. (For rules concerning parameter values, please see section “Parameters, variables and constants” on page 376.)
- Replaceable classes in the model. Please see the section “Parameter dialog – advanced” on page 385 for more information about this possibility.
- Hierarchical and other forms of modifiers, e.g. `x(start=2, unit="m")`. Please see the section “Parameter dialog – advanced” on page 385 for more information about this possibility.
- Add modifiers. This is a comma-separated list of free form modifiers, and can be used to add free form modifiers for new variables. The next time the modifier dialogue is opened the list elements have been moved to the items. The field does not appear if the model is read-only. The feature is an advanced feature, for more information and an example, see

section “The Add modifiers tab: Using the Add modifiers tab to add modifiers” on page 390.

For a component or connector that is not read-only, name, comment and modifiers can be changed (provided the component or connector is found at the top-level in the component browser). Note that it is possible to set parameters even if the component or connector is not at the top level; these are stored as hierarchical modifiers.

Component modifiers are displayed in a table with columns for name, value, unit, and description of the variables. The list of variables is extracted from declarations in the class and its base classes, and the actual modifier values are defined in the component in the edited class.

The actual value of a variable is changed by clicking in the corresponding field and typing in the new value. An existing actual value is erased by clicking in the corresponding field and pressing the **Delete** key. The **Tab/Shift-Tab** keys move from one input field to another.

Parameters can be organized in Tabs and Groups.

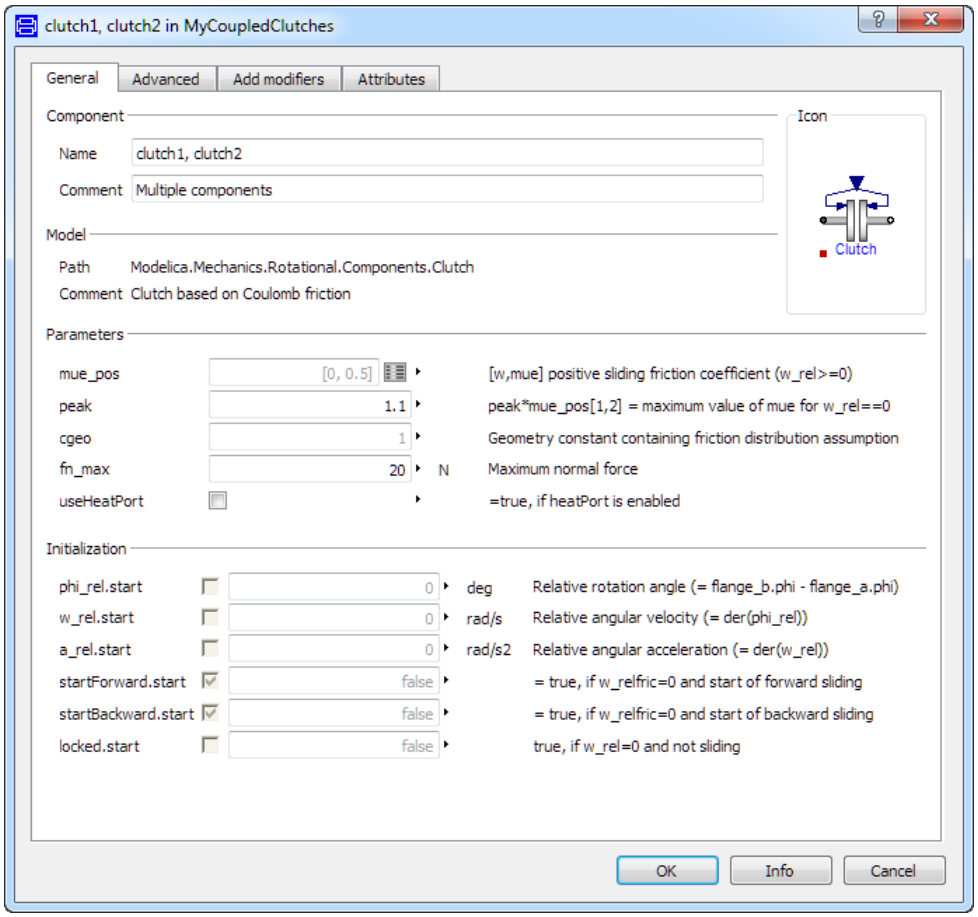
The parameters of the parameter dialog can be structured at two levels. The first level consists of tabs which are displayed on different pages of the window. The second level is a framed group on the page. Different variables with the same tab and group attributes are placed together.

The user can define tabs when declaring variables. See section “Annotations tab” on page 382.

A parameter dialog for a standard library component is exemplified above.

When multiple components have been selected, the parameter dialog contains the intersection of all parameters. If the selected components do not use the same value for a parameter, then the symbol <<...>> is shown instead of a value; changing this field sets the parameter for all selected components.

A parameter dialog for multiple selections.



General tab

Component group

Here the name and comment of the component can be edited.

Icon group

This group shows the icon of the component.

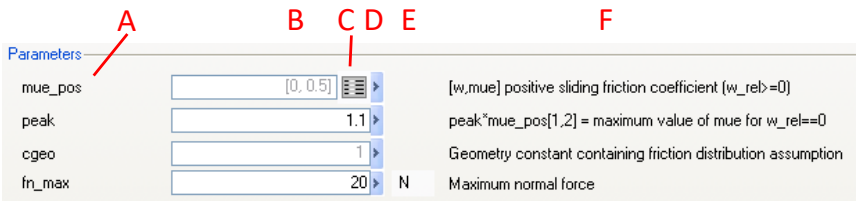
Model group

Here the path to the corresponding class is displayed (where the component comes from). In this case, it is a Clutch from Modelica Standard Library, package Mechanics, and sub-package Rotational. In addition, the class-specific comment is shown.

Parameters group

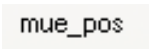
The parameters group lists the parameters in the component. From left to right, a number of “columns” can be seen

The parameters group.



(indicated by letters A-F also in the side head below). The parameter part of the parameter dialog automatically gets a scrollbar for any dialog tab that would otherwise be too large for the screen.

Let us use the first parameter in the figure above (mue_pos) as an example:

A. 

The *name* of the parameter is the first column.

B. 

An *input field* where grey values indicate default values, e.g. from the class of the component. One can see where the default originates from by clicking the question mark on the parameter dialog header, and then on on the input field, or using **View Parameter Settings** in the context menu of the parameter input field. Here a new value can be entered (which will then not be dimmed – compare the next parameter).

It is possible to enter values with prefixed units, also if that prefix is currently not selectable from the display unit after the value in the dialog. See item “E” below for details.

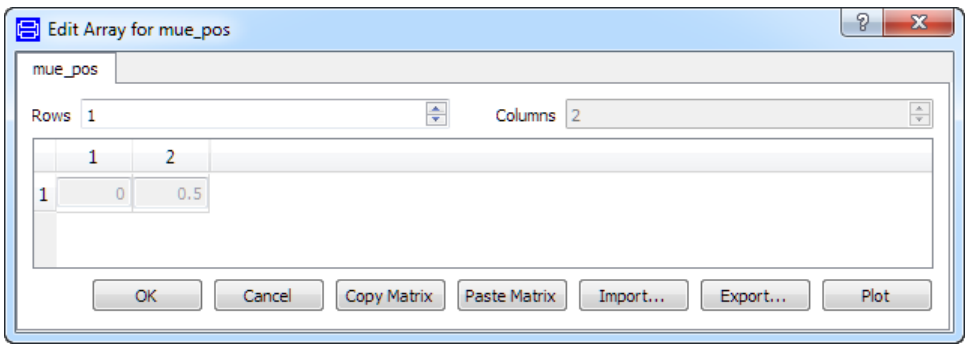
In some cases a drop-down menu is available to select between values. It should be noted that it is possible to enter values without using the drop-down menu. This enables the use of expressions, for example. Note that code completion by **Ctrl+Space** can be used to display a list of alternatives to select from. See section “Code completion” on page 283 for more information on code completion.

By right-clicking a context menu is available; the same menu that will be shown by clicking on the arrow button to the right of the input field, see that.

C. 

Sometimes an **Edit** button is available as a third column. By clicking on that the following window will appear (in this example; the **Edit** button can result in a color editor, file browser etc depending on the context):

Window from the edit button.

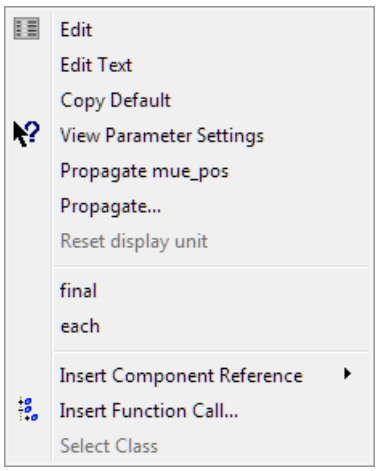


This is a matrix editor, which enables working with structured parameters. For more information, please see the section ”Edit menu of a matrix of values (matrix editor)” on page 561.

(You can actually paste a matrix copied from e.g. Microsoft Excel into the parameter field without using the matrix editor. You can only use **Ctrl+V** to paste it, and there is no check what can be pasted until you translate/simulate the model. (You can try to paste a 4x4 matrix into the parameter field of a parameter of fixed size 3x3; the parameter input line shows 4x4, the matrix editor only 3x3. You will get an error when trying to translate/simulate such a model.)

D. 

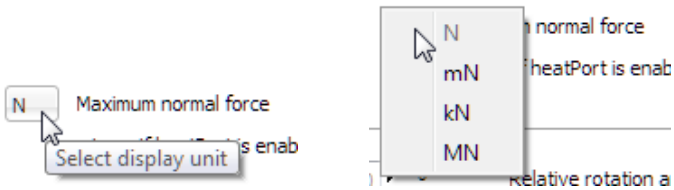
An *arrow button* will form the next column. That button will take up the following menu:



This menu is also the context menu in the parameter input field. Please see section “Context menu: Parameter dialog; parameter input field” on page 572 for more information about the alternatives.

E.

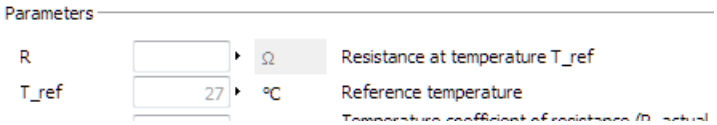
In the next column *display unit* (if any) is presented. If no display unit is defined in the variable, the default display unit will be shown. If the display unit is selectable, it has a white background. That is the case for the unit N for the parameter fn_max. By resting the cursor on the N, a button is visible (left figure below). By clicking on the button a selection can be made (the figure to the right below):



When the display unit is changed, the value of the parameter edit field is automatically converted to the chosen display unit if it hasn’t been edited.

Note that it is possible to use the feature of typing in value with prefixed unit to create prefixed display units, given the base unit is already available as display unit.

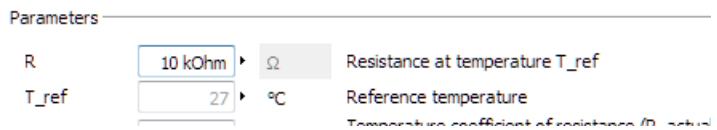
As an example, consider part of the parameter dialog of an ordinary resistor:



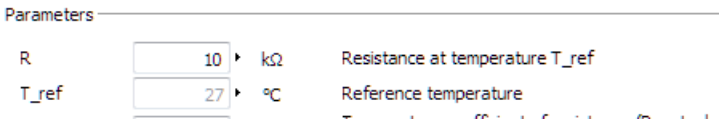
The gray background shows that there are no display unit selections. You can however enter any of the following in the input field to input 10 kiloOhms:

- 10 kOhm
- 10kOhm
- 10 k
- 10k

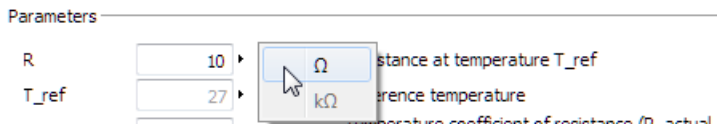
The below shows entering the first alternative:



Clicking **OK** and displaying the parameter dialog again results in:



The display unit selection has also been updated:



The display unit selection for Ohm will be updated for all display unit selections of Ohm. This new prefixed display unit will also be available for such parameters when plotting them, see next chapter.

The context menu above makes it possible to reset the display unit by the entry **Reset display unit**.

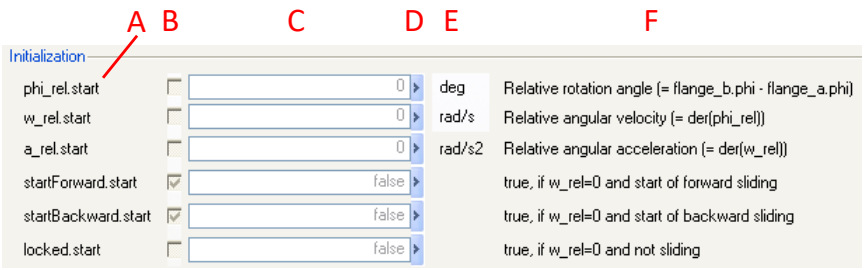
(It is possible as well to change the current display unit for a curve in the context menu of the plot window.) For more information about display units, please see section “Display units” starting on page 423.

F. Finally, a comment to the parameter will form the last column:

[w,mue] positive sliding friction coefficient (w_rel>=0)

Initialization group

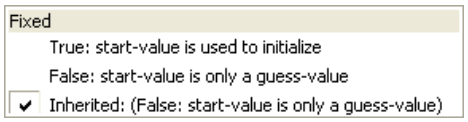
The Initialization group lists the initial values in the component. From left to right, a number of "columns" can be seen:



(indicated by the letters A-F also in the side head below). Let us use the first start value in the figure above (phi_rel.start) as an example:

A. phi_rel.start The *name* of the start value is the first column.

B. A *checkbox* that shows whether the start value is a fixed start value, guess value or if it is inherited forms the second column. When clicking on the example checkbox the following is displayed:



The fixed-attribute is described in Modelica and a value of true means that the start-value must be satisfied during initialization, whereas false means that the start-value is only intended as a guess-value for a non-linear solver. Inherited means that the fixed modifier is removed, and as a consequence, fixed modifier is set from other levels. The actual value with description is shown in the popup-menu.

C. 0 An *input field* where grey values indicate default values, e.g. from the class of the component.

One can see where the default originates from by clicking **What's This?** on the input field, or using **View Parameter Settings** in its context menu. Here a new value can be entered (which will then not be dimmed).

D. An *arrow button* will form the next column. Please see the arrow button above for description of the corresponding menu.

E. deg In the next column *physical unit* (if any) is presented. In this case the unit "deg" is also possible to select (it has a white background). Please see the corresponding entry in the description of parameters above.

F.

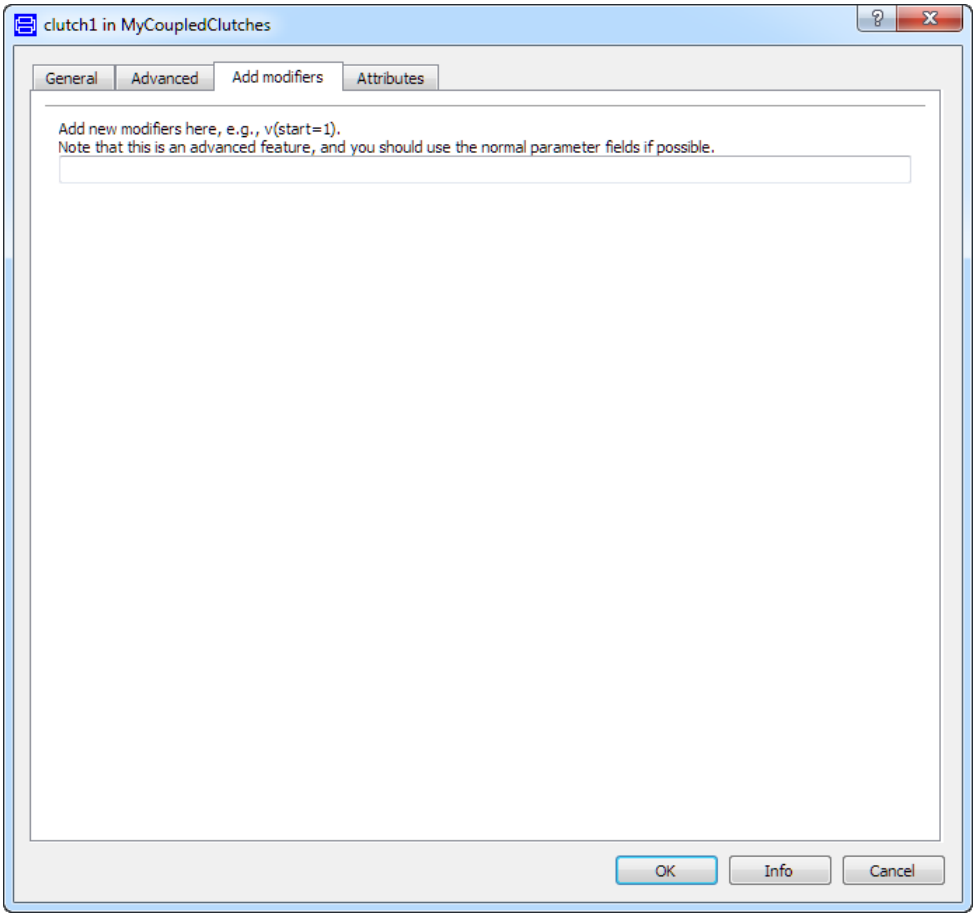
Finally, a comment to the parameter will form the last column:

Relative rotation angle (= flange_b.phi - flange_a.phi)

Advanced tab

The Advanced tab is not a general tab. For creation of tabs etc please see section “Parameter dialog – editing the dialog” on page 393.

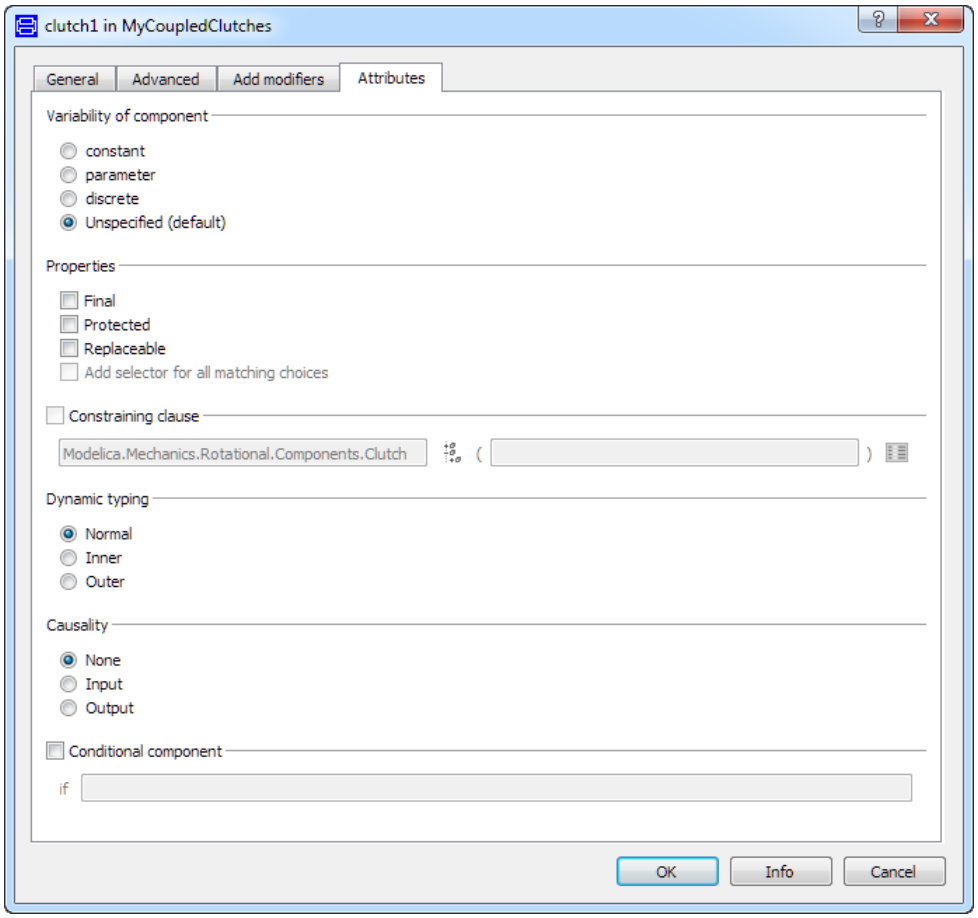
Add modifiers tab



The **Add Modifiers** tab is used to add new modifiers, e.g., start values or nested modifiers.

Component modifiers are displayed in a table with columns for name, value, unit, and description of the variables. The list of variables is extracted from declarations in the class and its base classes, and the actual modifier values are defined in the component in the edited class. The feature is an advanced feature, for more information and an example see section “The Add modifiers tab: Using the Add modifiers tab to add modifiers” on page 390.

Attributes tab



The attributes reflect component properties specified in the Modelica language. The settings in the figure above are the default settings.

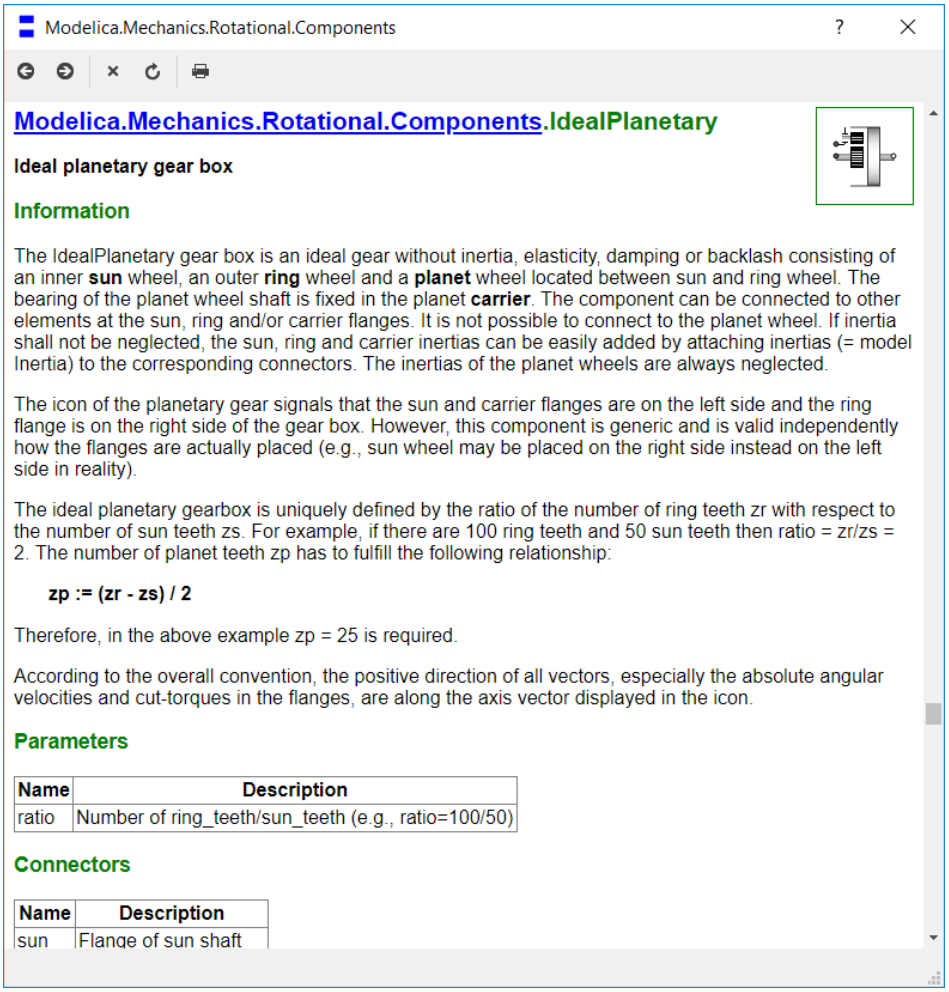
For more information about the settings in this tab, see section “The Attributes tab” on page 392.

Buttons

Three buttons are available in the parameter dialog, **OK**, **Info** and **Cancel**.

Pressing the **OK** button will generate a modifier list for the component. Only modifiers with non-empty value fields are used.

The **Info** button displays the Modelica documentation for the class using an information browser:

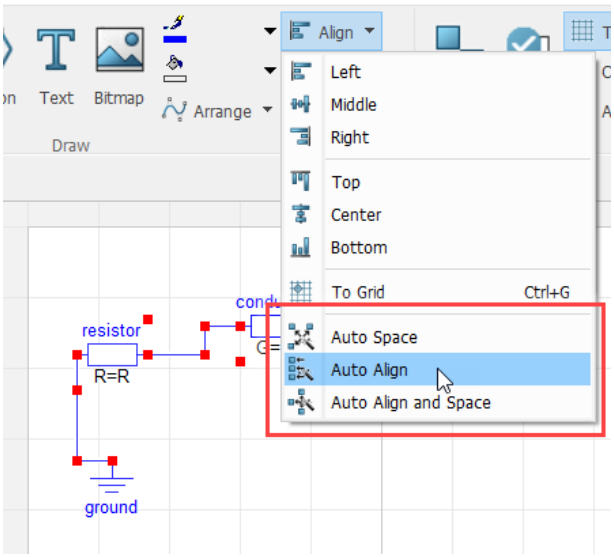


Depending on the complexity of the class, the documentation can contain pictures, tables, references etc. For details on what is shown in the browser, see section “Looking at the documentation layer (and more) using the Info button/menu entry” starting at page 306.

Spacing and aligninig components automatically

You can use space and align components using three commands. To work with specific components, you can multiselect them, otherwise, if no component is selected, the commands work on all selectable components. Note that the commands only works on components, not on graphical primitives like lines or rectangles.

The commands are:



- **Graphics > Align > Auto Space** to space the components. Note that the size of the components are not modified.
- **Graphics > Align > Auto Align** to align the components, in both x and y direction. Notes:
    - Components that you have previously generated aligned by other commands may not be aligned to each other anymore; they may be aligned with other components depending on the algorithms.
    - For manually well-designed classes, you may not improve the layout by this command.
- **Graphics > Align > Auto Align and Space** to both align and space the components corresponding to using both the above commands. The notes of these commands applies to this command as well.

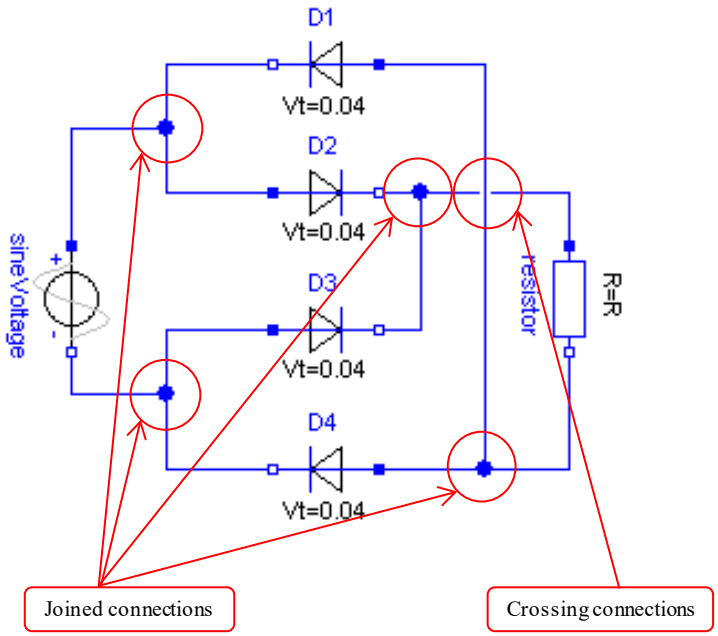
4.2.4 Connections



A connection is a graphical representation of a `connect` statement between two connectors. Connections can be made when the diagram layer of an edit window is in connect mode, as indicated by the pressed-down tool button. (This is the default mode.) If the model is read-only, or connect mode is off, connections cannot be made.

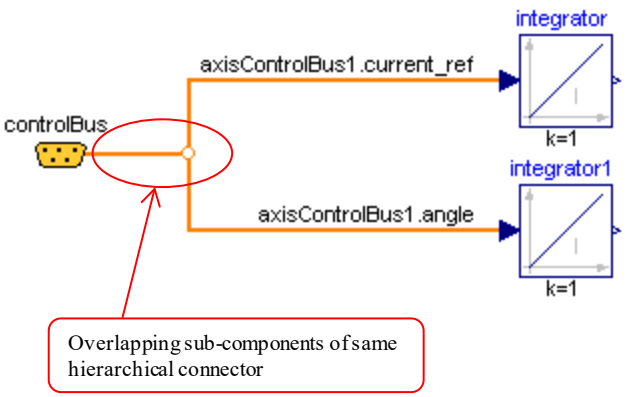
If connections are joined (more than one connection to the same connector), this will be graphically represented by a circular solid “blob”. Connections crossing each other will be indicated by a tiny space in the crossing for one connector.

Example of joined and crossing connections.



A circular *hollow* "blob" indicates graphical overlapping of connections of sub-components from the same hierarchical connector (e.g. a bus).

Example of overlapping sub-component connections.



Creating connections using Smart Connect

Smart Connect simplifies connecting objects. The two first features are enabled by default, but can be disabled generally by setting the flag `Advanced.SmartConnect=false`. If disabled, please see next section for the corresponding information about how to create connections. The third feature (start connection from connection / connect to connection) is always available.

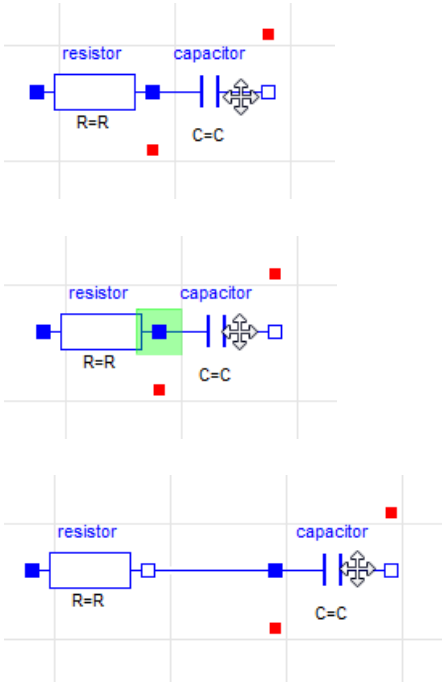


These features are also disabled when **Connect Mode** is not selected. You can select the connect mode on the **Graphics** tab, in the **Tools** group.

Working with the features, **Ctrl+Z**, or **Escape**, are the general commands used to undo an action.

Automatic connection when placing a connector on to of a matching connector

By placing a connector on top of a matching connector of another component, a connection is automatically created between the two connectors. In the example below, a capacitor is dragged to fit the connector of a resistor. A short highlight is seen when the connection is created and connected; the result is seen when dragging the capacitor away.



The color of the highlight indicates degree of success:

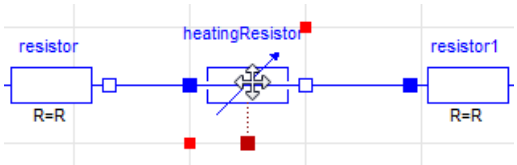
- Green: Connection succeeded.
- Yellow: It is possible to create a connection, but not with smart connect. This is the case with e.g. expandable connectors and array of connectors.
- Red: Connection not possible; e.g., connectors are not matching.

Note that if a connection dialog is needed to connect, it is displayed automatically. Only one connection that requires the connection dialog is supported per operation.

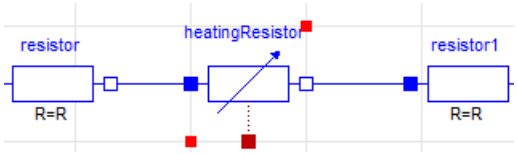
Drop component on connection

A component can be dropped on a connection for automatic connection (if the connection does not require a connection dialog).

Splitting a connection when dropping a matching component is also supported, e.g. dropping a heating resistor on the connection of two resistors:



will result in:



Two highlights are seen when dropping the component, one for the connection to the right of the component, and one for the connections to the left of the component.

The layout of the original connection is kept. Dropping on a vertical connection is also supported.

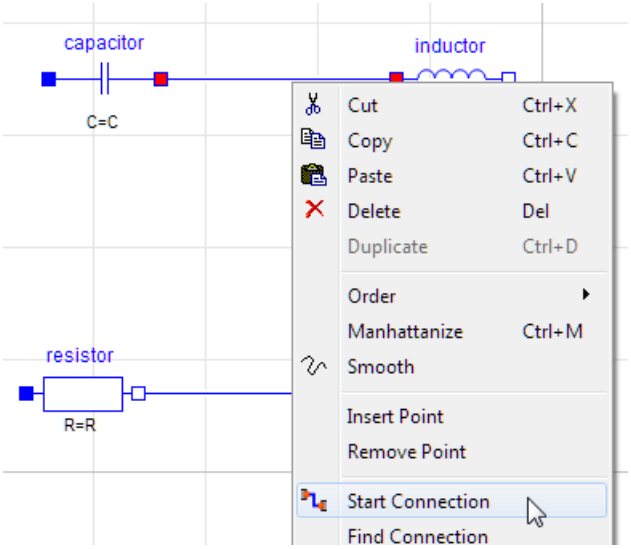
The connection is split only if the relevant sub connectors of the dropped component match the connection (in the above example the positive and negative pin).

Note that if a connection dialog is needed to connect, it is displayed automatically. Only one connection that requires the connection dialog is supported per operation.

Start connection from connection / Connect to connection

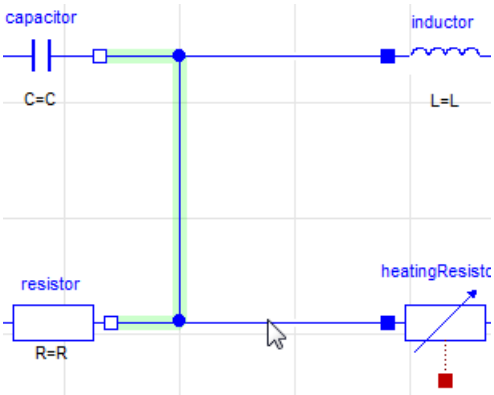
A connection can now be started from an existing connection, and a connection can be ended on an existing matching connection.

To start a new connection from an existing connection, right-click the existing connection and select **Start Connection** from the context menu:



The connect command is started and the connection can be created the normal way.

To end a connection on an existing matching connection, just click on the wanted location on the existing connection when connecting. (The cursor will be displayed as a hand on the existing connection if connection is possible.) The resulting connection is shortly highlighted:



Note that if a connection dialog is needed to connect, it is displayed automatically. Only one connection that requires the connection dialog is supported per operation.

Creating a connection without using Smart Connect

Connections are defined interactively in Dymola in a manner similar to drawing lines.

- Click on a connector. Keep the mouse button depressed and move the mouse. This will create the beginning of a graphical connection symbolized by a line. (Please note that some components are connectors themselves, and no specific connector is then visible, the whole component acts as a connector.)
- Once started this way, the mouse button can be released enabling the creation of drawing multiple line segments (see "Lines and polygons" on page 262). Note that the mouse button must be pressed down while drawing the first line segment.
- Click on another connector to finish the connect operation. Double-clicking outside a connector presents the context menu.

The line segments of the connection snap to the grid of the class by default, but are then adjusted so the connection end points reach the corresponding connectors. This may cause skewed lines if the grid is coarse, but it can usually be adjusted by right-clicking the connection line and select **Manhattanize**. The manhattanize operation inserts points until there are at least four points to work with, so the line can be drawn at right angles.

Automatic manhattanize is by default enabled, this means that connections are manhattanized immediately when created, moved or reshaped. Moving a component automatically manhattanizes all connections to the component. Note however that pressing **Shift** at the end of the connection operation negates the meaning of the automatic manhattanize option. Automatic manhattanize is controlled by the flag `Advanced.ManhattanizeConnection`. The value of the flag is by default `true`.

Manual Manhattanize can be obtained by right-clicking the connection and selecting **Manhattanize**. There is also a corresponding command in the **Graphics** tab: **Arrange > Manhattanize**.

When drawing a connection, the default color is taken from the starting connector. The connection gets the color of the outline of the first rectangle, polygon or ellipse in the icon layer of the starting connector. This change facilitates color-coding of connections depending on the type of connector.

In the graphical editor, connection lines are drawn *over* components and other graphical objects. This means that connections are not hidden by mistake. However, by default routing is automatically applied to avoid connection lines over components. See below.

Warning!

Note that drawing an ordinary line between two connectors (see line drawing below) does *not* create a connection!

Routing of connections

Automatic routing when creating connections

The creation of a connection is by default performed in two steps. The first step is applying e.g. Manhattanize (see above). In the second step, automatic routing is performed. Note that if you have added intermediate connection points, the automatic routing is not performed.

The command **Undo** is applied to each step, this means that after having created a connection, if you undo once, you undo the routing, if you undo again, the connection is deleted.

You can control if automatic routing should be used by the setting **Automatic routing of connections**, reached by the command **Tools > Options**, the **Graphical Editor** tab. The setting is by default activated. The setting corresponds to the flag `Advanced.Editor.Routing.Automatic=true`.

Routing of present connections

If the setting **Automatic routing of connections** is activated (see previous section), routing is automatically applied to the effected connection lines if you move the corresponding components more than half a visible grid unit.

You can also use two commands for routing present connections. One possibility is to use the context command **Route Connection** for one or several selected connectors. Note that you can select several connections using **Shift+click** before right-clicking to display the context command.

The other possibility is to use the command **Graphics > Arrange > Route Connections**.

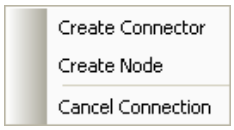
Notes:

- You can select a part of an open class by framing that part, and then applying the command to route all connections selected in that part.
- If you have nothing selected, the command works on all connections selectable in that class.

Avoiding crossing existing (unrelated) connections when routing connections

By default, when performing routing, crossing other (unrelated) connections is avoided, if possible. The feature is controlled by the flag `Advanced.Editor.Routing.AvoidCrossing`. The default value is `true`.

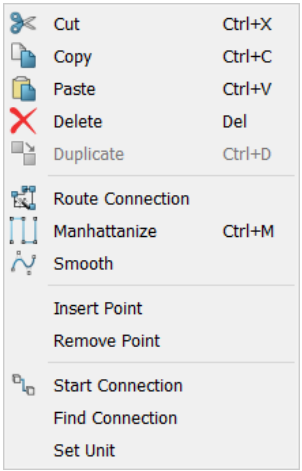
Context menu while connecting



Pressing the right mouse button or double-clicking the left mouse button while the connection is being drawn presents the menu in the figure to the left. For more information about the alternatives, please see “Context menu: Connection while connecting” on page 555.

Context menu for existing connections

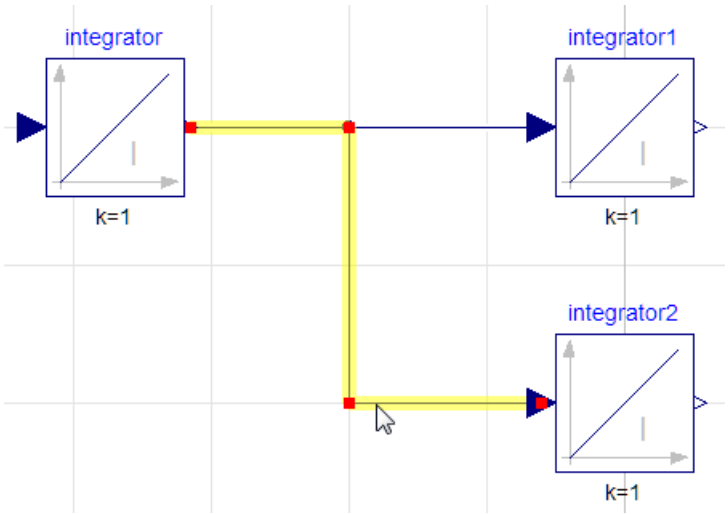
Right-clicking an existing connection will display the following context menu:



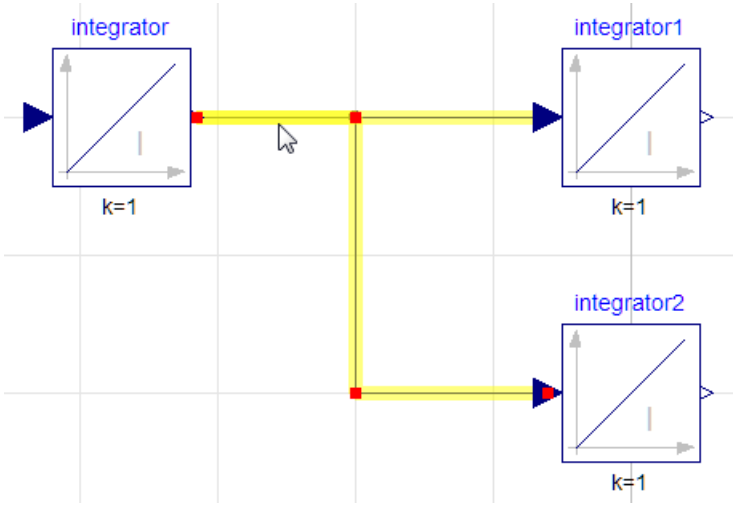
For details, see “Context menu: Connections” on page 554.

Highlighting connections

Connections can be highlighted by clicking them (if a specific option is activated):



If more than one connection exists where you click, all are highlighted:

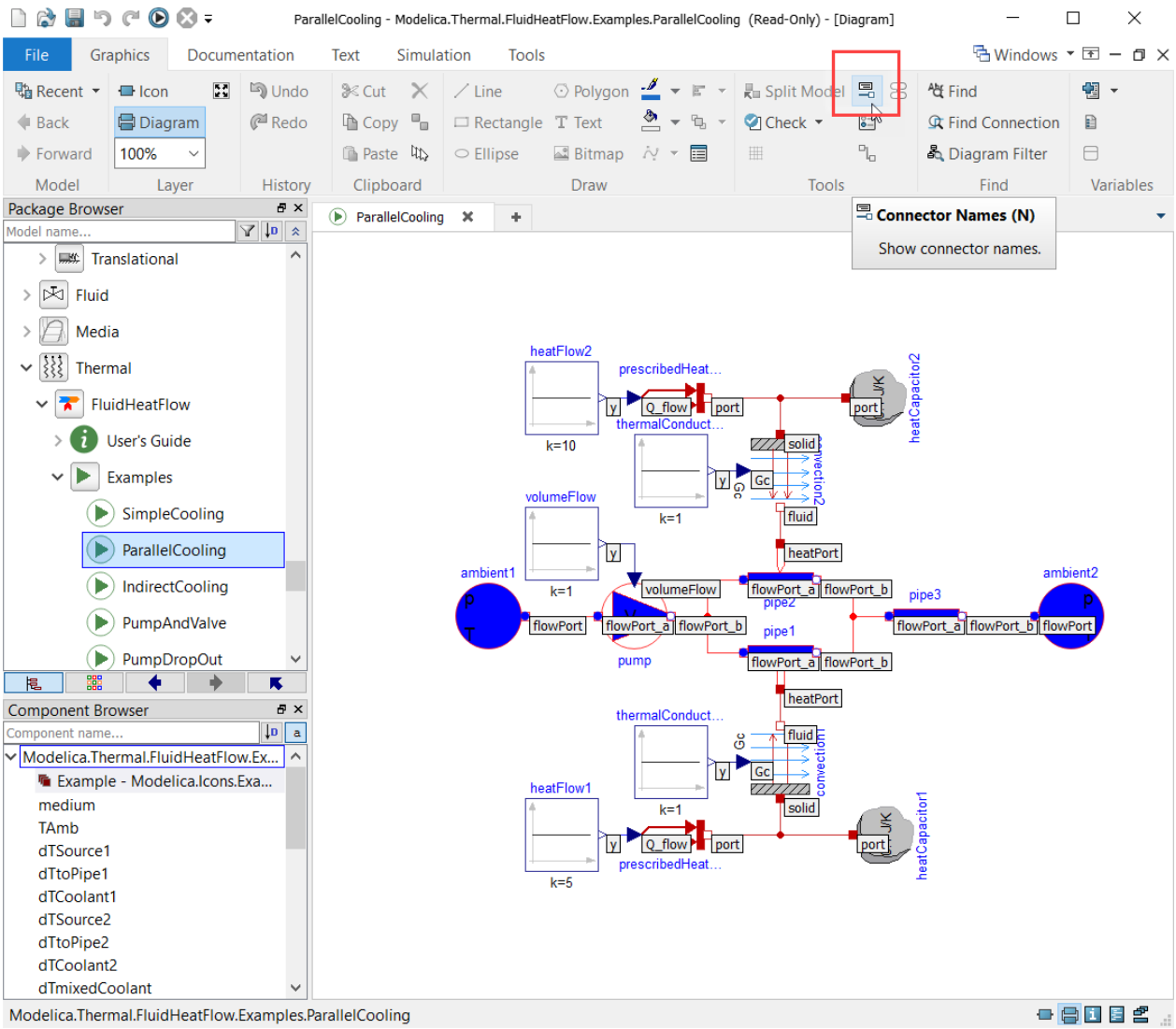


If overlapping connections are clicked, only the ones with a common port are highlighted.

The feature is disabled by default. To activate it, use the command **Tools > Options…**, select the **Graphical Editor** tab, and tick **Highlight connections when selected**.

Displaying connector names in the diagram

You can display the connector names in the diagram by the command **Graphics > Connector Names**:

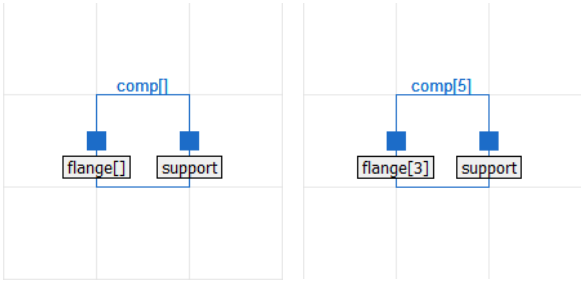


A corresponding flag `Advanced.Editor.ConnectorNamesDisplay` is available. The flag is by default false.

Note that array component connectors are shown with empty brackets by default. By setting the flag

```
Advanced.ArraySizeInName = true
```

the array size is also displayed in the brackets (compare the images below.)



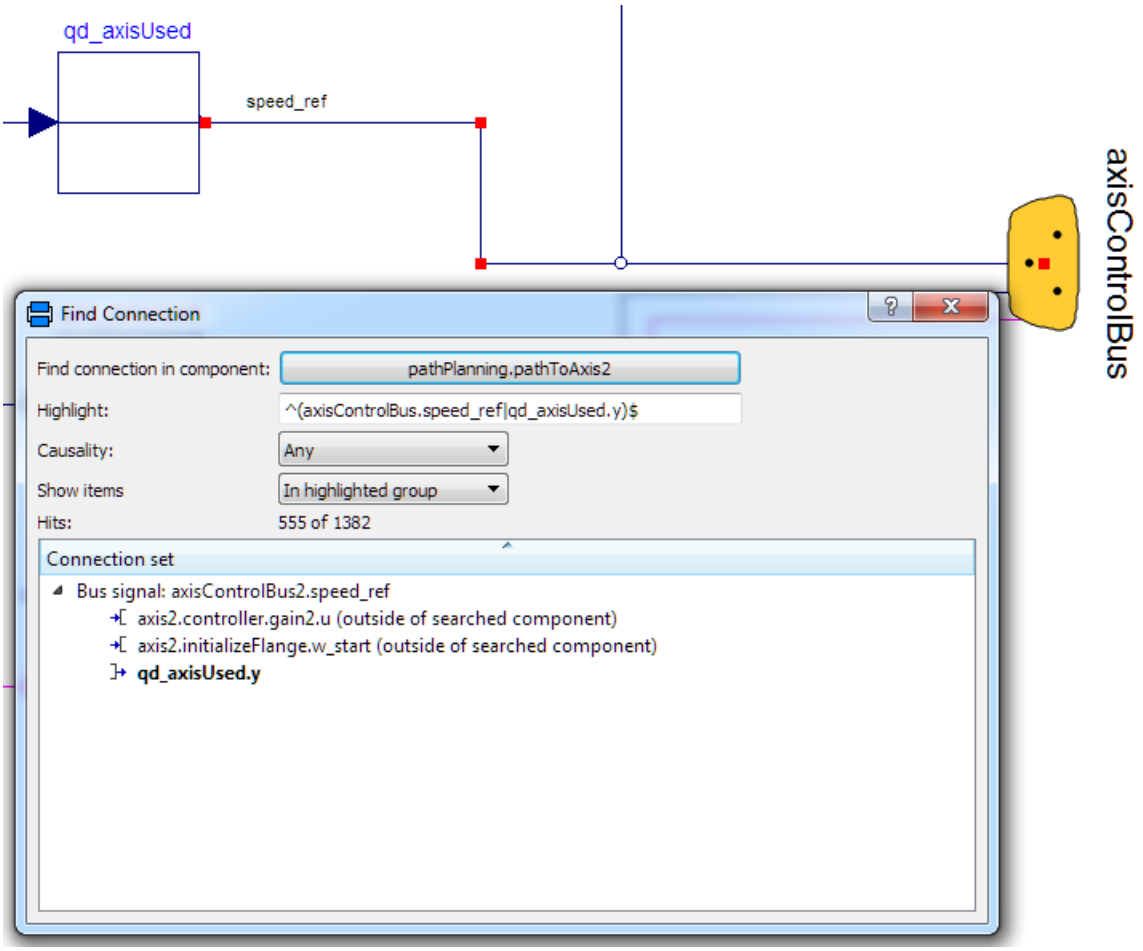
The flag is by default false.

Searching connections and connected components

In large models with expandable connectors, it might be hard to see where signals are created and used. To facilitate investigating such models, you can search connections, and the corresponding connected components, by the command **Find Connection** from the context menu of a connection in the diagram layer.

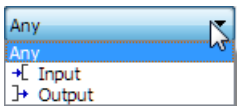
The search lists the connectors grouped by connector type.

An example of a search:

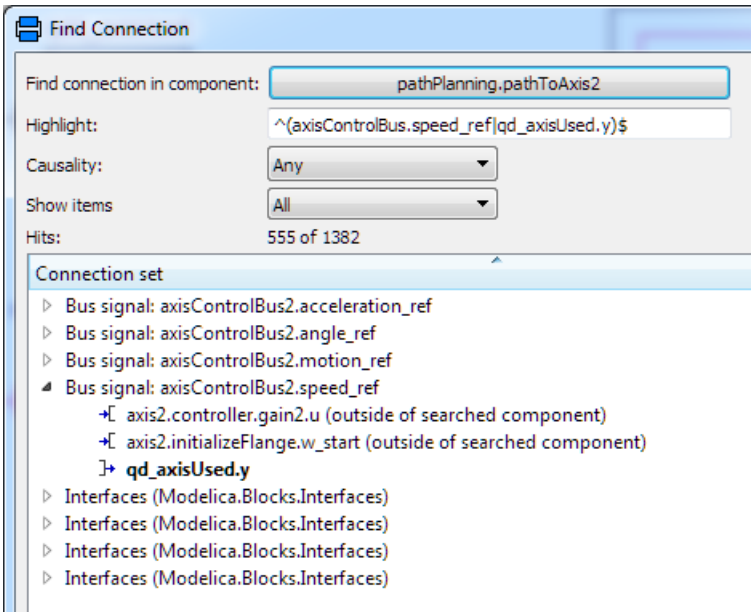


Some features of the command:

- The search is based on what is shown in the diagram layer when starting the search.
- What is entered in the **Highlight** field is searched for in the pane below, and hits are highlighted. Nodes are expanded to show the found occurrences.
- You can use the drop-down menu **Casuality** to refine your search by casuality:



- You can decide what is shown in the pane by the **Show items** selection. The alternatives are:
    - **All** displays all items, selecting this in the search above will give



  - **In highlighted group** displays only the variables in the groups where the hits are. This is the default selection (see first image in this section).
  - **Only highlighted** displays only the hits and the path to them.
  - **Filter instead of highlight** is similar to the above one, but filters instead of highlights.
- Clicking a result will select the corresponding component; if the component is inside another component than the one displayed, the **Show Component** command is used automatically to display the diagram of that component to select the searched component. This navigation does not influence the search; you can still select another component. You can go back to the original zooming by clicking the button after **Find connection in component**.

The command **Find Connection** is also always available on the **Graphics** tab, in the **Find** group.

Check of connections

When a connection has been defined, some checks are performed automatically.

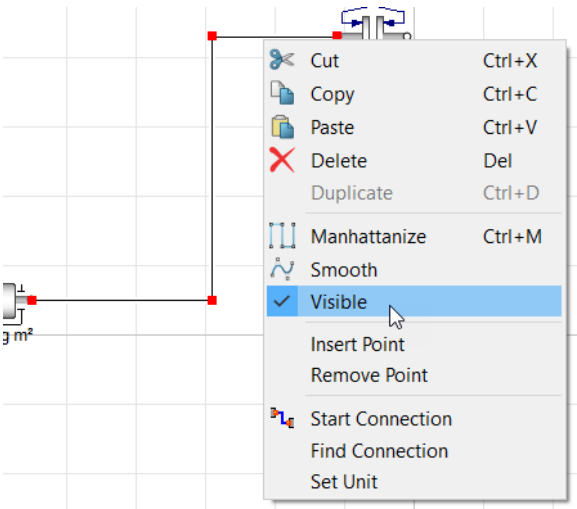
- Two connections between the same connectors are not allowed.
- The connectors at each end of the connection are checked for compatibility. The case of nested connectors is described below. If any other incompatibility is detected the error message displays the declarations of the two connectors. The connection can then be cancelled.
- If a model singularity is likely caused by incorrect/missing connections no additional diagnostics is given.

- Diagnostics for connection sets with multiple sources for the same causal signal, or no source for a causal signal. **Note**: In some models, the missing equation for the input is provided textually. Please change these models to use source-blocks instead.

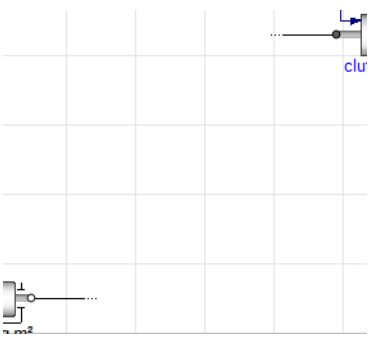
Hiding graphical connections

The user can easily use the context menu to hide graphical connections, improving the display of the diagram layer.

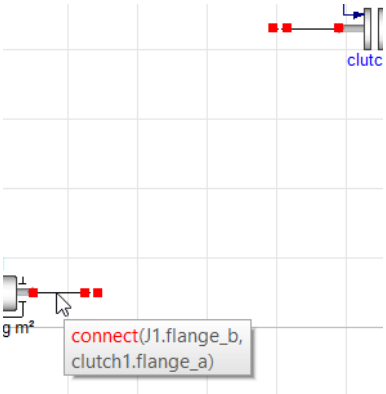
To use this feature, the user must set the flag `Advanced.InvisibleConnections=true`. The flag is by default false; the selection below is by default not shown in the context menu.



Unticking **Visible** will give the following:



Selecting one “stub” will also select the other:



Hidden graphical connections can easily be displayed by setting **Show hidden graphical objects** reached by the command **Tools > Options…**, in the **Graphical Editor** tab. See section "Graphical Editor options tab" on page 516. By default, such objects are hidden.

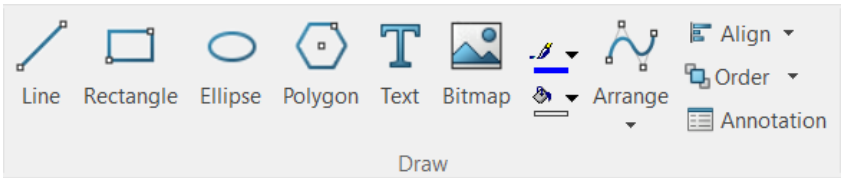
Accidental hiding is corrected using **Graphics > Undo** or **Ctrl+Z**.

4.2.5 Creating graphical objects

Graphical objects are created in the diagram layer of the edit window. They are drawn by clicking on certain buttons in the **Draw** group on the **Graphics** tab. The corresponding button is highlighed while the drawing operation is in progress.

The first six buttons are used for drawing graphical primitives. The next two buttons are for setting line and fill style attributes, with associated pull-down menus. The next button contains commands (seen by clicking the arrow) for e.g. rotating and flipping. **Align** and **Order** contains commands for aligning and ordering, and **Annotation** displays the annotation for a selected item.is used for toggling of the grid. The six first "drawing" buttons are treated in this section.

The Draw group on the Graphics tab.



The graphical tools creating graphical primitives described in this section (corresponding to the six first buttons above) work in essentially the same way:

- Select a tool by clicking on the appropriate button in the toolbar.
- Move the cursor to the "start" position in the edit window.
- Interact with the tool, for example, spanning a rectangle.
- After the interaction has been completed, Dymola returns to the default select/move/connect mode.

A common operation is to define two points, for example, the end points of a line segment or two opposite corners of a rectangle. This can be made in two different ways:

- Click the left mouse button at the start position, move the mouse to the end position, end click the left mouse button a second time.
- Press the left mouse button at the start position, move the cursor while pressing the left mouse button, and release the button at the end position.

The drawing operations snap the definition points to the grid of the model class. The details of the drawing operations are described below.

In editing the objects (see also later section); the context menu that is displayed by right-clicking on the border of the object is convenient when it comes to specific features or very common ones (e.g. cut/copy/paste). Note however that for operations like flip/rotate/align/order, the commands in the **Draw** group must be used.

Lines and polygons



Click on the **Line** or **Polygon** button and draw line segments. Interaction ends when the left mouse button has been clicked twice at the same position.

It is possible to add a point (corner) on a line or a polygon using the context menu and selecting the entry **Insert Point**. The point (corner) is inserted in the point closest to the cursor in the selected object. An existing point (corner) can be deleted by placing the cursor in the vicinity and use the entry **Remove Point** in the context menu.

Polygons are automatically closed by Dymola.

Skewed lines can be cleaned up with the context menu command **Manhattanize** which applies a manhattan-mode algorithm to the intermediate points of the line.

You can resize polygons by pressing **Shift** and drag one of the polygon points. This will lead to a uniform scaling of all polygon points relative the center of the polygon´s bounding box.

Lines are not connections!

Note that drawing a line between two connectors does *not* create a connection, although the difference may be impossible to see in the editor.

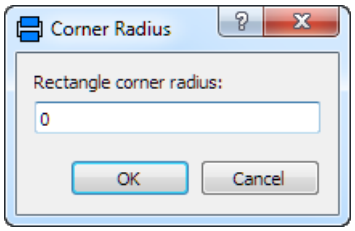
Rectangles



Click on the **Rectangle** button and draw the shape in the diagram. Holding down **Shift** when drawing the rectangle will keep the aspect ratio 1:1.

It is possible to edit a rectangle to give it rounded corners using the context menu command **Corner Radius…**. The following menu will appear:

Corner Radius menu.



Here the (maximum) radius of the circle that should form the corners can be specified (in grid points). If the height and width of the rectangle is larger than two times the specified radius, the specified radius will be used. If the height or width is lesser, the radius will be adapted to that.

Ellipses



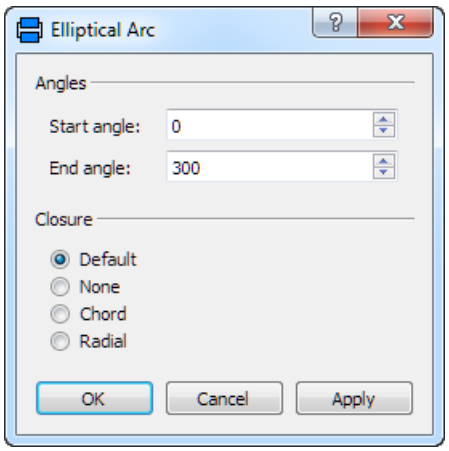
Click on the **Ellipse** button and draw the shape in the diagram. Ellipses are drawn to touch the bounding box at four points.

An ellipse could be made circular by keeping the **SHIFT** button pressed while drawing the object.

It is possible to create circle sectors/elliptic arcs. Do the following:

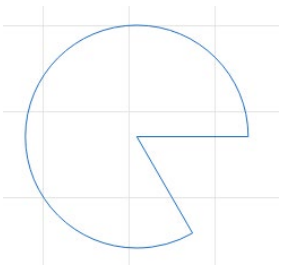
Draw a circle/ellipse. Color it. Select **Elliptical Arc…** from the context menu. The following menu will appear:

Edit angles menu.

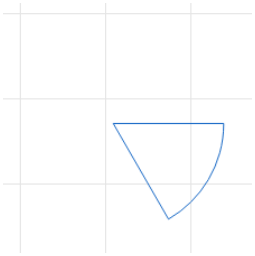


Changing the values to 0 – 300 for a circle will give:

Examples of circle segments.



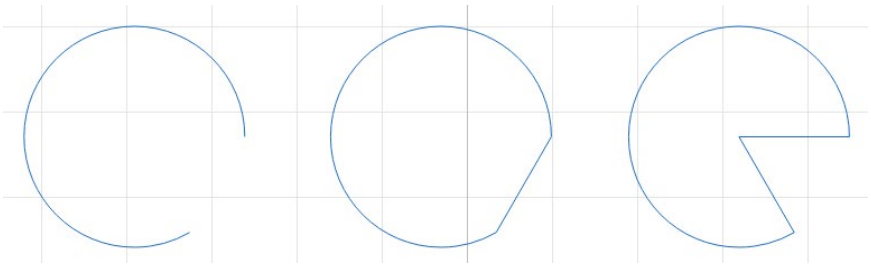
Changing the values to 300 – 360 will give:



The closure of the circle sector/elliptic arc can selected as any of the following when creating the circle sector using the menu above:

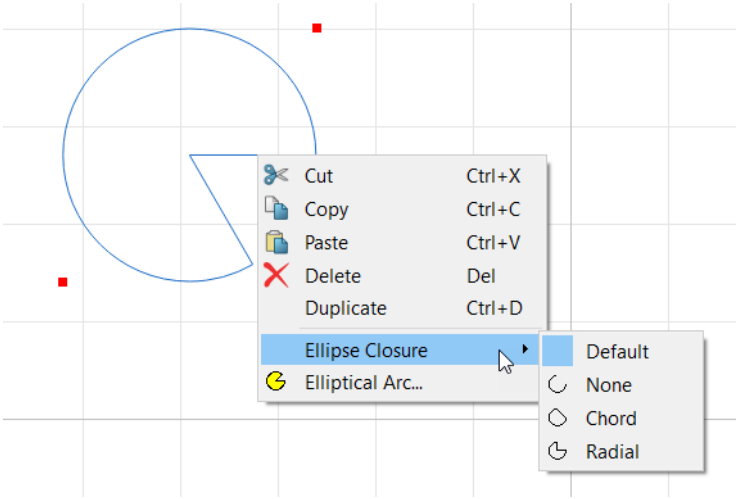
- Default
- None
- Chord
- Radial

Selecting **None**, **Chord** or **Radial** for a circle sector 0 – 300 will give the following:



Selecting **Default** means **Chord** if elliptic arc/circle sector is 0- 360, **Radial** otherwise. For more information about this, please see the Modelica Language Specification.

For a present elliptic arc/circle sector, it is possible to change the closure by right clicking and using the alternatives under **Ellipse Closure**:



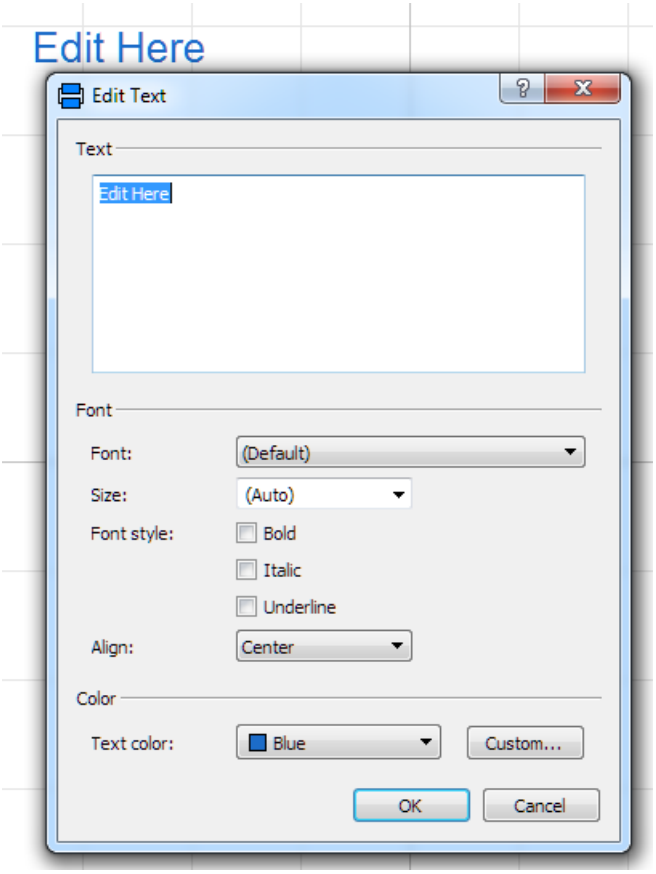
(You can also use the Line Style menu to change the ellipse closure.)

The start angle, end angle and closure are not remembered; when you create a new ellipse/circle you will have the default values again.

Text

Click on the **Text** button and draw the text object in the diagram, or double-click to create a "null-extent" text object. (Holding down **SHIFT** when drawing the text will keep the aspect ratio 1:1.) Dymola prompts for a new text string in a separate dialog window:

Text dialog.



This dialog is also displayed when double-clicking an existing text object.

The Edit Text menu

The selections shown here are the default selections.

Text – The text can be more than one line. A context menu is available in the editor.

Font – A number of fonts are available, including fonts for symbols etc. Fonts installed on the computer are supported. **Default** is the default font of Dymola; typically, Arial. It is possible to select **serif**, **sans-serif**, and **monospace** when selecting fonts (they are listed directly after the Default selection). The fonts used for these selections for Windows and Linux are

	Windows	Linux
Serif	Times New Roman	Times
Sans-serif	Arial	Helvetica
Monospace	Courier New	Courier

Size – A number of fixed sizes are available, ranging from 8 pt to 72 pt. **Auto** means that the text is chosen as big as possible bonding box; please also see below. It is also possible to enter any size by entering it in the input field and pressing Enter.

Align – **Centered** (default), **Left** and **Right** are available.

Font style – **Bold**, **Italic** and **Underline** can be selected.

Text color – The text color can be selected from a number of pre-defined colors; a custom color can also be defined. For more information, please see below.

Font size considerations

The text is by default centered in the bounding box; the font size of the text is by default chosen as **Auto**; that is, as big as possible without overflowing the bounding box. If the minimum font size option is set (see the **Tools > Options** command, section "Text Editor options tab" starting on page 518, the setting **Restrict minimum font size**), the text may be truncated (indicated with ellipses ...) and a small bounding box may overflow vertically. If space is extremely limited the text is not drawn at all. However, a "null-extent" text is always shown if the minimize font size option is set.

Tokens expanded in text strings

These tokens are expanded in a text string:

`%class` Name of the enclosing class.

`%name` Name of the component this text is enclosed in.

`%path` The full path name of the component.

`%par` The value of parameter named par in the model.

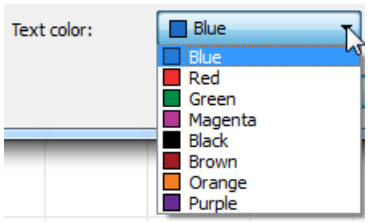
`%=par` The value of parameter named par in the model, printed as par=value.

Strings in the diagram layer displaying values can contain periods: `%recordName.Component`, for example to display record component values. An example of such a text string could be `%myRecord.a`.

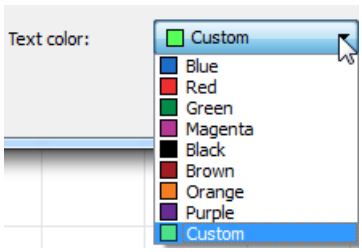
Note that also curly brackets are allowed when specifying tokens, for example, `%{name}`. This increases clarity, and also makes it possible to avoid blanks, for example, to use `%{MyRecord.a}m` to display the record component a with unit m.

Color selection

The following colors are available by default:



Custom color can be added for the selected signal by a color palette displayed by selecting the Custom… button. That color (see also below) will then appear last in the color dropdown list:



The window for defining/selecting colors is the standard one in Dymola. For more on color selection, please see section “Graphics > Line Style” starting on page 471.

“Null-extent” text objects

Selecting the text button and then double-clicking in the wanted location of the text will create a “null-extent” text object (both coordinates of the text object extent are the same). The text will be centered on the specific point.

The font size will be Auto (see above). Two handles will be present, on top of each other. By dragging the top one, that handle will move, changing the object from a “null-extent” one.

The text can be selected by clicking/double-clicking on the center of it (where the two handles are located).

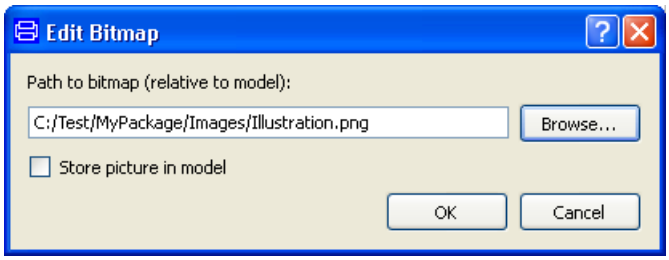
Bitmap



Draws a bitmap which is read from an external file. Click on the **Bitmap** button, draw the outline in the diagram and then specify the bitmap filename. Supported file formats are BMP, GIF, JPEG, PNG, and SVG. The bitmap is scaled preserving aspect ratio, centered in the bounding box. Holding down **SHIFT** when drawing the bitmap will keep the aspect ratio 1:1. Incorrect filename displays a default bitmap (same as toolbar button).

Please note that the menu for specifying the file also contains a possibility to store the picture in the model!

Open bitmap menu.



This menu pops up when creating the bitmap image, but also when later double-clicking on the bitmap.

The starting point of the browser is the folder where the relevant top package resides. It is recommended to store images that should be used in Dymola documentation in a folder "Images" located in the same folder as the relevant top package. (It will then be intuitive to move that folder also when moving the package, which will preserve the image references.)

When browsing for the image the file path is shown in the menu. When clicking **OK**, the image will be inserted.

The corresponding result will be an insertion of an annotation in the code. In that annotation the Modelica URI (Uniform Resource Identifier) 'modelica://' scheme is used by default when creating the path to the image.

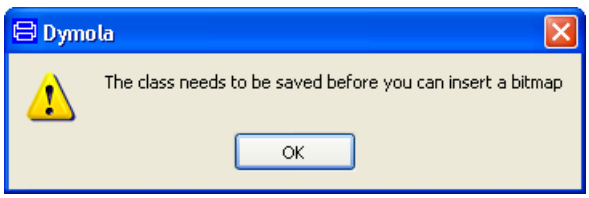
Using the recommended location for images above for an image "Illustration.png" for a top package "MyPackage", the URI

```
modelica://MyPackage/Images/Illustration.png
```

will be the resulting URI in the annotation. This annotation can be seen by selecting the image, and then, in the **Draw** group of the **Graphics** tab, selecting **Annotation**.

(This URI can also be used in all sub-packages of MyPackage to refer to the image.)

In order to be able to generate correct links, the model/package has to be saved (using e.g. **File > Save**) before an image can be inserted. If initial saving has not been done, a warning will be displayed:



The bitmap annotation also has the Dymola-specific attributes:

- `__Dymola_stretch=false` The bitmap is not scaled to fit the bounding box. The original (pixel) size is preserved.

- `__Dymola_preserveAspectRatio=false` The bitmap is scaled to fill the bounding box, possibly distorting the image.

The default values are “true” in both cases.

Toggling the grid



The button **Toggle Grid** in the **Tools** group on the **Graphics** tab is used to decide whether the grid lines should be shown in the diagram layer of the edit window or not. Default is showing the grid. In read-only models the grid cannot be shown.

Default graphics

Classes without graphical information are automatically given a simple default layout when displayed by Dymola. The purpose of the default graphics is to provide some meaningful graphics while the model is initially edited.

- Components and connectors without graphics are represented by a rectangle and a text with the name of the component at the center.

The default layout can easily be changed by the user.

Representation of origin

The origin is shown by a red cross when any operation working on the origin has been applied, e.g. a rotation or a flip.

4.2.6 Changing graphical attributes

There are a number of ways to change the graphical attributes of graphical objects. The basic idea is to use the context menu that is displayed by right-clicking on the border of the object for simple operations like cut/copy/paste, and operations specific to the object type, e.g. corner radius of a rectangle. However, options like rotating, flipping, ordering, color and filling must be changed using separate commands in the **Draw** group on the **Graphics** tab.

Please also note that editing using **Annotation** works in a specific way. See below. This menu is by default displayed for most graphical object when double-clicking the border.

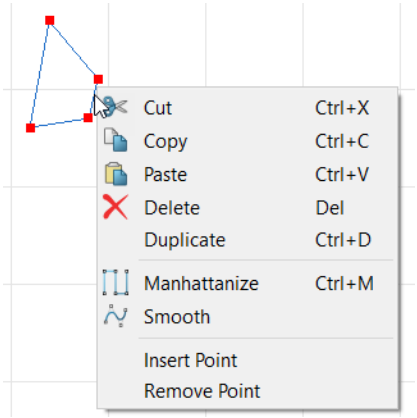
Menus used for editing

Context menu

A context menu is available by right-clicking on the border of a graphical object (or by right-clicking when the object is selected). More information about this menu is available in section “Context menu: Graphical objects” on page 555.

The context menu is adapted for each graphical object, with only relevant entries being displayed. As an example, the context menu for a polygon looks like:

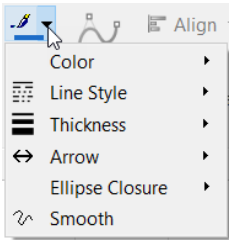
Context menu: Example for polygon.



Please note that the context menu includes specific options (e.g. corner radius in the context menu for a rectangle and elliptical arc for the context menu of an ellipse).

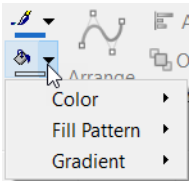
The command **Annotation** makes it possible to edit graphical annotations by a menu, in a specific way. Please see section "Working with annotations using the Annotations dialog" starting on page 426.

Line style and Fill style



The user must first select one or more primitive graphical objects, and then pick one of the style buttons in the **Draw** group on the **Graphics** tab to change the corresponding visual attributes of the selection. Note that line color and fill area color are different attributes (a rectangle has both, for example), and have different buttons in the toolbar. Text color is set as line color.

Selecting an attribute from the menu sets that attribute for the selected objects (when applicable), and also marks it as the default. Clicking on the button sets all line (or fill) style attributes for the selected objects.



The line style and fill style alternatives are shown in the images to the left. More information about the alternatives is available in the sections "Graphics > Line Style" starting on page 471, and "Graphics > Fill style" starting on page 473.

4.2.7 Programming in Modelica

Introduction

Programming in Modelica language is done in the Modelica Text layer of an edit window. By default, it contains the Modelica text for declarations of non-graphical components of the class e.g. constants, parameters, variables and equations of a model.

Modelica text such as declarations, equations and algorithms can be edited.

The *starting* point working with Dymola for many users is seldom to write code in Modelica Text layer; the natural starting point is to use ready-made components in libraries and put them together graphically in the diagram layer. Doing this will automatically create the corresponding code in the Modelica Text layer.

However, some users (e.g. those building libraries for others) will soon be working in the Modelica Text layer. This is where any user-defined code has to be inserted.

The chapter "Getting started with Dymola" illustrates the above; the simple pendulum example illustrates starting to work in the Modelica Text layer, while the motor drive example illustrates starting to work with pre-defined components in the diagram layer.

Please note that it is possible to show the Modelica Text layer with mathematical notation; that is, with formulas etc presented as formulas and not as plain text.

The Modelica language

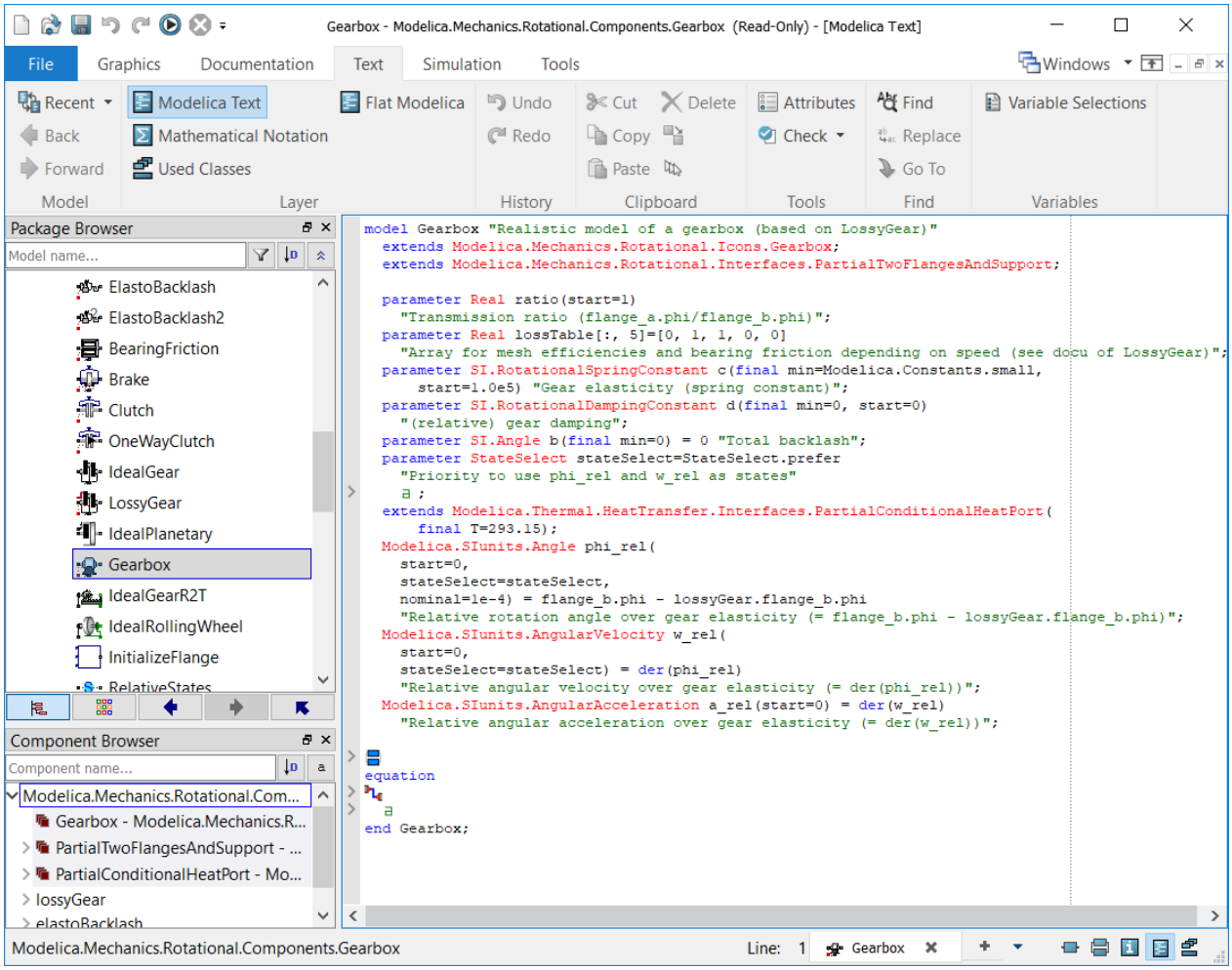
Modelica is an open, object-oriented language for modeling of large, complex and heterogeneous physical systems. For more information about the language, please see Modelica Language Specification, available on the **Tools** tab, in the **Help** group, clicking **Help Documentation**. More information is also available at www.modelica.org.

Looking at the Modelica Text layer



If the code for a certain class should be looked at (or edited), select that class by double-clicking on it in the package browser, then, on the **Text** tab, in the **Layer** group, select **Modelica**. That will open the Modelica Text layer for that class. (**Modelica** is by default selected when entering the **Text** tab.)

Doing this for Modelica.Mechanics.Rotational.Components.Gearbox reveals the following:



The content of the Modelica Text layer differs somewhat depending on what class is shown; when creating a new class Dymola supports creating a model, connector, record, block, function or package as separate entity. Each of these has its own characteristics.

Class information

The first line in the Modelica Text layer states what class is presented, the name of the class and (an optional) description. The last line in the editor concludes this first line.

The possible classes that are presented as separate units are: Model, connector, record, block, function and package.

Parameters, variables and constants

Variables can be of variability type parameter, variable, constant or discrete. They are presented by variability type, full type name, name, start values, comment etc. Annotations might be present, more about this later.

Components, connections and annotations

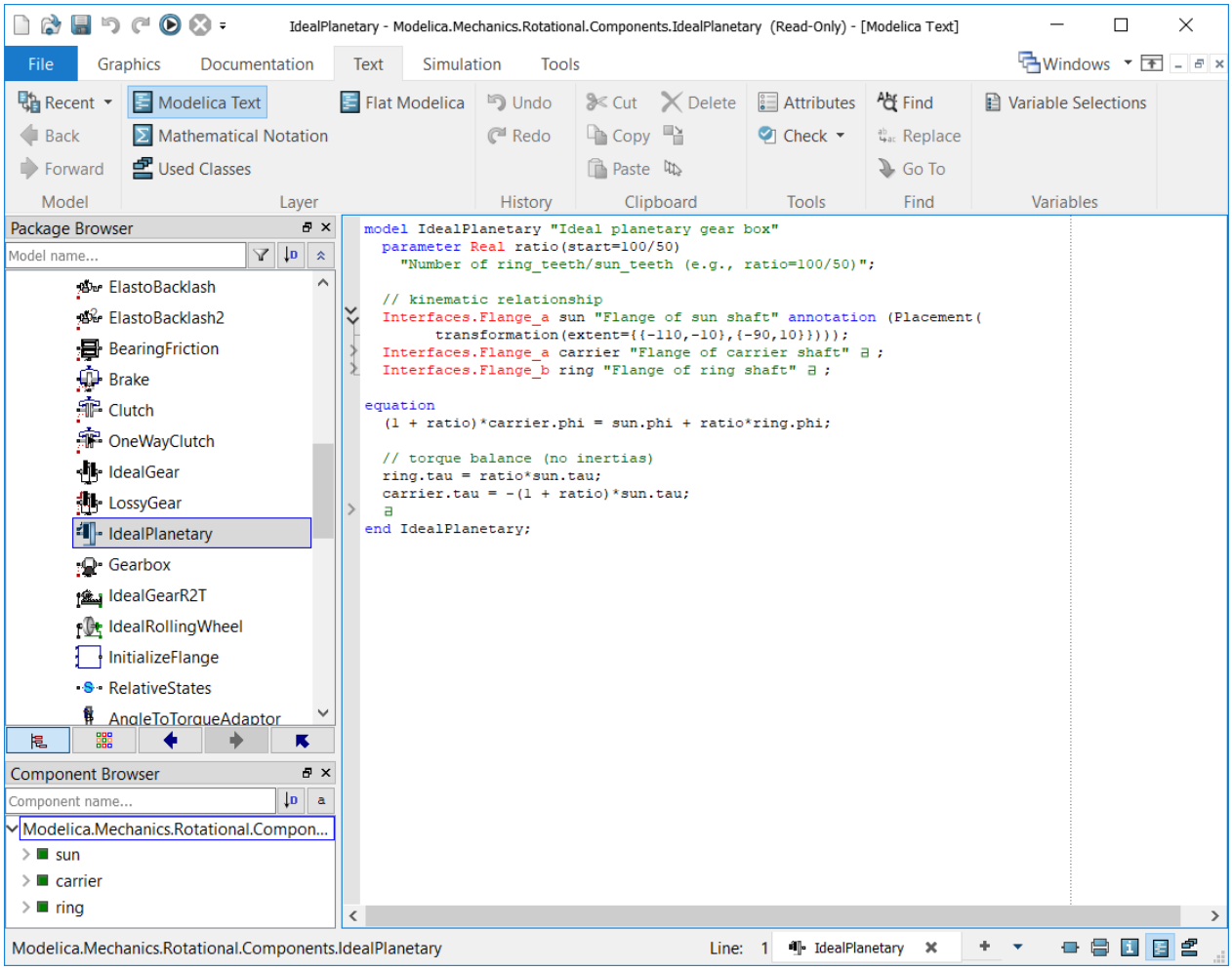


Graphical components and connections in a model are by default shown as symbols inserted in the text (the left one indicates components and the one to the right connections).

Annotations are definitions of graphics, documentation, menu layout and other attributes that does not in any way influence the simulation result. Annotations can be annotations of the class (top level) or annotations of components (variables etc.) in the class. Annotations of the class is by default represented by a free-standing inserted in the text, annotations of a component is by default represented by an in the end of the line of that component.

Showing components, connections and annotations.

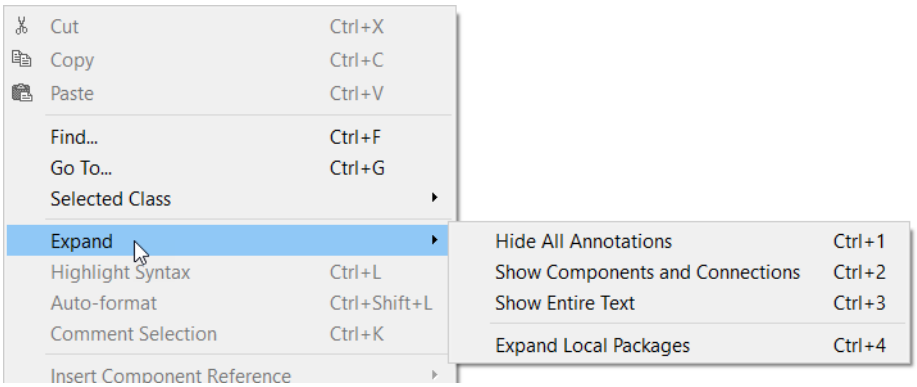
Besides being presented as the symbols above, components, connections and annotations are also indicated by expand icons in the sidebar to the left of the code. If expanded, a collapse icon with a corresponding line indicating the extension of the expansion will appear.



In the example above the Component annotation is expanded, revealing the flanges. The annotation of Flange_a is in turn expanded, revealing the placement annotation for that flange.

Expanding can either be done individually for each annotation, by clicking on the expand icon ▷ (or the corresponding symbol) or collectively for all annotations by right-clicking and using the context menu.

Expand possibilities in the context menu.



Hide all annotations will only display the symbols for components, connections and annotations.

Show components and connect will display the components and connections, but still only symbols for annotations (annotations will not be expanded).

Show entire text will display all information in textual format (all possible expansion will be done).

Collapsing can either be done individually for each expansion by clicking on the corresponding collapse icon ◢, or collectively using any of the context menu commands mentioned.

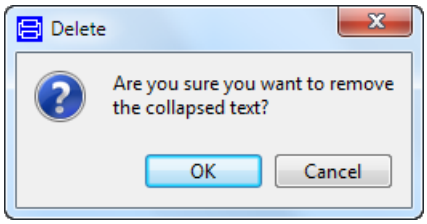
Individual expansions are remembered; if e.g. an annotation that contains an expanded annotation inside is collapsed, the expanded connection inside is showed again when the "primary" annotation is expanded again.

Editing components, connections and annotations.

Components and connections are usually the result of actions in the diagram layer (dragging in components from the package browser and connecting them). Usually no editing is done.

The editing of annotations depends on type of annotation. All annotations can be edited in the Modelica Text layer. Top-level annotations and annotations related to graphical representation (graphical primitives, components, and connections) can also be edited using the **Annotations** dialog (available by, being on the **Graphics** tab, selecting the object, and, in the **Tools** group, clicking **Annotation**). Annotations of variables and parameters not related to graphics can be edited using the variable declaration dialog (see above, about editing variables).

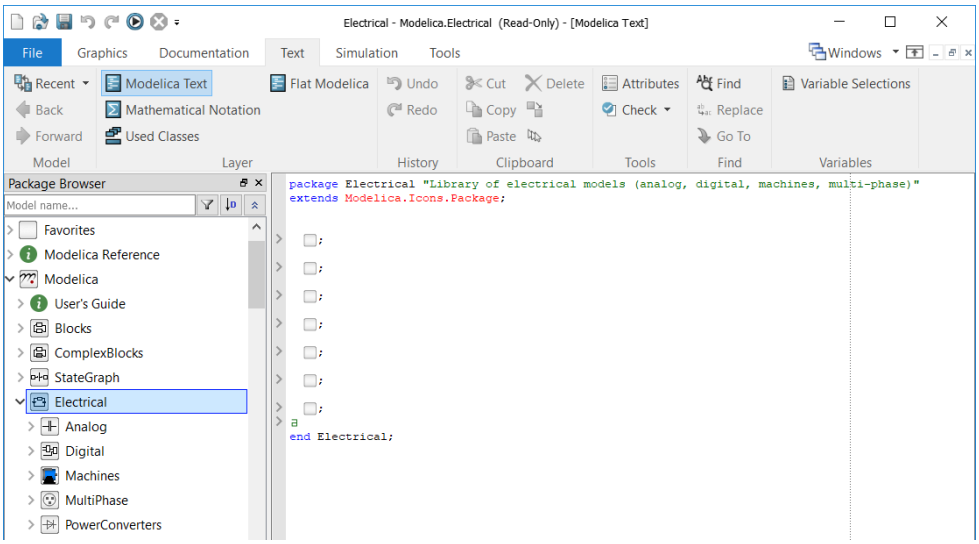
Code containing collapsed item (component, connection, annotation or local package) can be selected and deleted; a question will be displayed whether to do it:



Local packages

Local packages are packages defined in another package. In the figure below, the six package symbols are local packages; they correspond to the six packages under Electrical in the package browser:

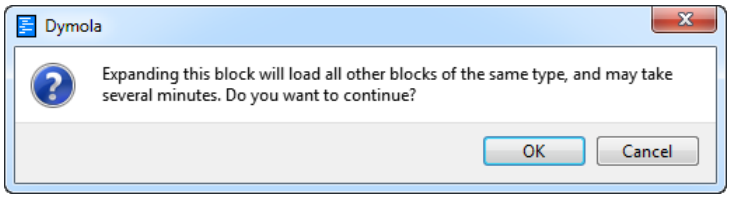
Local packages.



Local packages are by default only shown as symbols the Modelica Text layer. They can be expanded in two ways.

1. Right-click in the Modelica Text layer and select the command **Expand > Load Local Packages**. All local packages will be loaded, including any local packages deeper in the tree structure Then each package can be individually expanded by clicking on the corresponding package symbol or the corresponding expand icon ▷.
2. Click on the icon of a package. The following warning will be given:

Expansion message.



By selecting **OK**, all local packages will be loaded, and the selected package will be expanded.

If the total text to be handled is larger than 2 MByte, all local packages will be expanded, the sidebar will disappear and dynamic color coding will not work.

Comments and descriptions

Lines starting with // are comments, comments inside expressions etc. are marked by /* */. Comments can also be present in the end of line, then they will start by //. Comments will always be marked by green color.

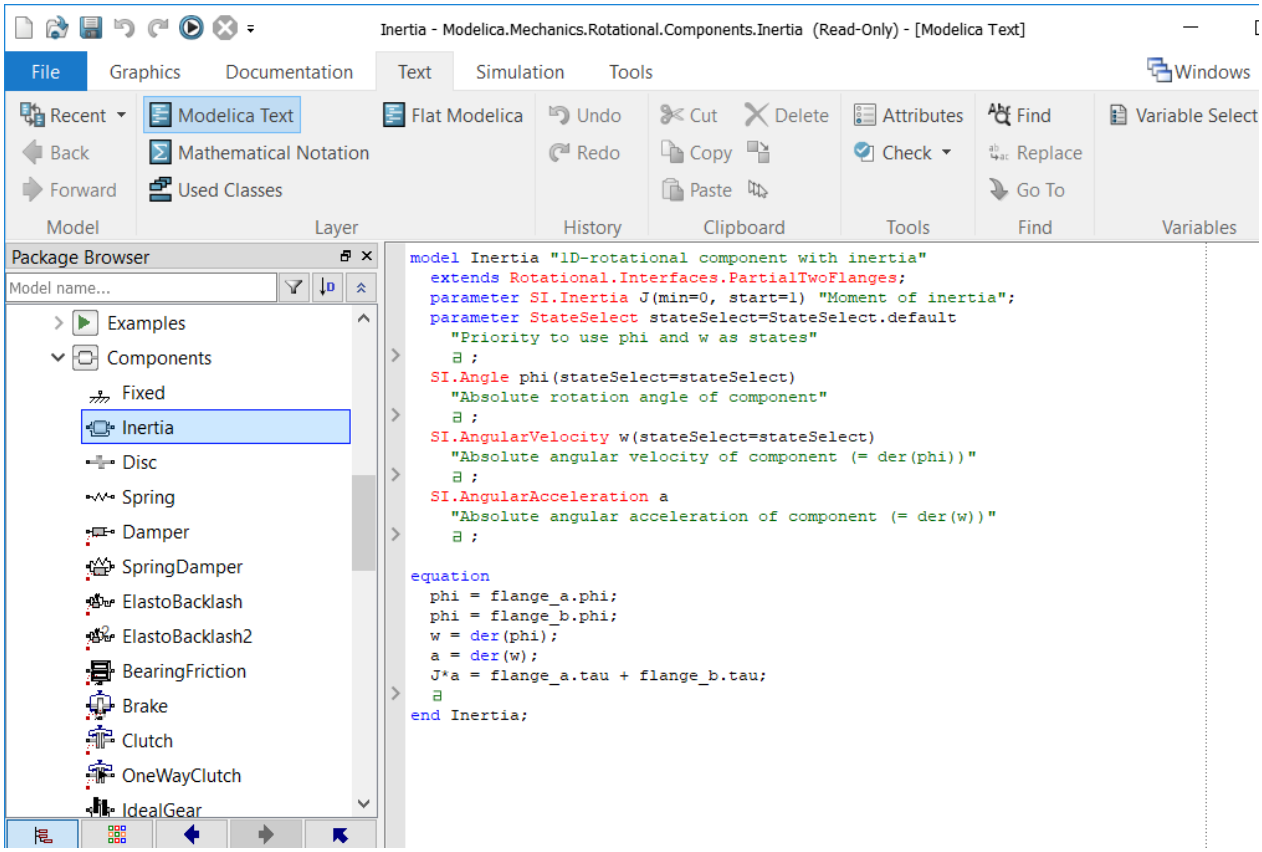
Navigation in the Modelica Text layer

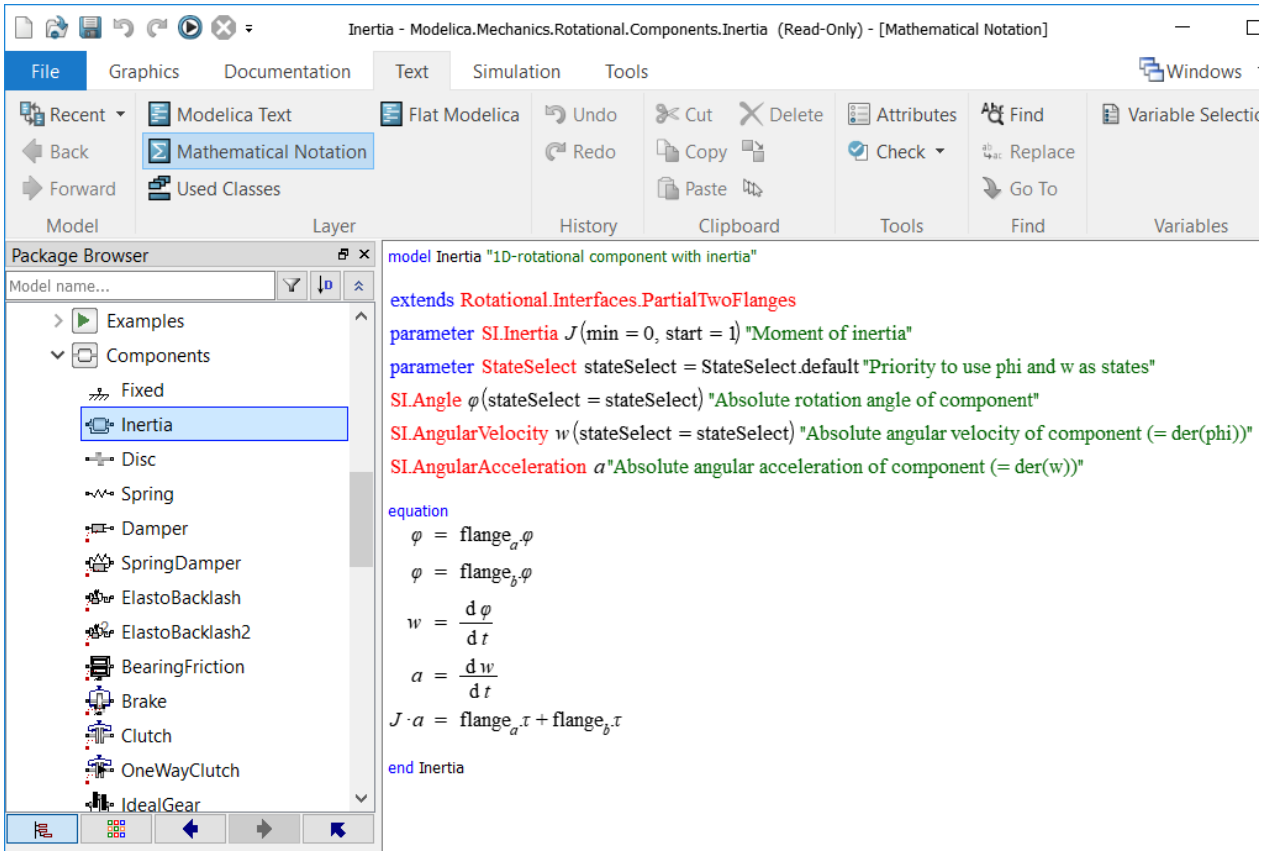
It is possible to select a class in the text, and then open that class in present window or a new window. It is also possible to look at the documentation of that class. Please see the corresponding section below.

Displaying the Modelica Text layer with mathematical notation

It is possible to display the Modelica Text layer with mathematical notation, showing formulas as formulas and not as plain text.

The two figures below will illustrate, the first is without mathematical notation, the second with mathematical notation.


Inertia - Modelica.Mechanics.Rotational.Components.Inertia (Read-Only) - [Modelica Text]
File
Graphics
Documentation
Text
Simulation
Tools
Windows
Recent
Back
Forward
Model
Modelica Text
Mathematical Notation
Used Classes
Layer
Flat Modelica
Undo
Redo
History
Cut
Copy
Paste
Delete
Clipboard
Attributes
Check
Tools
Find
Replace
Go To
Find
Variable Select
Variables
Package Browser
Model name...
Examples
Components
Fixed
Inertia
Disc
Spring
Damper
SpringDamper
ElastoBacklash
ElastoBacklash2
BearingFriction
Brake
Clutch
OneWayClutch
IdealGear
model Inertia "1D-rotational component with inertia"
extends Rotational.Interfaces.PartialTwoFlanges;
parameter SI.Inertia J(min=0, start=1) "Moment of inertia";
parameter StateSelect stateSelect=StateSelect.default
"Priority to use phi and w as states"
;
SI.Angle phi(stateSelect=stateSelect)
"Absolute rotation angle of component"
;
SI.AngularVelocity w(stateSelect=stateSelect)
"Absolute angular velocity of component (= der(phi))"
;
SI.AngularAcceleration a
"Absolute angular acceleration of component (= der(w))"
;
equation
phi = flange_a.phi;
phi = flange_b.phi;
w = der(phi);
a = der(w);
J*a = flange_a.tau + flange_b.tau;
end Inertia;



The mathematical notation mode is selected by, on the **Text** tab, in the **Layer** group, clicking **Mathematical Notation** (see above image).

How the mathematical notation displays different items is described in the next chapter, section “Model simulation”, sub-section “Documentation”.

Please note that there are some limitations with the mathematical notation mode!

- Mathematical notation is only a display of the Modelica text editor; there is no edit possibility while in this mode.
- The expansion bar is not accessible in this mode.
- Only items expanded when changing to the mathematical notation mode is shown; the user have to expand what should be shown before entering the mathematical notation mode.
- Comments will not be shown in this mode.

Copying is possible. As an example, copying the whole content of a small model and inserting it into Microsoft Word will result in the pure text parts (e. g. parameter declarations) being inserted as text, while the equation part will be inserted as a picture.

More about copying and other selections available in the context menu of the editor is described in the section "Context menu: Edit window (Modelica Text layer – mathematical notation)" starting on page 540.

Features and tools of the editor

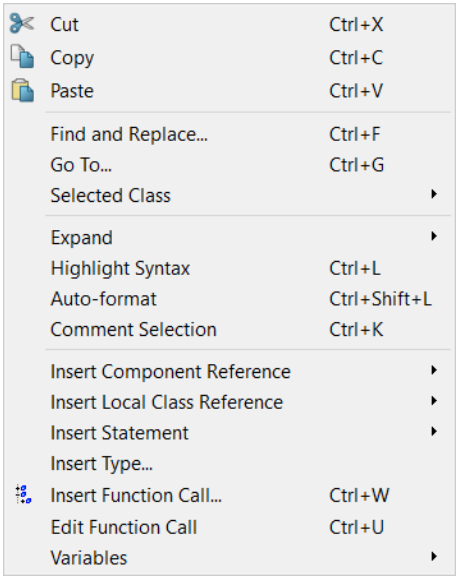
The features of the Modelica text editor can be summarized by the following table, which also contains information about where to find the description of each feature, respectively.

Feature	Where to find information
Undo. Redo	This section (context menu) and reference section
Cut, Copy, Paste, Select All	This section (context menu) and reference section
Find, Go to line	This section (context menu) and reference section
Expanding/collapsing of components, connections and annotations	Previous section
Loading/expansion of local packages	Previous section
Automatic and manual color coding	This section
Code completion	This section
Tooltip displaying the short description for functions	This section
Automatic indenting, adjustable maximum line length, and other text formatting	This section
Font size	This section
Bracket handling	This section
Comment out selected rows	This section
Navigation (classes, info)	This section
Syntax error checking	This section
Printing (all or selected section)	This section
Defining variables using the context menu	Later section (Editing of certain components in Modelica Text layer)
Defining variables using drag and drop	Later section (Editing of certain components in Modelica Text layer)
Using Insert Type to define the type of a variable	Later section (Editing of certain components in Modelica Text layer)
Inserting statements by menu	Later section (Editing of certain components in Modelica Text layer)
Handling of function calls	Later section (Editing of certain components in Modelica Text layer)
Inserting component references and local class references	Later section (Editing of certain components in Modelica Text layer)
Showing mathematical notation	Previous section

The context menu

Right-clicking presents a context menu with common text operations. Most of these operations are also available in the **Edit** menu.

Context menu of the Modelica Text layer.



The three first sections are not treated here, for more information on those please see the reference part “Context menu: Edit window (Modelica Text layer)” on page 533.

The last two sections will be treated here, in following sections.

Automatic syntax highlighting

The color codes of the syntax highlighting are:

blue	keywords
red	types, operators etc.
black	values, variables, parameters etc.
green	comments

New text will be syntax highlighted as you type, except types (e.g. Real). To get also types color coded, you can do any of the following:

- Right-click and select **Highlight Syntax** from the context menu (or use the shortcut **Ctrl+L**). The command will also perform a syntax check (see below).
- In the **Tools** group, click on the **Check** button (or press **F8**). This main function is of course checking, but it will also color-code text.

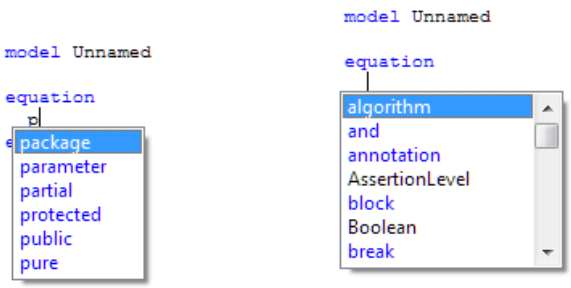
The above commands will color-code text, but not reformat it. For formatting, please see below.

HTML text in annotation will have automatic color-coding of tags as well. Please see section “Documentation” below.

Code completion

Code completion can be activated by pressing **Ctrl+Space** in the Modelica Text Layer. (It is also available in parameter input fields and in the script editor.) Pressing **Ctrl+Space** will bring up a context menu containing all available contextual words beginning with the letters written so far or all contextual words available if nothing has been written yet. As you type, the list of possible choices will decrease until only one word remains. If something not matching the words in the completion list is typed, the menu will automatically be closed.

Code completion examples.



The code completion is contextual, that means that the menu contains:

- All Modelica keywords (highlighted in blue).
- All variable names defined in the class.
- Names of open packages.
- Names of other classes available at the code completion level (models, functions etc).

The code completion also works for Modelica texts that have syntactic errors. Thus you can add a declaration of a new component and the code completion will give you its subcomponents, even if the declaration of the component is not yet syntactically correct.

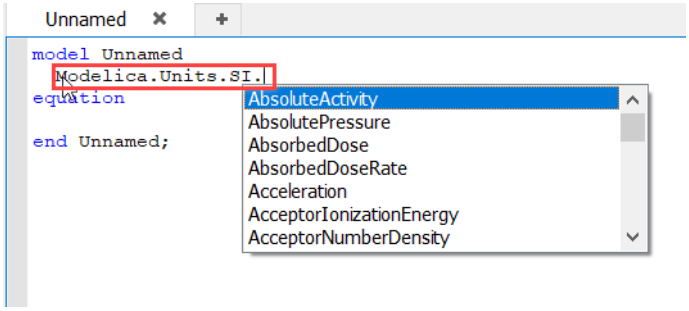
The code completion supports lookup in Modelica classes. When completing e.g. `Modelica.Math`, the menu will only show functions within that library.

To select the word to input and enter it, do any of the following:

- Click on the wanted word in the menu.
- Navigate to the wanted word and then press **Enter**. You can navigate by the arrow keys, but you can navigate to the next alternative also by clicking **Ctrl+Space** again, or keeping **Ctrl** pressed and clicking **Space** again.

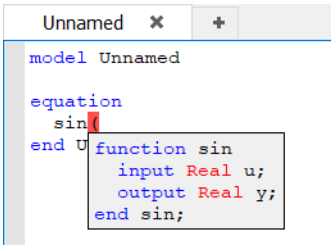
Closing the menu without inputting any word can be done by pressing **Escape** or clicking outside of the menu.

Note. There is a specific code completion keyboard shortcut for adding Modelica SI units. To add such a unit, you can click **Ctrl+Shift+U**. The shortcut adds the text `Modelica.Units.SI.` and also opens the meny for code completion, listing all Modelica SI units:



Tooltip displaying the short description for functions in the Modelica text editor

You can display a tooltip containing the short description (signature) for a function in the Modelica text editor. Type the function name (or use the code completion feature above) and then press left bracket `(` to display the short description. As an example, for the `sin` function:



You can also press **Ctrl+Space** after the left bracket to display the function description.

You can hide the function description by pressing right bracket `)`, **Escape**, **Enter**, **Up**, or **Down**.

This feature is also available in the script editor.

Automatic indenting

Indenting is automatically handled while typing. The following figure illustrates the result of typing in a model without any manual indenting at all:

Example of automatic indenting.

```
model TestIndentation
  Real r;
  parameter Boolean b=false;
equation
  when initial() then
    if b then
      reinit(r,1);
    else
      reinit(r,-1);
    end if;
  end when;
  der(r)=-r;
end TestIndentation;
```

Manual indenting

Tab for indenting and **Shift+Tab** for decrease of indenting can be used. Note that also description strings and class names can be indented.

Adjustable maximum line length

A dotted vertical line is present in the editor to indicate the maximum line length. It can be moved, by pressing **Shift** and dragging, to change the maximum line length. The user can then adapt to the new value by right-clicking and selecting the command **Highlight Syntax**, or by the corresponding short command **Ctrl+L**. This dotted vertical line is local for each Modelica Text editor window.

The value corresponds to the setting **Max line length** reached by the command **Tools > Options…**, the **Text Editor** tab. If you press **Shift** and drag the dotted line in the Modelica text editor, the value of **Max line length** is updated with the new value. If you change the **Max line length** setting, the dotted line in the editor is automatically updated to that value. See also the section “Text Editor options tab” starting on page 518.

Other text formatting

Apart from the features mentioned above, the user can select any part of the text, right-click and select **Auto-format** in the context menu (or press **Ctrl+Shift+L**). This action will autoformat that section in pretty print format.

The formatting of declarations, equations, etc is kept by default. The formatting is however not kept for:

- Multiple statements or declarations on one line.
- Indentation of end tags (such as `], ), end <class>`).
- Indentation of separators (such as `,, ;, then, else`).

- Multiple consecutive empty lines, if seen as not intended by the user, e.g. multiple empty lines created when removing variables via GUI. Other multiple consecutive empty lines are kept.
- Comments inside declaration of components, or type prefixes of classes (e.g. `inner /* */ class a end a;`)
- Comments inside expressions.
- Equal sign for modifiers.
- Minor errors that are automatically corrected (e.g. incorrect use of = vs. :=).

Font size

If the text seems too small to work with, the font size can be changed using the **Tools > Options** command and temporary changing the **Base font size** in the **General** tab. It is a good idea to set it back afterwards.

Bracket handling

When an excess bracket is inserted, it is marked by red (figure to the left below). When the corresponding enclosing bracket is added, both brackets will be marked with green (figure to the right below).

Bracket handling.

```
model Pendulum
  parameter Real m=1;
  parameter Real L=1;
  parameter Real g=9.81;
  Real phi(
equation

end Pendulum;
```

```
model Pendulum
  parameter Real m=1;
  parameter Real L=1;
  parameter Real g=9.81;
  Real phi(start=0.1)
equation

end Pendulum;
```

When clicking on any bracket, that bracket and the corresponding enclosing bracket will be marked by green (like in picture to the right above).

Similar bracket handling is also applied for brackets enclosing tags when working with html code in annotations.

Annotations handling

Please see section “Components, connections and annotations” on page 274 above concerning expanding/collapsing of annotations.

Documentation annotations in HTML format will have automatic color coding of HTML tags, and the bracket handling above will also work for brackets enclosing HTML tags.

Expanding components and connections

Please see section “Components, connections and annotations” on page 274 above concerning expanding/collapsing of components and connections.

Comment out selected rows

Sections of the code can be commented out, to facilitate programming and testing. To comment out a section, select it, right-click and select **Comment Selection**; that will comment out that section.

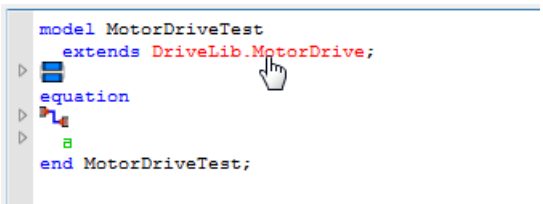
Line number

The number of the current line is shown in the lower status bar.

Navigation in the Modelica text layer

You can navigate in two ways; you can (ctrl)click links or use the context menu.

You can open a link to a class by pressing **Ctrl** and clicking the link. The linked class will be opened in a new tab.

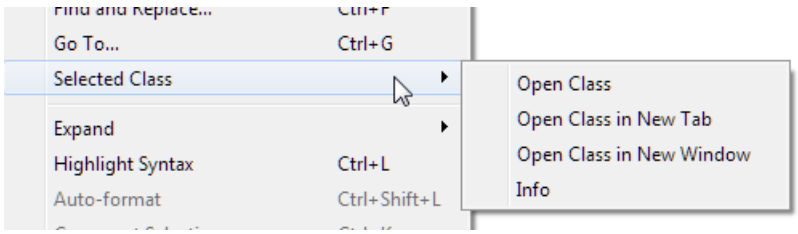


The context menu gives more options. It is possible to select a class in the text, and then open that class in the present window (in the present tab or in a new tab), or in a new window. It is also possible to look at the documentation of that class. Do the following:

Select the class (e.g. a model or a function) by marking it or by putting the cursor inside the name of it. The latter is only possible if the class is not read-only. Please note that the full class name has to be selected.

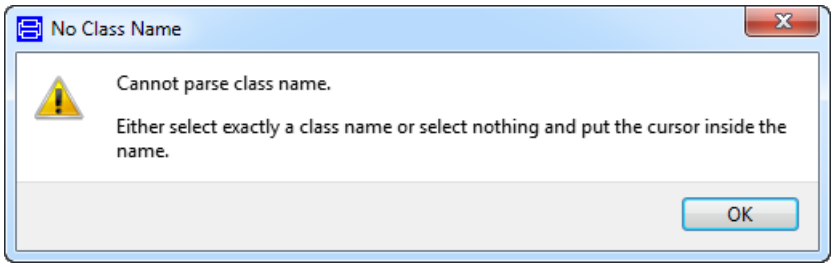
Right-click and select **Selected Class**. The following menu will be the result:

Navigation in Modelica Text editor.



It is now possible to open the selected class in the present window (in the present tab or in a new tab), to open it in a new window, or to look at the documentation of the selected class (by selecting **Info**).

If the selection of class is not successful, the following will appear:



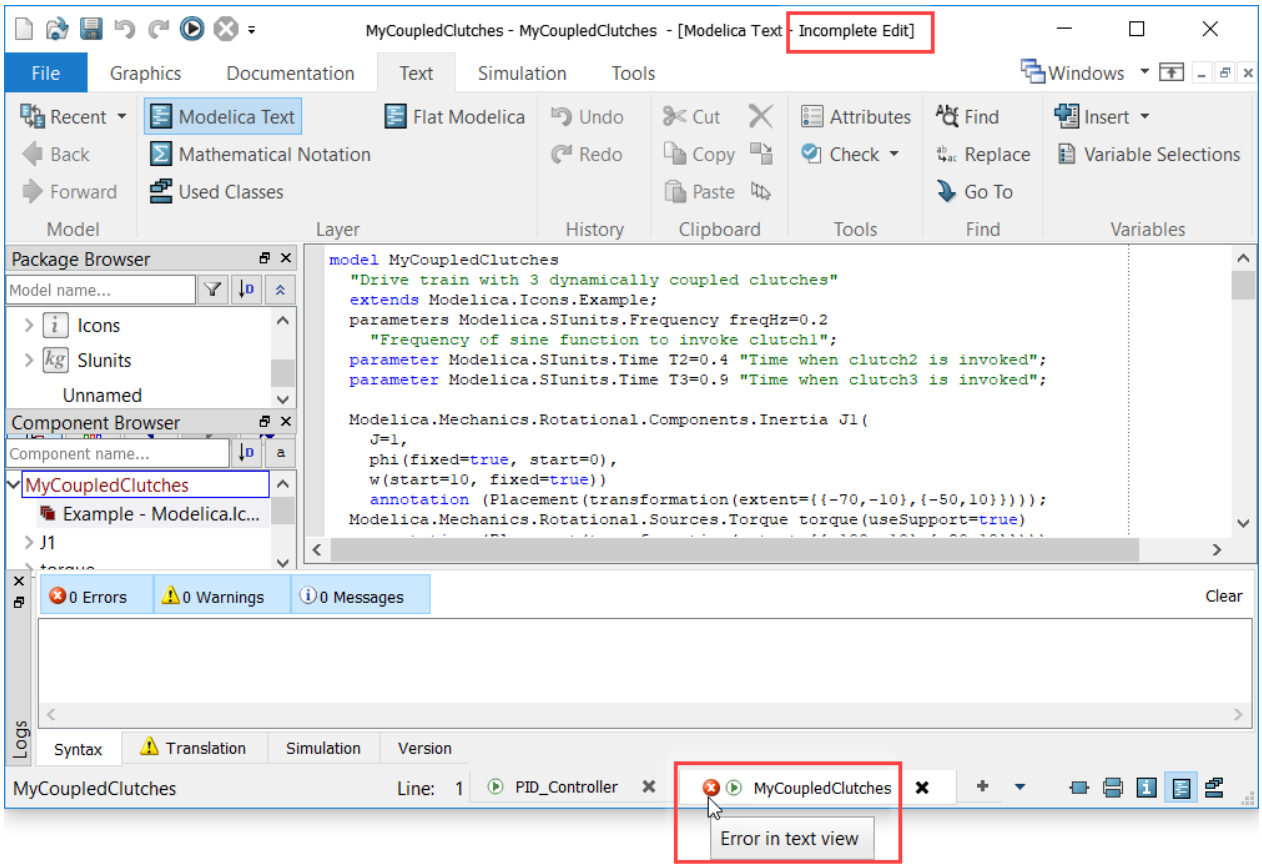
Syntax error checking

There are a number of ways to perform a syntax check:

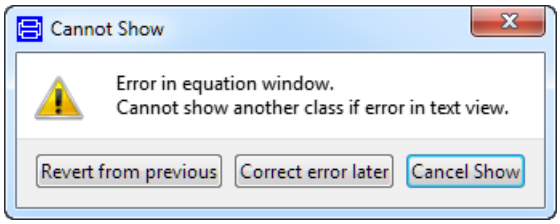
- Right-click and select **Highlight Syntax** from the context menu (or use the shortcut **Ctrl+L**). The command will also automatically color-code the existing code, please see above.
- In the **Tools** group, click on the **Check** button (or press **F8**). This will check more than the syntax, see description of this command.
- Click on the **Modelica Text** button.
- Switch what layer is presented, e.g. change to the diagram layer.
- Display another class in Modelica Text layer.
- Display another tab.
- Save or exit Dymola.

If a syntax error is detected, the position of the error is shown by the cursor.

Note that when a syntax error is present in a model, it is indicated by an error icon in the tab. Also, when the model is active, the text in the top header of the window contains “Incomplete Edit”.



When the user wants to apply certain actions (e.g. display another class in the Modelica Text layer or exit), the present text is checked for syntax errors. If such errors are found, the user has to select according to the following:

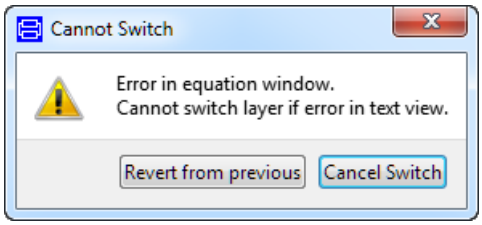


Revert from previous will discard the erroneous changes (and all changes after that) made in the displayed class before action. Please observe that such discarded changes cannot be displayed again.

Correct error later will allow the action although errors are present in the class presently displayed. When later going back the erroneous code will again be show.

Cancel Show will cancel the action – the present class will still be displayed.

If trying to switch layer, the following message is given:



Please note that the error check applied is a syntax check – a number of errors will not be discovered in this way – please use the **Check** button (or press **F8**) for fault-finding.

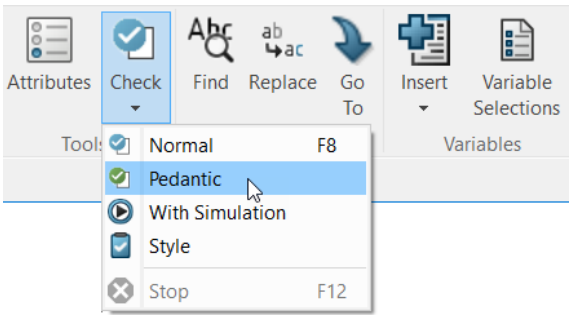
Printing

The command **File > Print…** is used for printing. A selection of Modelica text layer can be printed. If you have selected a part of Modelica Text, and then do **File > Print…** there is an option to print only the selected text instead of the whole model.

Forcing Modelica compliance

Translation (compilation) of Modelica models may generate warning messages if the model does not comply with the semantics of the Modelica Language Specification. In most cases, these warnings are non-critical and the model will still be possible to simulate. The relaxed checking enabled by default makes porting of old models easier.

In order to simplify development of compliant Modelica models, you can make a pedantic check by selecting **Pedantic** when checking:



This checking will treat all warnings corresponding to semantic mistakes as errors. This will simplify for the model developer to develop compliant Modelica code. Non-recognized annotations will generate warnings.



You can also force the pedantic check mode to be active all the time, independent of what selection you make in the GUI above, by a setting **Pedantic mode for checking Modelica semantics** in the simulation setup. On the **Simulation** tab, in the **Simulation** group, select **Setup**, and select the **Translation** tab - here is the setting. Switching to a pedantic compiler mode will treat all warnings corresponding to semantic mistakes as errors. This will force the model developer to develop compliant Modelica code.

If pedantic mode is set, non-recognized annotations will generate warnings when using the Check command.

Editing of certain components in the Modelica text layer

Parameters, variables and constants

Declaration of variables.

Declaration of variables can either be done using menus or by typing in text in the Modelica text layer. The former is mostly the most convenient. (Variables can be of variability type parameter, variable, constant or discrete.)

To be more precise, new variables can be defined in the following ways:

- By using the context command **Variables > New Variable…** in the Modelica Text layer
- By, on the **Graphics** tab, in the **Variables** group, clicking **Insert**  .
- By dragging a type class from the package browser to the component browser, selecting **Add Component**.
- By typing in the Modelica Text layer. Selecting the type could be done using the context menu command **Insert Type…**.

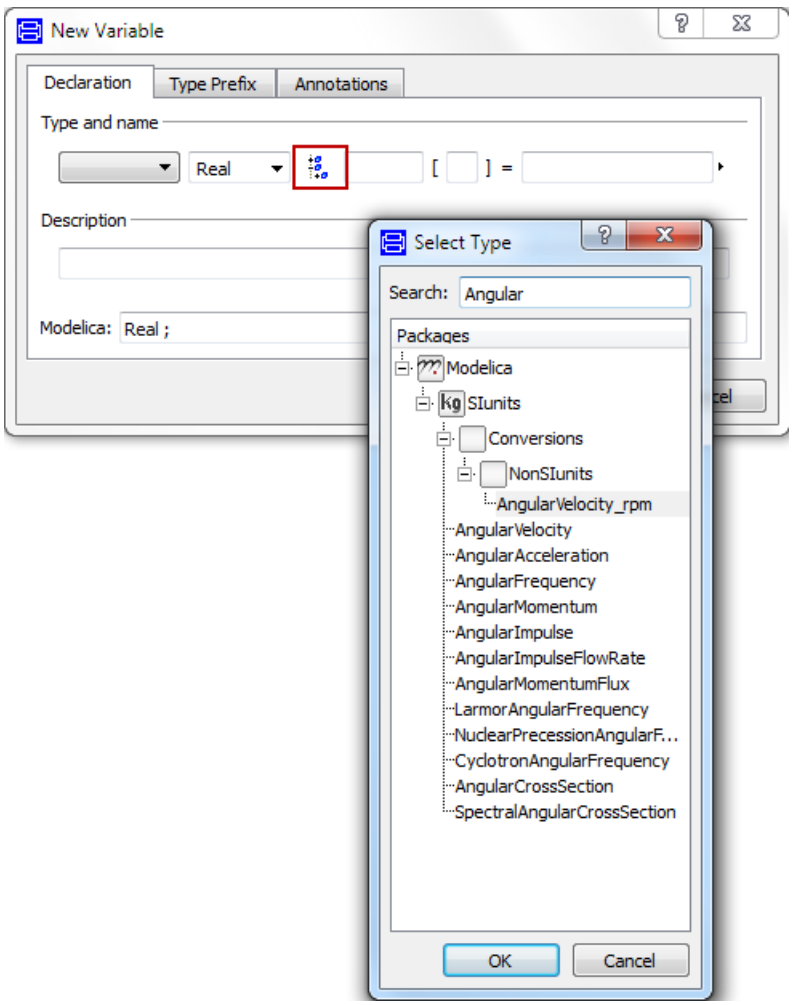
The first three alternatives use the variable declaration dialog, the last one text.

The second and third alternatives are described in detail in section “Declaration of parameters, variables and constants”, starting on page 376.

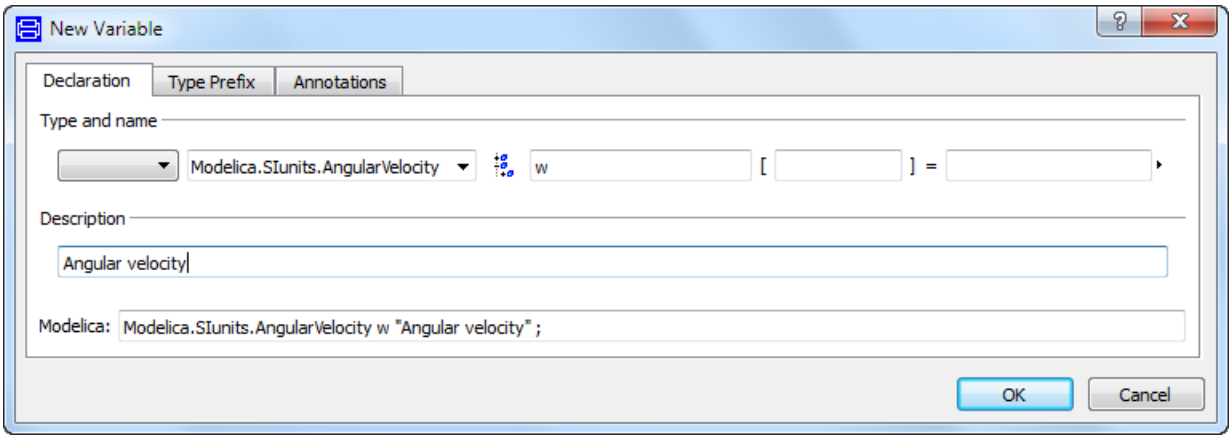
Using any of the three first alternatives about for declaring a new variable, the result will be a variable declaration dialog (with some pre-defined data if dragging is used).

A useful feature in this dialog is to browse and search for a type:

The declaration dialog – browsing for type.



Selecting AngularVelocity, and clicking **OK**, the result is:



Note that the variable declaration dialog contains many possible entries, e.g. for annotations. This dialog is described in detail in section "Variable declaration dialog" starting on page 378".

Clicking **OK**, to the new line will be inserted into the Modelica text layer. The first parameter in the model below corresponds to the menu above.

A variable defined in the Modelica Text layer.

```
model Pendulum
  parameter Modelica.SIunits.Mass m=1 "Mass of pendulum";
  parameter Modelica.SIunits.Length L=1 "Lenght of the pendulum";
  parameter Modelica.SIunits.Acceleration g=9.81 "Gravity of acceleration";
  parameter Modelica.SIunits.MomentOfInertia J=m*L^2 "Moment of inertia";
  Modelica.SIunits.Angle phi(start=0.1) "Pendulum angle";
  Modelica.SIunits.AngularVelocity w "Angular velocity";
equation
  der(phi) = w;
  J*der(w) = -m*g*L*sin(phi);
  ⊟;
end Pendulum;
```

(If dragging is used to define a parameter in an empty model, the "equation" line above will disappear and have to be entered afterwards manually.)

The resulting line can of course be entered directly by typing.

Editing of variables.

The most convenient way of editing an existing variable is to right-click in the Modelica Text layer and use the context command **Variables** and then select the variable. This will pop the variable declaration dialog for that variable. As an alternative, you can, on the **Graphics** tab, in the **Variables** group, clicking the arrow next to **Insert** and select the variable from the list.

Variables can of course be edited directly by typing in the Modelica Text layer.

For more information about the variable declaration dialog, please see section "Variable declaration dialog" starting on page 378. For more information about variables in general, please see section "Parameters, variables and constants" starting on page 376.

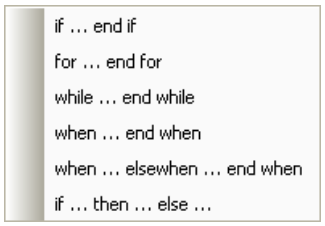
Deleting variables.

Deleting existing variables can also be done by the variable declaration dialog; there is a **Delete** button in the dialog. Deletion can of course also be done by deleting the corresponding text in the Modelica Text layer, but it is better to use the variable declaration dialog since this will give you the possibility to update related modifiers. For more information, see section "Deleting parameters, variables, and constants" starting on page 383.

Statements

Statements can be conveniently inserted in the text by right-clicking and putting the cursor on the entry **Insert Statement**. The following selections can then be made:

Statement insertion using menu.

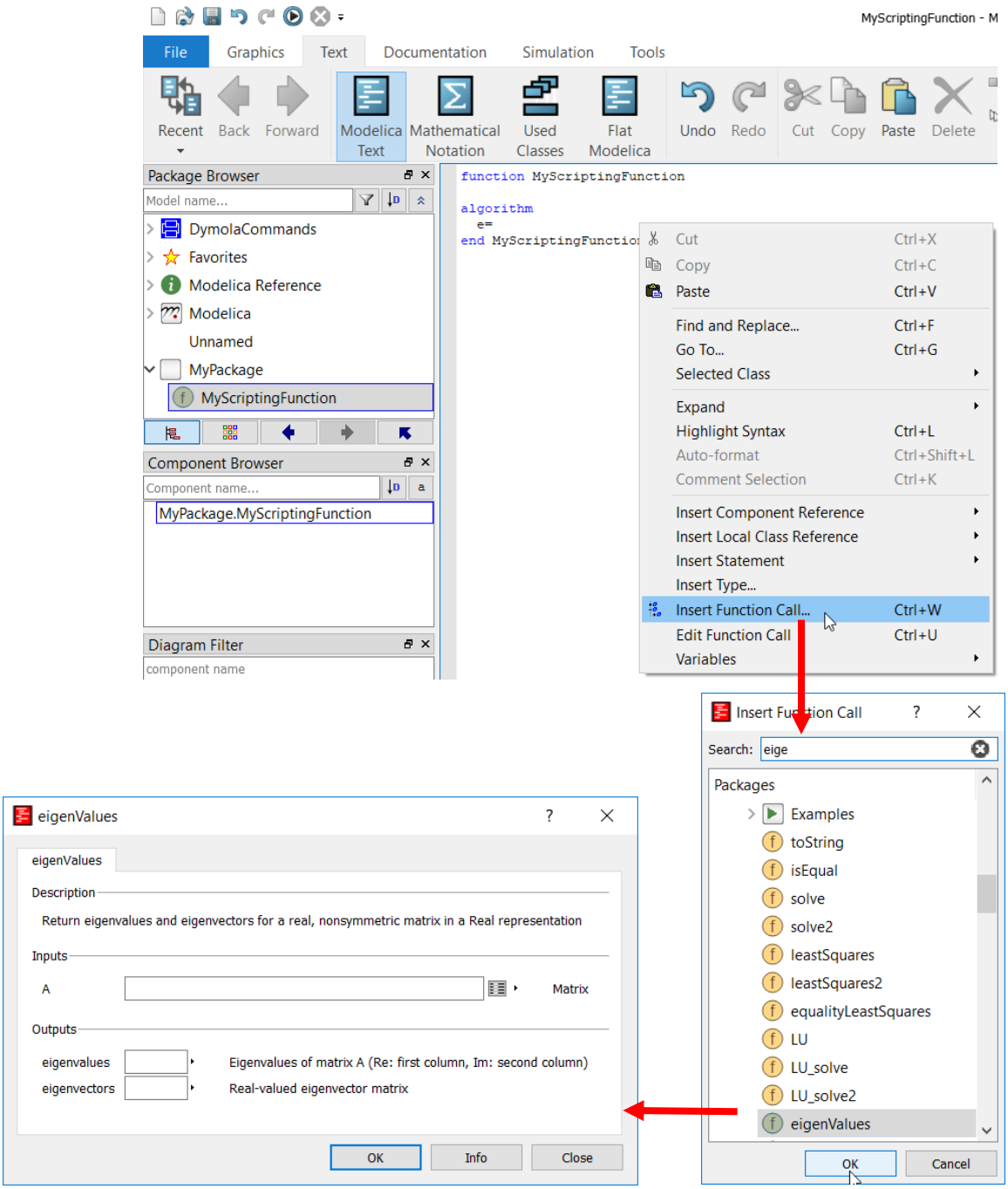


Handling of function calls

Function calls are important e.g. in scripting. Two features are implemented in the context menu of the Modelica Text editor – **Insert Function Call** and **Edit Function Call**.

Insert function Call...

By right-clicking and selecting **Insert Function Call...** (or using the short-cut **Ctrl+W**) a browser will pop, which enables searching and selection of relevant function call and entering of relevant arguments. The below example shows how to use **Insert Function Call...** to insert the function call `Modelica.Math.Matrices.eigenValues` in the function `MyScriptFunction`.



What is entered in the **Search** field will dynamically select what is shown in the Packages pane. The choice selected by default is marked with a light grey background. If the default

selection is the wanted one, it is sufficient to click **OK** to get the function call of that function. Otherwise the wanted function can be selected by clicking on it and then clicking on **OK**: An inserted function is remembered; the next time the **Search** is used, the previous inserted function is shown as default selection in the Package pane.

Please note that nesting of function calls is allowed (function calls in function calls).

Please also note that selected commands (built-in functions) are automatically accessed from DymolaCommands. This library is by default opened when opening Dymola. If not opened, use the command **File > Libraries > DymolaCommands**. For more information about built-in functions, please see next chapter, section "Scripting", sub-section "Built-in functions in Dymola".

Edit Function Call

If a function call is present and should be edited, it is convenient to select it up to ending parenthesis (or putting the cursor inside the name), right-click and select **Edit Function Call** (or using the shortcut **Ctrl+U**). This will open the parameter dialog of the function call, where editing can be made. (The parameter dialog of the function call eigenValues is shown to the left above.)

More information about the use of functions and function calls can be found in next chapter, section "Scripting".

Component reference and Local class reference

Component references and local class references can conveniently be inserted by right-clicking and putting the cursor over the entry **Insert Component Reference** and **Insert Local Class Reference**, respectively. Possible selections will be shown, and by clicking on any of those, that one will be inserted.

Comments and descriptions

Comments and descriptions can be inserted in a number of places by a number of tools. Some examples:

- Any line starting with // will be treated as a comment. A command to comment out selected rows is also available in the context menu – **Comment Selection**.
- A comment can be added in the end of a line by typing // followed by the comment.
- A comment can even be put inside a declaration of a component, inside an expression etc., by using the syntax /*comment*/.
- Description of the class can be added when creating it (or afterwards).
- Descriptions for variables can be entered using the Declare variable dialog or by typing in the Modelica Text layer.
- Descriptions for components can be added using e.g. the parameter dialog of each component.

Programming when simulating – saving start values in model

Modified initial conditions and parameter values entered in the variable browser when simulating can be saved to the model, typically after being validated by a simulation, by right-

clicking the result file in the variable browser and selecting **Save Start Values in Model** from the context menu.

It is possible to save to the current model as well as first extend the model before saving.

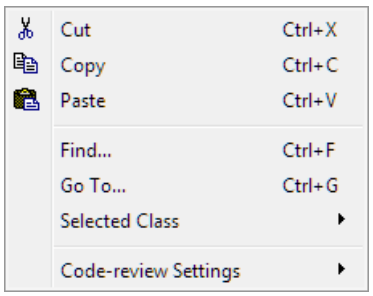
For more information, refer to the next chapter; you can use the index entry *"save : start values to model"* in the index in the end of the manual to find the information.

Inspecting code in the Used Classes layer

The Used Classes layer can be used either to display Used Classes (of any relevant class) or the Flat Modelica text (of certain items).

Please note that no editing is possible in the Used Classes layer.

A context menu is available by right-clicking in the Used Classes layer:



More info about this menu can be found in the section "Context menu: Edit window (Used Classes layer)" on page 541.

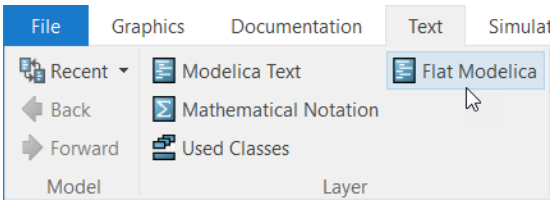
Displaying Used Classes

This is the default setting of the Used Classes layer.

The Used Classes' layer shows the base classes, classes used for components and connectors, and the Modelica text of the class. This facilitates searching for the location of e.g. a specific variable.

Displaying Flat Modelica text

If any of the following classes is selected in the package browser, it is possible to show the Flat Modelica text of that item. This is done by **Flat Modelica**.



The classes supported are:

- Models
- Blocks
- Media packages
- Connectors
- Partial models
- Functions
- Type and other short class definitions
- Attributes of inherited classes (e.g. buses).

As an example, consider the model `Modelica.Electrical.Analog.Basic.Inductor`. The first image below shows the model displayed in the Modelica Text layer; the second image shows the model displayed as Flat Modelica text in the Used Classes layer.

Modelica text.

```
model Inductor "Ideal linear electrical inductor"
  extends Interfaces.OnePort;
  parameter SI.Inductance L(start=1) "Inductance";
equation
  L*der(i) = v;
  a;
end Inductor;
```

Flat Modelica text.

```
model Inductor
parameter Modelica.SIunits.Inductance L(start = 1) "Inductance";

Modelica.SIunits.Voltage v "Voltage drop between the two pins (= p.v - n.v)";
Modelica.SIunits.Current i "Current flowing from pin p to pin n";
Modelica.SIunits.Voltage p.v "Potential at the pin";
Modelica.SIunits.Current p.i "Current flowing into the pin";
Modelica.SIunits.Voltage n.v "Potential at the pin";
Modelica.SIunits.Current n.i "Current flowing into the pin";


// This model
// class Modelica.Electrical.Analog.Basic.Inductor
  // extends Modelica.Electrical.Analog.Interfaces.OnePort
  equation
  v = p.v-n.v;
  0 = p.i+n.i;
  i = p.i;
  // end of extends
  L*der(i) = v;

end Inductor;
```

Comments are automatically added in the text: containing e.g. information about the origin of redeclared classes.

Keywords are in blue, comments in green, other items are black. (Presently types and operators are not red but black.)

Displaying a class as Flat Modelica text is useful e.g. when investigating a model where the concept of replaceable classes is heavily used (like large models using replaceable classes for many media calculations).

If the class selected in the package browser is *not* any supported class in the list above, Used Classes will be displayed, in spite of Flat Modelica being selected.

In two cases it is not possible to show an item as Flat Modelica text:

- The item contains errors. When trying to display e.g. such a model, the following text will be shown:

  ```
  // Errors were detected when making a flat Modelica
  description.
  // Please, check the model and correct the errors.
  ```

- The item is encrypted. When trying to display an encrypted class, the following text will be shown:

  ```
  // Flat Modelica could not be generated for this class.
  ```

4.2.8 Documentation

Introduction

The documentation in Dymola can be divided into an internal part and an external part, where the internal part is viewed in Dymola, while the external part builds on exporting documentation in various forms from Dymola for use in external applications, e.g. a free-standing HTML document or a MS Word document.

An important part of the documentation is the documentation layer that is available for any Modelica class in Dymola. The documentation layer can be viewed directly but it is also the foundation for extended documentation that can include more information than is present in the documentation layer itself (e.g. Modelica code, icons etc.) This documentation can be reached internally (using e.g. the **Info** button or **Info** entry in a context menu) but can also be made available externally by exporting the documentation.

This documentation in Dymola is available in HTML format (Markdown format is also supported). Dymola can automatically produce HTML code for Modelica classes. Models are described both graphically with bitmaps of the icon and diagram layers, and textually with descriptions and full model code. The HTML code may include all classes used as base classes or as components to ensure a complete description of the model or the package. All references to classes are hyperlinks, which makes browsing easy and efficient. Class libraries are described with a clickable list of its components, followed by the description of each component.

The user may include arbitrary HTML code in the documentation. This makes it possible to take advantage of images, tables and other forms of formatting.

Cascading Style Sheets (CSS) are supported.

We will start by mentioning some ways of viewing information internally in Dymola, then looking at the content of the documentation layer. Having that information, we will look at how to view the documentation layer (and more). We will discuss how to edit parts of the documentation layer using a documentation editor (or using HTML direct). Finally, we will mention the creation of HTML documentation and the external documentation (e.g. export for printed documentation).

Internal documentation

Some ways of viewing information inside Dymola

There are a number of ways to display information of different type inside Dymola:

- Using the documentation layer of an edit window. Most of the information about a class is displayed here, however not code etc.
- Using the **Info** button in e.g. the parameter dialog or the **Info** entry in the context menu of the instantiated class. One part of the information is automatically generated and cannot be changed, one part can be configured. This alternative can display the maximum amount of information.

- Using the parameter dialog will display certain information, e.g. comments on parameters etc.
- The Modelica text layer (the Modelica code) can contain comments on the code.
- Hovering over a component displaying the tooltip. The tooltip will include any comment added by the user for that instance of the class.

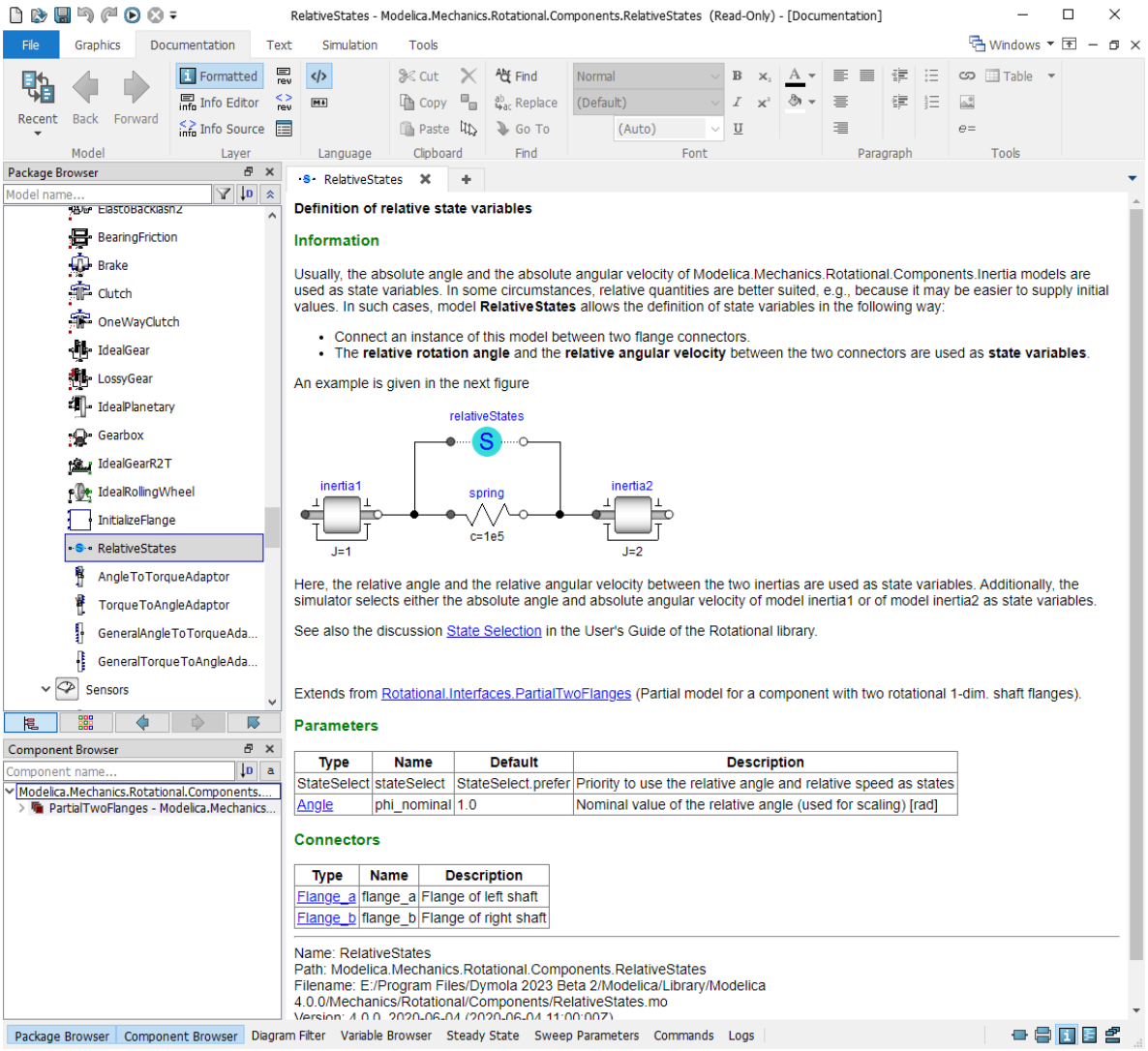
The three last items have been dealt with previously. For more information about this, please see the description of these features. The first two items will be treated here.

Any documentation must of course in the first place be provided by the author of the model. That goes for the documentation layer as well as for any parameter dialog etc. If no documentation is available in the model, Dymola will search the base classes for documentation.

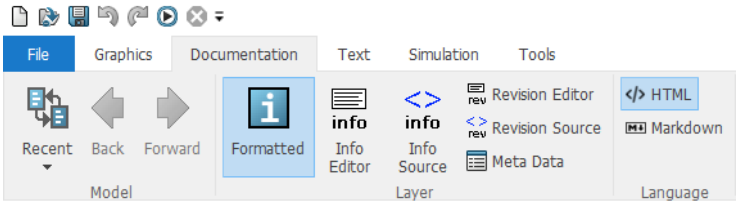
The content of the documentation layer

A suitable starting point in describing the documentation is the documentation of a class in the documentation layer of the edit window. To display this layer, click the **Documentation** tab.

Looking at such a layer, the following might be seen (please also note that the component browser gives the full path to the class):



The documentation layer can be used to look at the formatted documentation (like in this image), but can also be used to edit the documentation or display the HTML code, of the information part and revisions part of the documentation. Metadata and markdown can also be handled. All this is done using the layer and language alternatives:



The alternatives are the following:

 **Formatted** – this is the result from using the below modes. This is the default mode; the icon is the same as the one for the documentation layer.

 **Info Editor** – this mode is the documentation editor, to edit the information part of the formatted documentation. If the class is read-only, this button will not be active.

 **Info Source** – this mode displays the HTML text of the information part of the formatted documentation.

 **Revision Editor** – this mode is the documentation editor to edit the revisions part of the formatted documentation. If the class is read-only, this button will not be active.

 **Revision Source** – this mode displays the HTML text of the revisions part of the formatted documentation.

 **Meta Data** – this is a separate command to add or modify metadata. For more information, see section “Generating and editing metadata” on page 320.

 **HTML** – this the default language mode of using HTML text.

 **Markdown** – this mode means using markdown language. For more information, see section “Markdown language support” on page 322.

This section describes the default, formatted documentation; later sections will describe other alternatives.

Looking at the documentation in the layer, the first part “**Definition of relative state variables**” (always present) is a short one-line description string of the class. This information can be viewed in several places e.g. the parameter dialog, using the **Info** button in the context menu and here in the documentation layer. This part of documentation can be edited by the user, please see below.

The second part “**Information**” (always present) contains an auto-generated header (that cannot be edited) and two sections of documentation.

The first section is a longer text describing the class more in detail. This section can be edited by the user in a documentation editor if the class is not read-only, please see below.

The second section appears if the class is extended from one or several other classes. In that case the class(es) are listed in the order they are declared. Descriptions are also presented. This section is auto-generated and cannot be edited.

Both sections are available using the **Info** button in the context menu and in the documentation layer.

The header "**Parameters**" (if parameters are defined) is header of a section consisting of information of the parameters defined (type, name, default and description). This information comes from the parameter declaration of the class and can be partly edited by the user in the parameter dialog (if allowed).

The header "**Connectors**" (if connectors exist) is header of a section consisting of information about the connectors available. This information comes from connector declaration and can be edited by the user in the parameter dialog for each connector (if allowed).

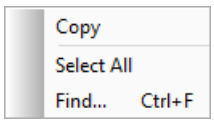
The header "**Inputs**" and the header "**Outputs**" usually is available for functions. This information comes from function input and output.

The header "**Package content**" will be available if the package contains sub-components – this is the case for most models. This section will contain a list of the packages with names and descriptions. The descriptions correspond to the first part of documentation in the documentation layer of the corresponding sub-package.

(Sometimes custom sections might be available, e.g. "**Release Notes**".)

The last part of the documentation is the **Revision** documentation (not included in the figure above). This information is used to list changes in the class. This information is normally not included in generated external documentation. This part of documentation can be edited by the user in a documentation editor if the class is not read-only, please see below.

At the end of the documentation pane, there are a number of autogenerated texts, containing name, path, filename, version, and uses of the model. "Uses" (not visible above) is a list of libraries used in the model.



By right-clicking in the formatted documentation layer, a context menu is available.

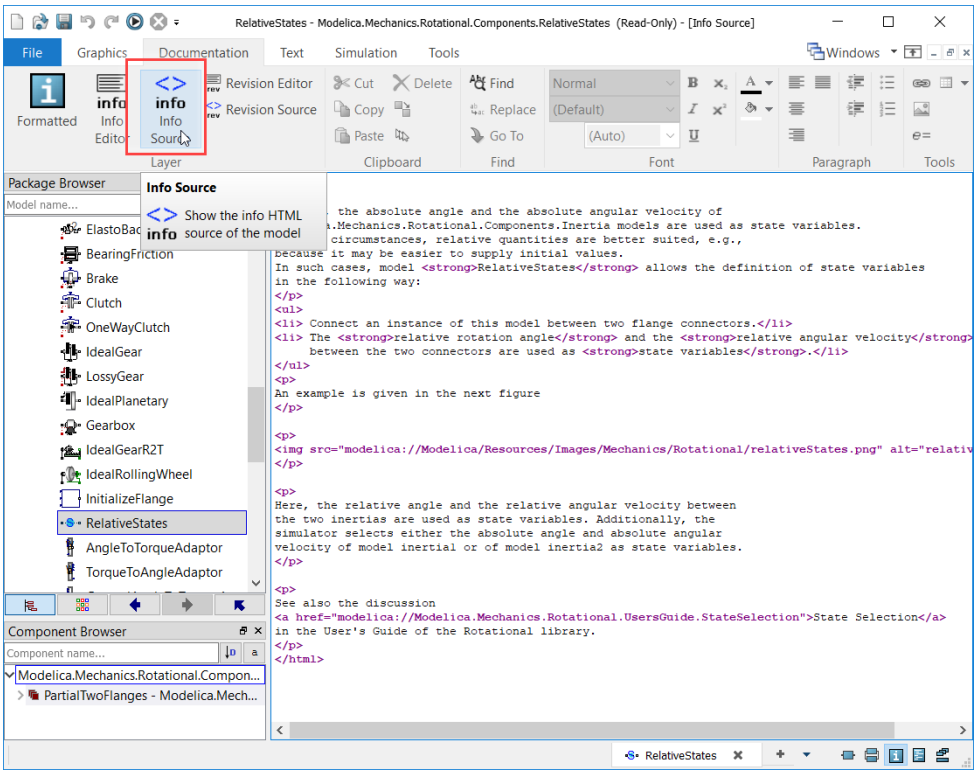
More information about the context menu can be found in the section "Context menu when viewing formatted documentation" on page 531.

Note that when copying text from the formatted documentation layer, the text is now only copied as plain text. The recommended way to copy text with HTML tags is to copy from the source text (see below). The copying can however be restored to the way it worked in previous versions (Dymola 2016 FD01 and older) by setting the flag `Advanced.CopyFormattedHtml=true;`.

Looking at the HTML source code of the documentation layer

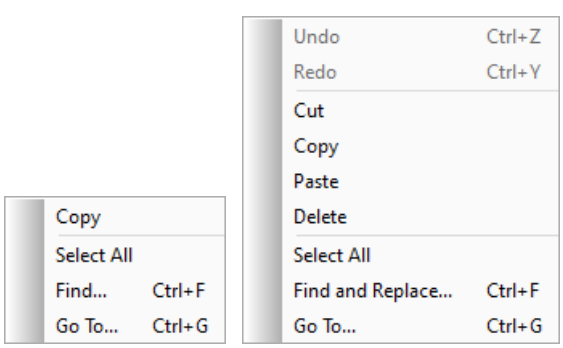
Two parts of the documentation layer can be viewed as unformatted HTML source code when in the documentation layer. It is the information part and the revisions part. By using the command buttons **Info Source** and **Revisions Source**, respectively, in the toolbar of the documentation layer, the HTML source code can be seen.

The HTML source code of the information part of the documentation layer.



Please note that auto-generated components are not shown, e.g. the header “Information” and information about “extended from” (if any).

A context menu is available by right-clicking; it looks different depending if the text is read-only or editable (read-only to the left):



For more information, please see section “Context menu for HTML source code for info/revisions part” on page 533.

To go back to the formatted documentation, click the command **Formatted**.

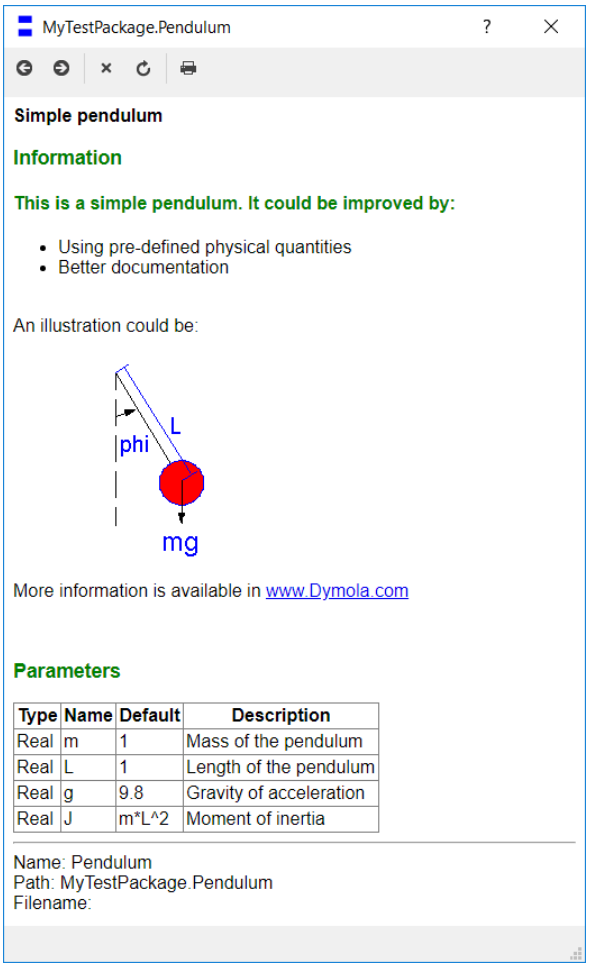
Looking at the documentation layer (and more) using the Info button/menu entry

Info is available as a button (in e.g. the parameter dialog of an instantiated class) or as an entry in the context menu. **Info** can be made to display the maximum information of a class.

What information **Info** displays depends on two things; whether HTML documentation is generated for the class in question or not, and the selection on what information should be generated when generating the HTML files.

If no HTML files are generated, the browser displays the content of the documentation layer, plus some info about filename, path etc.

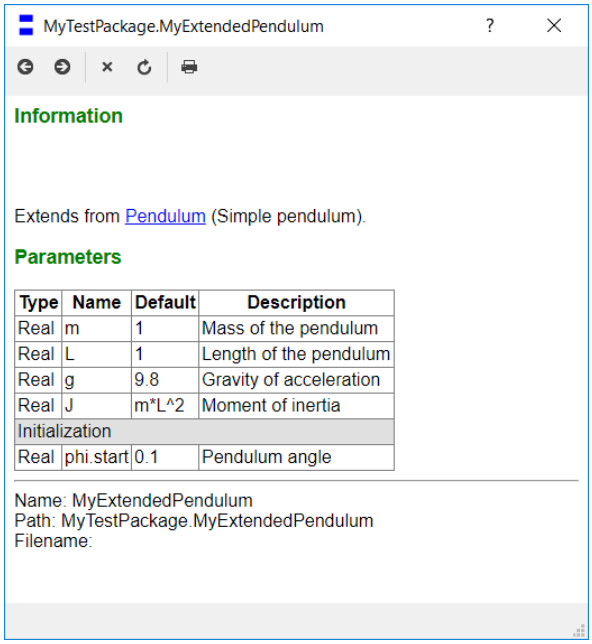
Info on class that has no generated HTML file.



Notes:

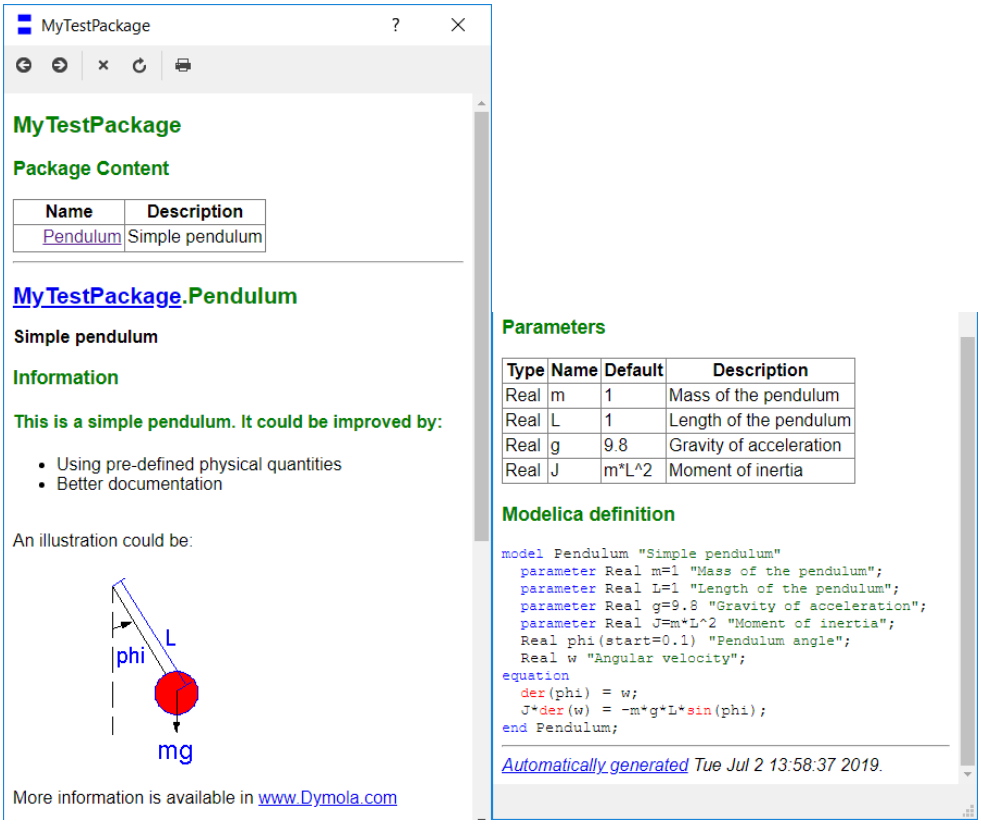
- The model in this example is very simple. In most cases, a section Connectors is also displayed.

- If a model has been extended, and no additional documentation has been added, the corresponding parts (Information and Revisions) are empty for such a model. However, the information that the model is created by extending is displayed:



If HTML files have been generated, the result might look like (the size of the window has been adapted to show all info, and the images has been split in two to fit into the document):

Info on class with generated HTML file.



By comparison it can be seen that by default more information is generated if HTML documentation is generated (however, by default the Revisions part is not generated). Such documentation is either generated by the library developer before delivering Dymola (all free and commercial libraries have such files generated before deliverance to the customer) or, when it comes to user-defined packages; the user himself/herself can generate the files with wanted content. Please see section “Tools > [Export] HTML” starting on page 497 for more information about generation and content of such files.

The documentation viewed in this way usually contains more information than looking at the documentation layer, e.g. an icon as well as some code. Sometimes the class itself is not shown initially, rather the package containing the class. It is however easy to reach the right class using the links under the **Package Content** header.

Protected classes are not included if not shown in the package browser.

The browser used for viewing the information contains the usual **Go back**, **Go forward**, **Stop**, **Reload**, **Copy** and **Print** facilities.

Editing the documentation layer

The one-line description string (the first part of the documentation in the documentation layer) can be edited by, on the **Text** tab, in the **Tools** group, selecting **Attributes**. This command can also be used to change a number of other class attributes, see “Graphics > Attributes” starting on page 479 for more information.

The second part of documentation, the longer information text (with the exception the auto-generated information about extending, if any), can be edited either using a documentation editor or directly in the HTML source code if not read-only. Please see separate sections about the documentation editor and working in the HTML source code below.

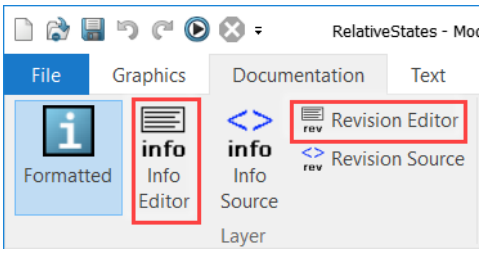
The revisions part in the end of the documentation layer is editable in the same way (documentation editor/HTML source code).

The rest of the documentation parts are not editable in the documentation layer.

Using the documentation editor

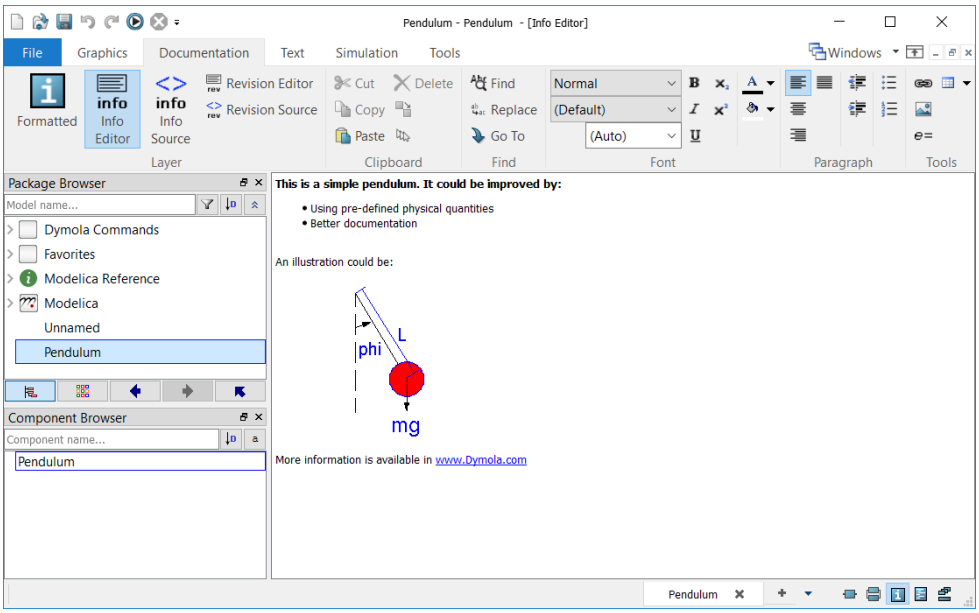
The documentation editor.

The documentation editor is reached in the documentation tab by clicking any of the command buttons **Info Editor** or **Revisions Editor**, respectively (depending on what should be edited). Note that the class must not be read-only.



Using the former might yield the following result:

Example of documentation editor for info part.



Details of the documentation editor

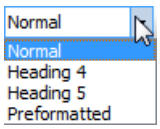
The Font, Paragraph, and Tools group of the tab looks like the below when using maximum window size:





Style contains presently four selections, **Normal** being default. **Preformatted** is the text style of text generated from Dymola.

Text style alternatives.



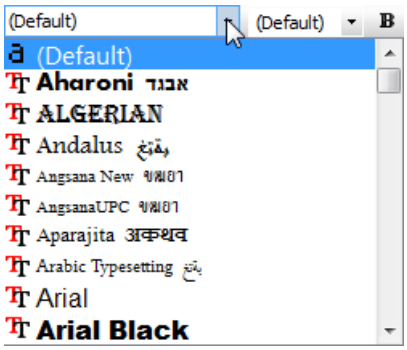
When a text has been entered in a heading style, the next paragraph will by default be of style Normal.

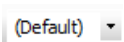
Heading 1-3 are reserved for the automatic Modelica documentation and cannot be used in the editor.



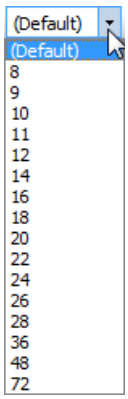
Font contains a number of fonts.

Font alternatives.





Font Size contains a number of font size alternatives.



When changing the base font size in Dymola, the text sizes in the documentation layer will change accordingly. (E.g. if the base font size is changed from 8 to 10, all text styles will be 2pt larger in the documentation layer; the default size will then be 10pt as well.)



The **Bold** button will change the selected text to bold (or, if no text is selected, any text inserted afterwards will be bold). The button will be marked activated, as it will be for any bold text marked. An active bold button looks like: 



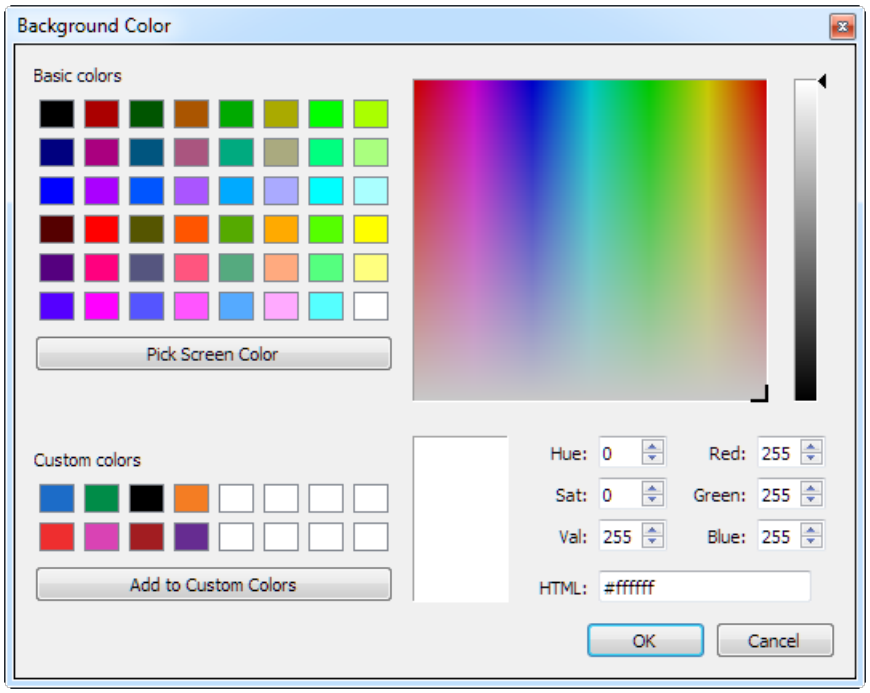
The **Italic** button will change the text to italics; the **Underline** button will change the text to underlined.



Two buttons are available for subscript/superscript.



Two buttons are available for text color/background color. Clicking the arrow after any of those gives a color selection menu (with corresponding header, below the background color one):



Here any basic or custom color can be selected. Custom colors can be added by using the color definition boxes to the right, then clicking **Add to Custom Colors**.

Selecting a text and the setting the color of the text or the background, will also define the color selection as default, and change the color under the button accordingly. If this default color is to be used, it is sufficient to just click the button when a text is selected, to get the default color.



Four buttons are available for the alignment of selected text: they are in order **Left Align**, **Center Align**, **Right Align** and **Justify**. The default is **Left Align**; that button will be shown active by default.



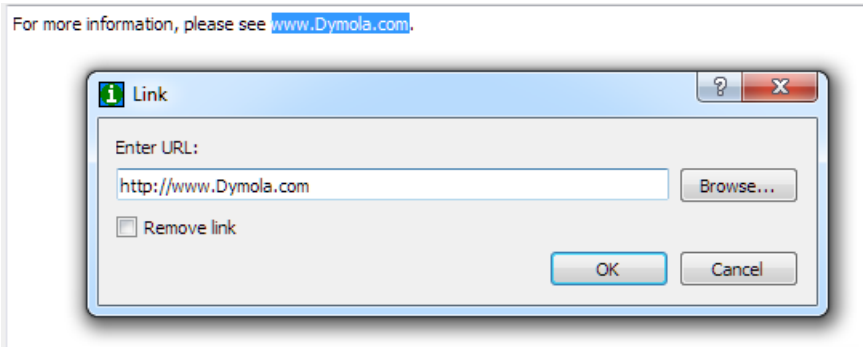
Two buttons are available for changing the indent: **Decrease Indent** and **Increase Indent**, respectively.



Bullet List will create a bullet list; **Numbered List** will create a numbered list. Presently only one level is available. They work the same as the bold button when it comes to activation etc.



The **Link** button will open a menu for entering of an URI according to usual html rules. If you put the cursor anywhere on a link already present and click the **Link** button, that link will be shown.



Note that if you type a text in the documentation layer that can be interpreted as a link this link is automatically created when you do any of **Space**, **Enter** or **Tab** after having typed the text. (This feature can be disabled by setting the flag `Advanced.AutoLinksDoc=false`.)

In the dialog for the **Link** button, the **Browse…** button makes it possible to create local links to e.g. .pdf and .html files (see below). This feature enables creation of documentation that refers to external documentation.

When an URI is entered and the **OK** button is pressed, the selected text will be underlined and blue.

In the dialog for the **Link** button, you can use **Remove link** to remove an existing link.

There is no syntax check of the URI. By pressing **Ctrl** and hovering over the link, the URI will be shown in the status bar of the Dymola main window.

The Modelica URI (Uniform Resource Identifier) 'modelica://' scheme is supported, e.g. hyperlinks "modelica://*Package.Class*". Such links makes it easy to reference Modelica classes in the documentation (used for example in the MultiBody library). An example of such a link is:

```
modelica://Modelica.Mechanics.MultiBody.Joints.Cylindrical
```

Please note when creating the URI that such an URI is case sensitive.

It is possible to select a formatted text when creating the link, as well as an image, and a mixing of text and image. Images will not be underlined when in links, however.

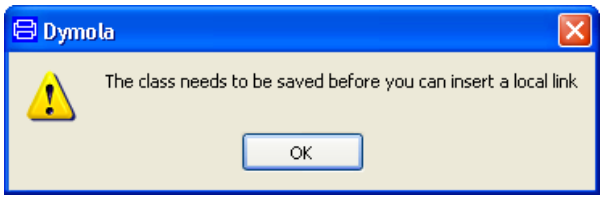
The browser supports anchor-tags using the syntax href="Step" (using normal lexical lookup in Modelica) or href="modelica://Modelica.Blocks.Sources.Step" (using full lexical names). These are transformed into references that the browser can understand when exporting HTML-code.

Anchors with normal HTML syntax are supported. Hyperlinks using fragments #diagram, #text, #info and #icon link to the corresponding layer ("text" refers to the Modelica text layer, "info" refers to the documentation layer). All other anchors are assumed to refer to anchors in the documentation layer. *Note:* When generating HTML documentation, several classes may be included in one file; the anchors should thus be unique independent of the class.

Local links created using e.g. the Browse button will use the modelica:// scheme if possible; absolute path will be used if the document referred is located on e.g. another drive. When

referring to external documentation, it is recommended to store such documentation in a folder (named e.g. LinkedDocumentation) created in the folder where the top package resides. Doing so facilitates moving the package without losing the links; the folder LinkedDocumentation should be moved together with the package.

In order to be able to generate local links, the model/package has to be saved (using e.g. **File > Save**) before a local link can be inserted. If initial saving has not been done, a warning will be displayed:



The resulting link can be activated in the formatted documentation by clicking on it. By hovering over it, the URI is shown in the status bar of Dymola main window.



The **Image** button will open a browser for browsing an image. A number of formats are supported, e.g. .png, .xpm, .jpg, .tiff and .bmp. There is no scaling of the image.

The starting point of the browser is the folder where the relevant top package resides. It is recommended to store images that should be used in Dymola documentation in a folder "Images" located in the same folder as the relevant top package. (It will then be intuitive to move that folder also when moving the package, which will preserve the image references.)

When browsing for the image the file path is shown in the menu. When clicking **Open**, the image will be inserted.

Opening the corresponding html text by right-clicking and selecting **Info source**, it can be seen that the Modelica URI 'modelica://' scheme is used by default when creating the path to the image.

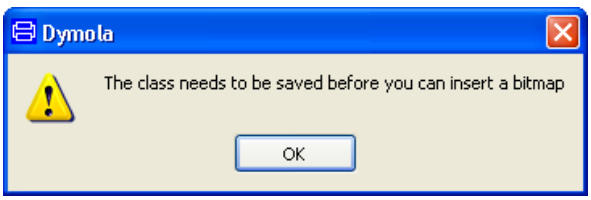
Using the recommended location for images above for an image "Illustration.png" for a top package "MyPackage", the URI

```
modelica://MyPackage/Images/Illustration.png
```

will be the resulting URI in the annotation.

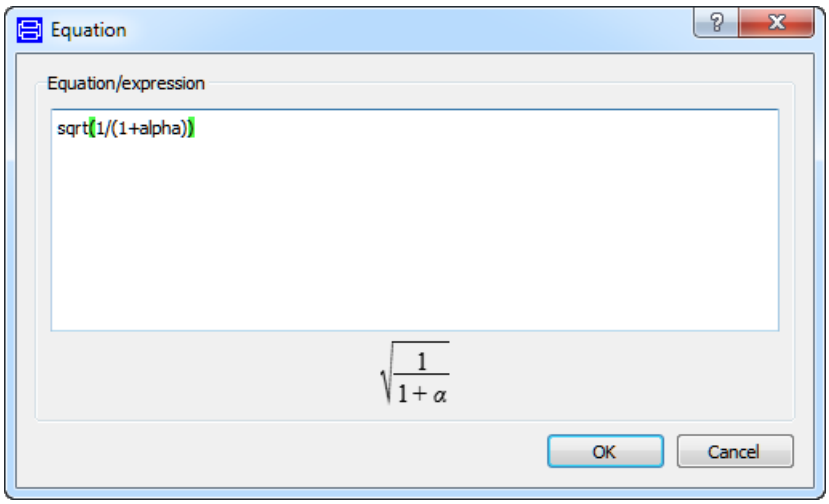
(This URI can also be used in all sub-packages of MyPackage to refer to the image.)

In order to be able to generate correct links, the model/package has to be saved (using e.g. **File > Save**) before an image can be inserted. If initial saving has not been done, a warning will be displayed:



If the link is invalid (image not found) it will be indicated with a symbol.

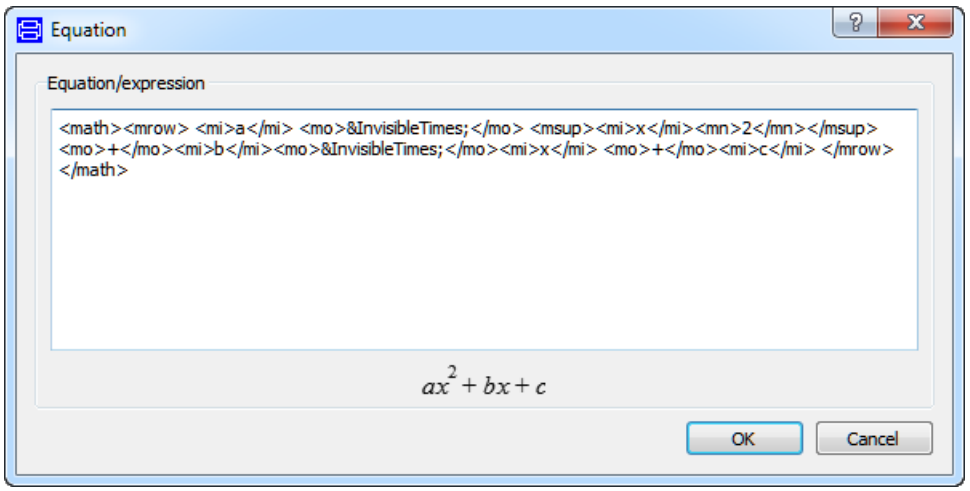
Clicking the button **Equation** opens an editor for the creation of an equation/expression.



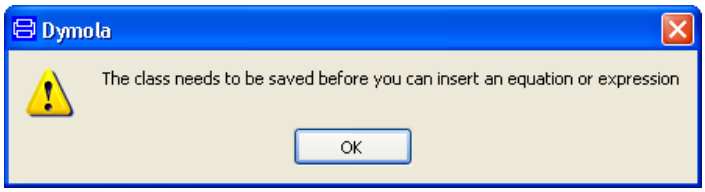
Example of creation of a square root expression.

The result of the editing is shown below the editing pane. If an equation/expression is not concluded, the text `Incomplete equation/expression` is displayed. Clicking **OK** the equation/expression is inserted in the documentation editor where the cursor was located when pressing the **Equation** button in the first place.

Instead of using Modelica syntax, you can use MathML. An example:



In order to be able to generate correct links, the model/package has to be saved (using e.g. **File > Save**) before an image can be inserted. If initial saving has not been done, a warning will be displayed:



Equations or expressions created this way are displayed in italics. They are in fact images; nothing inside can be selected, copying a section containing such items and pasting it into e.g. Microsoft Word will display these items as pictures. They can however be edited in Dymola anyway.

If such an expression or equation should be edited, select that equation/expression. The cursor will place itself after or before the equation/expression. Now click on the **Equation** button. The editor described will be displayed, and editing of the equation/expression can be made. Clicking **OK** the edited equation/expression will replace the previously selected one.

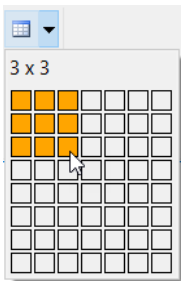
(The images created using **Equation** are stored in a folder `Images\equations` that is located in the same folder as the top package resides. The link to the image will be according to the modelica:// URI scheme. Editing such an equation/expression will create a new image for each editing. The folder `Images\equations` will be automatically created if not present.)

For some examples of the rendering of examples/expressions, please see next chapter, section "Model simulation", sub-section "Documentation".

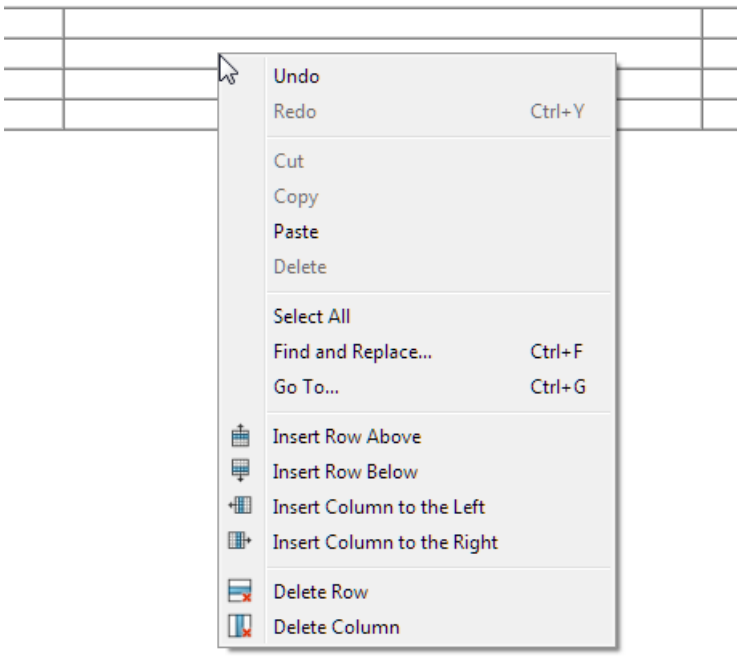
Please note that it is still possible to enter simple expressions/equations by typing them in directly without using the button.



The button **Insert Table** inserts a table. Click on the arrow to insert a table, and use the cursor to span the number of rows and columns.



Selecting a cell in the created table, and right-clicking, gives a context menu that enables editing of the table, e.g. adding/removing columns and rows:

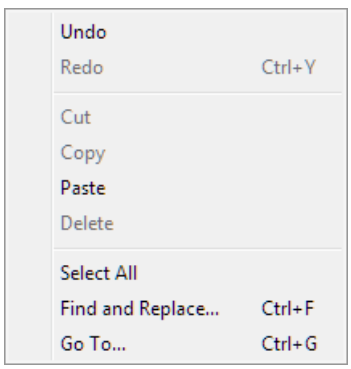


For more information about this context menu, see section “Context menu for table cells in the editable info/revisions part” on page 532.

The revisions part of the information in the documentation layer can be edited using the same editor, but reached by clicking the command button **Revisions Editor** in the documentation toolbar.

A context menu is available by right-clicking in the documentation editor:

Context menu of documentation editor.

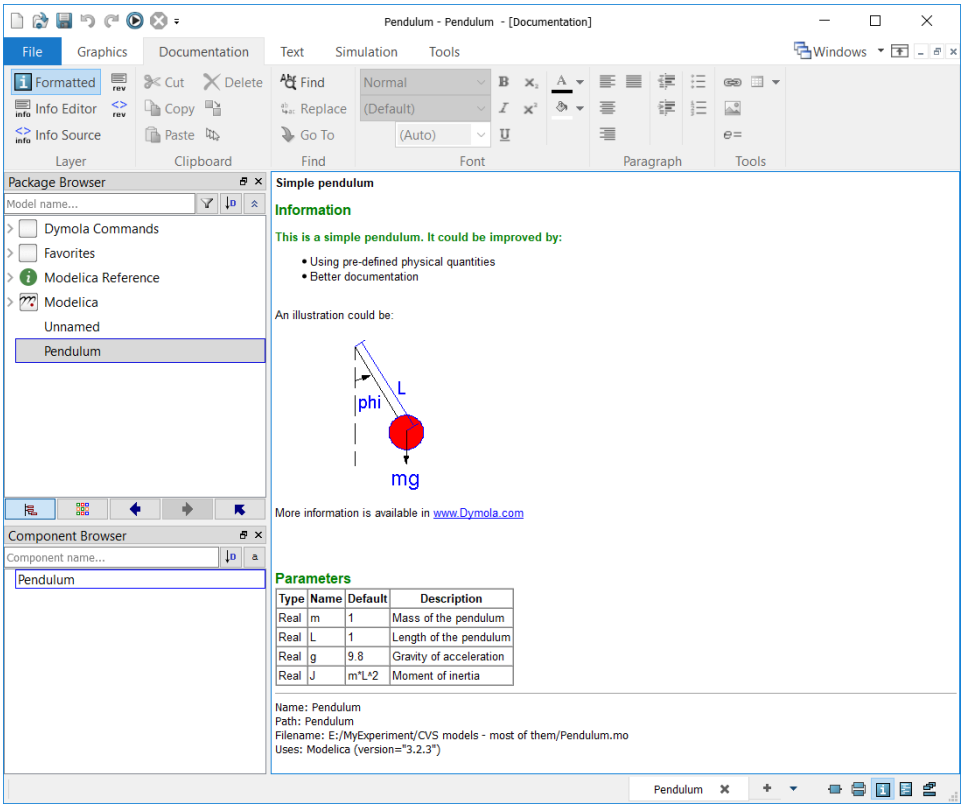


More information about this menu is available in the section “Context menu for editable info/revisions part” on page 531.

Note that when copying text from the Info editor or Revisions editor, the text is now only copied as plain text. The recommended way to copy text with HTML tags is to copy from the source text (Info source/Revisions source, see below). The copying can however be restored to the way it worked in previous versions (Dymola 2016 FD01 and older) by setting the flag `Advanced.CopyFormattedHtml=true;`.

To go back to the formatted documentation, click the command button **Formatted** in the documentation layer toolbar. In this, that might yield the result (here also a simple revisions part has been added):

The resulting documentation layer (including Revisions).





Please note the convenience of using Recent Models button to return to previous model when working with links. (The arrow next to the button makes it possible to select from a number of previously shown classes.)

Editing directly in the HTML source code

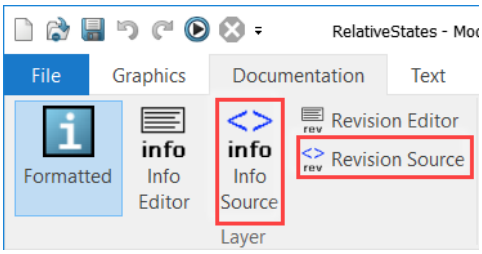
Editing the HTML source code enables including arbitrary HTML code, including constructions that cannot be created using the documentation editor.

Please note that the documentation editor regenerates *all* HTML code when making *any* editing in it. This means that a construction that is not at all supported in the documentation

editor (that is, not supported for rendering) will disappear completely when the HTML code is regenerated.

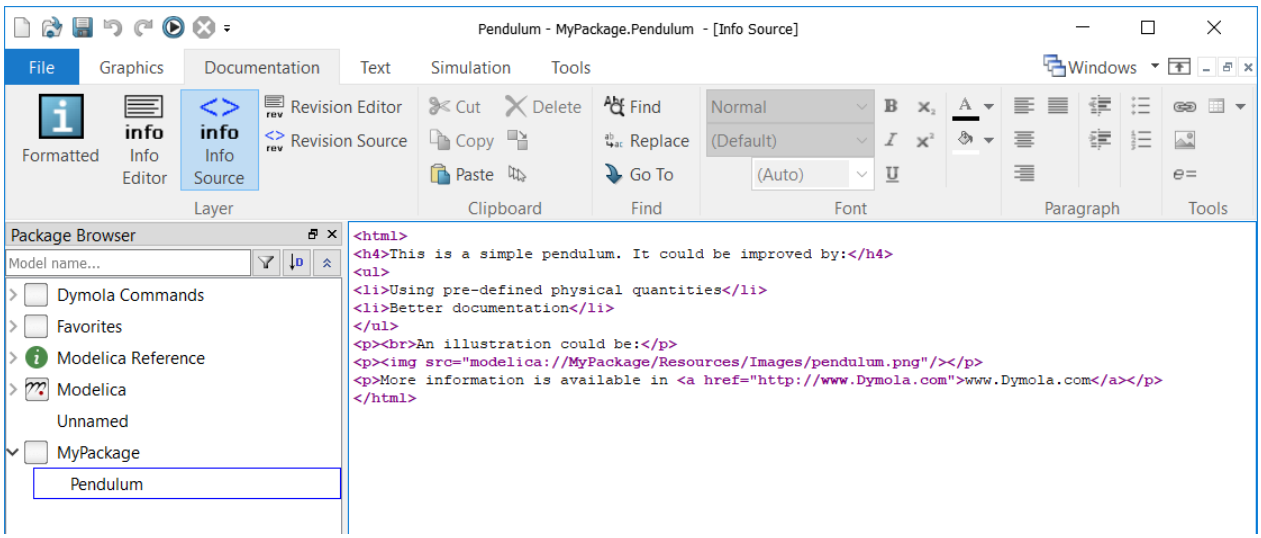
This means that when having started creation of more complicated HTML code, you have to keep away from editing in the documentation editor for that class. (It might be wise in such a case to save the HTML code as a text file before trying the documentation editor.)

The HTML source code for the information and revisions part of the documentation layer is reached by clicking the commands **Info Source** or **Revisions Source** respectively, in the documentation layer toolbar.



The corresponding html source code.

Selecting the formerly for the pendulum above will show:



If the class is read-only, the HTML code can only be viewed.

Protected classes are not included if not shown in the package browser.

Empty pictures are removed.

By right-clicking a context menu is available; the same context menu as in previous section.

To go back to the formatted documentation, click the command button **Formatted** in the documentation layer (see the image above).

Working with Cascading Style Sheets (CSS)

The documentation can be configured using Cascading Style Sheets (CSS).

The default style sheet is `style1.css`, located in `Program Files\Dymola 2023\insert`. This style sheet can be used as a reference, since it lists all tags. (Do not change the name of the file. It might also be wise to keep a copy of the unchanged file if you want to go back to the default setting.) The file is referred from inside Dymola.

If a library has a style sheet `style.css` located in the `Resources` folder of the library, it will be cascaded. This allows for overriding specific tags.

Note. There is currently a limitation when creating a user-defined style sheet; headers must be entered in a descending order to work.

Generating HTML documentation

Typical demands for internal HTML documentation can be:

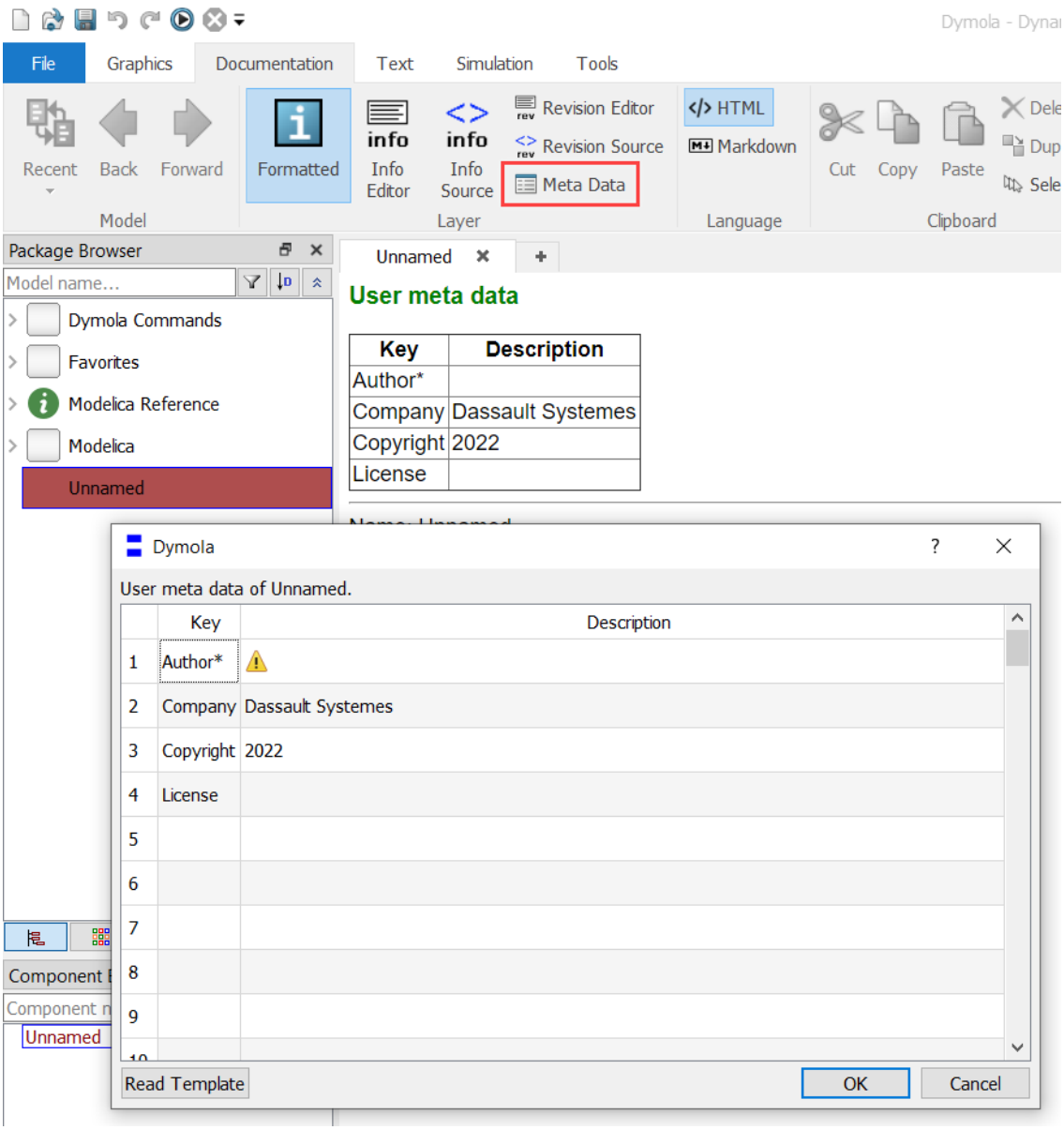
- Online documentation of a selected model, including code.
- Online documentation of e.g. a library without code included.

What content should be included in the files is also possible to change.

For more information about how to create these types of HTML documentation and the different options for contents etc. please see section “Tools > [Export] HTML” starting on page 497.

Generating and editing metadata

Model metadata, in the form of key/value pairs, can be stored in models. The metadata is stored as an annotation in the model, and is automatically displayed in the documentation layer, if present.



The figure above shows the dialog for creating and editing metadata, as well as the result.

The metadata dialog is opened by the command **Documentation > Meta Data**.

Keys and descriptions are arbitrary text strings. Key that ends with * is regarded as mandatory, in the above example `Author*`. A warning sign is shown if the corresponding description field is empty.

If the model is read-only, the metadata is displayed, but the **OK** and **Cancel** buttons in the editing dialog are replaced by a **Close** button.

The **Read Template** button opens a text file with metadata keys, stored as one key and its description (separated by one or more TAB characters) per line. Such a .txt file for the above figure would be:

```
Author
Company          Dassault Systemes
Copyright        2022
License
```

Note. The existing keys and descriptions are erased when reading a template file.

Markdown language support

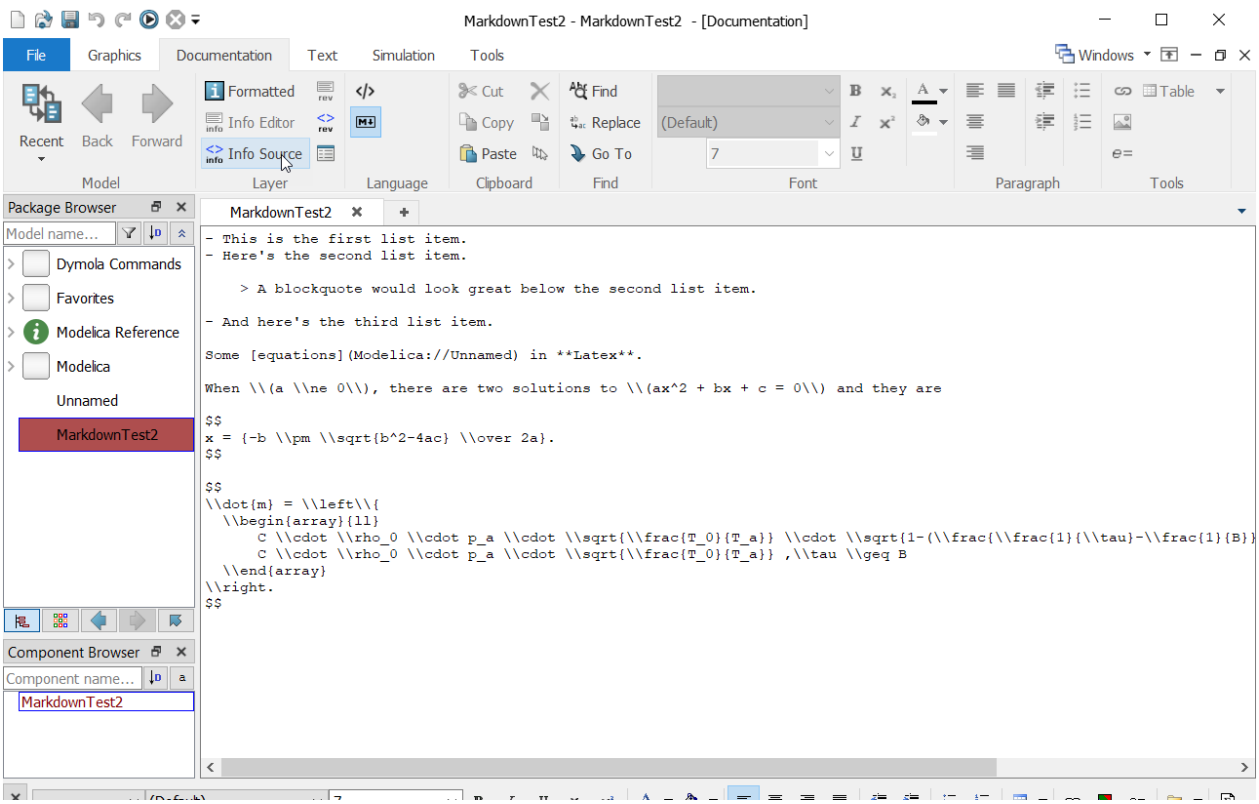


Markdown language of documentation is supported, you can activate it by the command **Documentation > Markdown**. (By default, markdown is not activated, but HTML.)

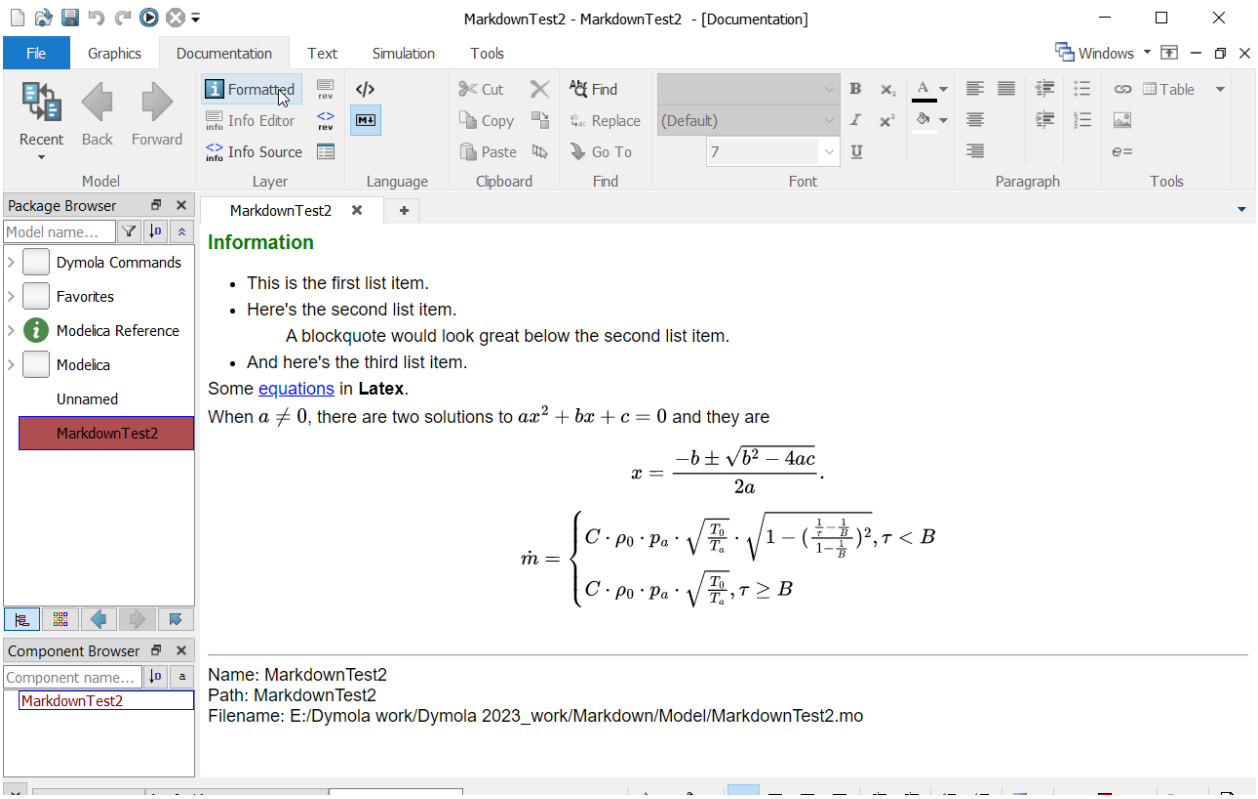
For more information about markdown, see https://www.markdownguide.org.

Note that equations in Latex and MathML are supported. MathJax is used for rendering. An example of Latex text with equations:

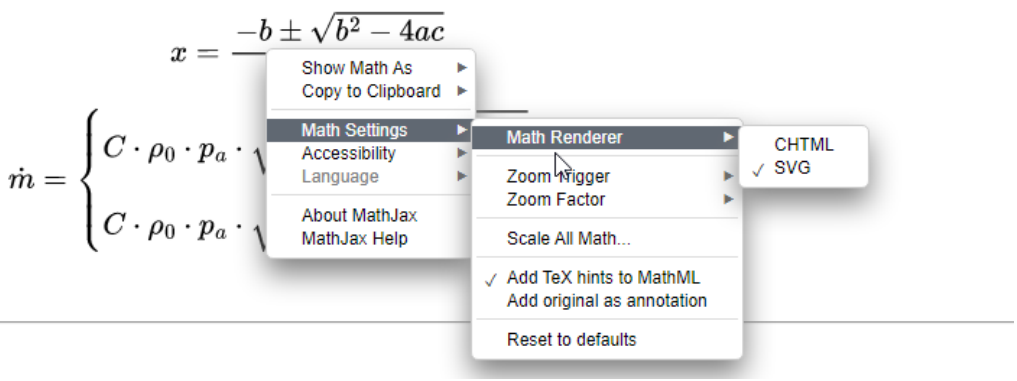
The editor:



The formatted text:



Note that there is an option to change the rendering by right-clicking any equation:



MathJax is by default activated by the flag `Advanced.Editor.MathJax=true`. You can deactivate MathJax by setting the flag to `false`.

External documentation

Documentation for web publication

It is possible to generate complete HTML for a model and all referenced classes. All HTML files are saved in the same directory, which makes them easy movable to e.g. a web-server. For details, please see about the setting **Generate HTML for referenced classes** in section "Advanced tab" starting on page 502. (using the command the **Advanced** tab reached by the command **Tools > [Export] HTML**.) Please observe that a number of other settings in the mentioned command can be used to specify the content of the HTML files.

Printable documentation

The base for printable documentation is by default one HTML file, generated by using the setting **Printable package documentation** in the **General** tab reached by the command **HTML** that you can find on the Tools tab, in the Export group. Please see section "Tools > [Export] HTML" starting on page 497 for details on the command and how to use it, including location of file and bitmaps. Please observe that a number of other settings in the mentioned command can be used to specify the content of the HTML files.

The single HTML file generated in this way automatically contains index fields for each class name that can be used to create an index.

The HTML file can be used for creating a pdf file, either directly or using Microsoft Word or OpenOffice from Sun Microsystems to create a file that can be used to create a pdf. The advantage doing it in this way is that pagination will be automatically created; contents and index can be generated, as well as nice page layout etc.

Two templates are available for the latter case.

- One template `Documentation template.doc` is available in `Program Files\Dymola 2023\Documentation`. This Microsoft Word file builds on linking the HTML file into the template. Instructions on how to use the template are included in the template.
- Another template `Modelica-Documentation-Template` is available in the folder `Program Files\Dymola 2023\Modelica\Library\Modelica 3.2.3\Resources\Documentation`. Actually, it is two templates, one `.doc` for Microsoft Word and one `.ott` for Open Office from Sun Microsystems. This template is the most recent one, well adapted to the present look of the HTML files exported from Dymola. It builds on inserting the relevant HTML file into the document.

Since the result produced with the two templates looks different, it is recommended to try both and use the one that fits best to the present demand.

Whatever template used, it is easy to create index as well as list of content; all relevant information is included in the HTML file.

It is wise to save the result file with another name, not to "destroy" the original template. The resulting file can of course be further elaborated in Word/OpenOffice, depending on the demands of the user.

One thing that usually needs to be improved is mathematical formulas. For e.g. MS Word MathType can be used to generate nicer representation of formulas.

Presentations

When wanting to insert code that has been copied from the Modelica Text layer into Microsoft PowerPoint for presentations, please consider using the command **Paste special** as HTML format in PowerPoint for keeping the formatting from Dymola.

Another alternative is to insert screen shots, of course, but then the text cannot be edited.

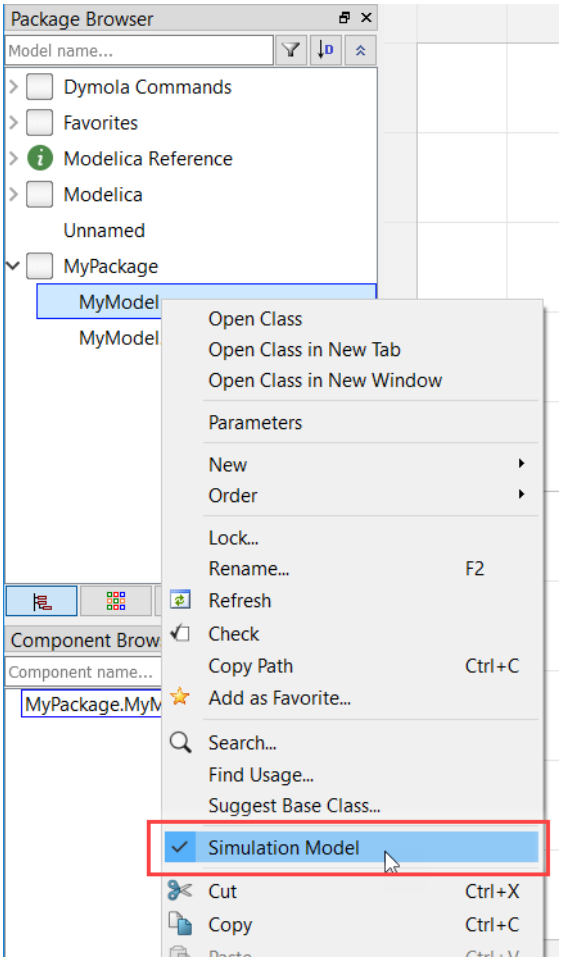
4.2.9 Selection of active simulation model

You can select a certain model that should always be used as active simulation model when giving certain commands (Simulate, Translate, etc.), even if you work with another model.

By default, the active simulation model is the current model in the currently selected tab, but you can do a manual selection of a certain model as the active simulation model.

The active simulation model can be selected in two ways, from the package browser or from the context menu of the model tab that contains the model.

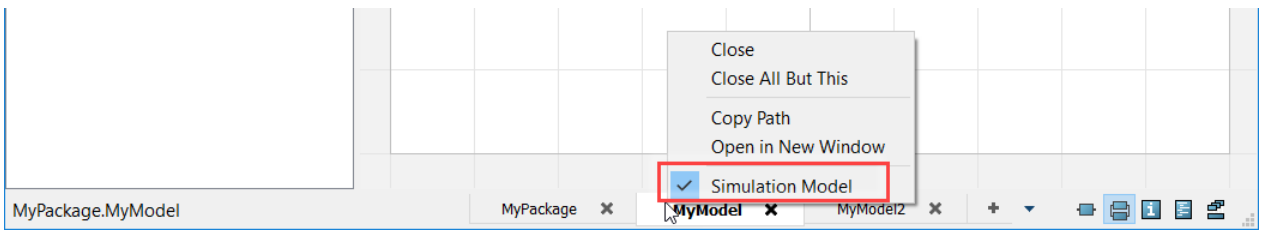
To select a model as active simulation model from the package browser, you can use the command **Simulation Model** in the context command of the model in the package browser:



This menu entry works in toggle mode; by default it is not set – the above figure shows MyModel selected as the active simulation model by the user.

If the selected model is not open, it will be opened in a new model tab (only an open model can be the active simulation model).

To select the active simulation model from the corresponding model tab, right-click the model tab and select **Simulation Model**.



The model name of an active simulation model set by the user is indicated in boldface in the model tab (as above).

If the model tab containing the active model is closed, Dymola will work again in default mode; the model in the currently selected tab becomes active simulation model.

If a model in a model tab has been selected as active simulation model by the user, some operations are not allowed when this tab is selected:

- In this tab, the context menu entry **Open Class** will open the selected class in a new tab, not in the present tab.
- Double-clicking a class in the package browser, or right-clicking and selecting **Open Class** will open the selected class in a new tab, not in the present tab.

For information on what commands are affected when simulating by this selection, etc., see next chapter.

4.2.10 Support for national characters

Dymola supports Unicode. Comments and documentation can be written in your local language, especially supporting Japanese characters in Modelica files.

Modelica allows comments and annotations to use Unicode, but not identifiers and string constants. Dymola fully supports this; and furthermore, in order to support scripting with file names in Unicode, all string constants in Dymola can also contain Unicode characters.

Starting with Dymola 2018, Dymola assumes Modelica files to be UTF-8 encoded, and files are saved UTF-8 encoded. Reading a Modelica file that is not valid UTF-8 encoded will result in warnings and automatic error recovery.

The warnings are displayed in the Syntax Error tab of the Dymola log window. For backwards compatibility, UTF-8 encoding errors are presently just warnings.

Encoding errors are by default handled by automatically transforming erroneous byte sequences to correct UTF-8 multi-bytes, if possible. This applies to outdated ISO/IEC 8859-1:1998 encoded files (commonly mislabeled as ISO-8859-1 or just Latin-1). Such bytes with values greater or equal to 0xA0 and followed by an ASCII character are by default automatically transformed to their respective UTF-8 multi-bytes. Byte sequences that cannot be transformed are removed.

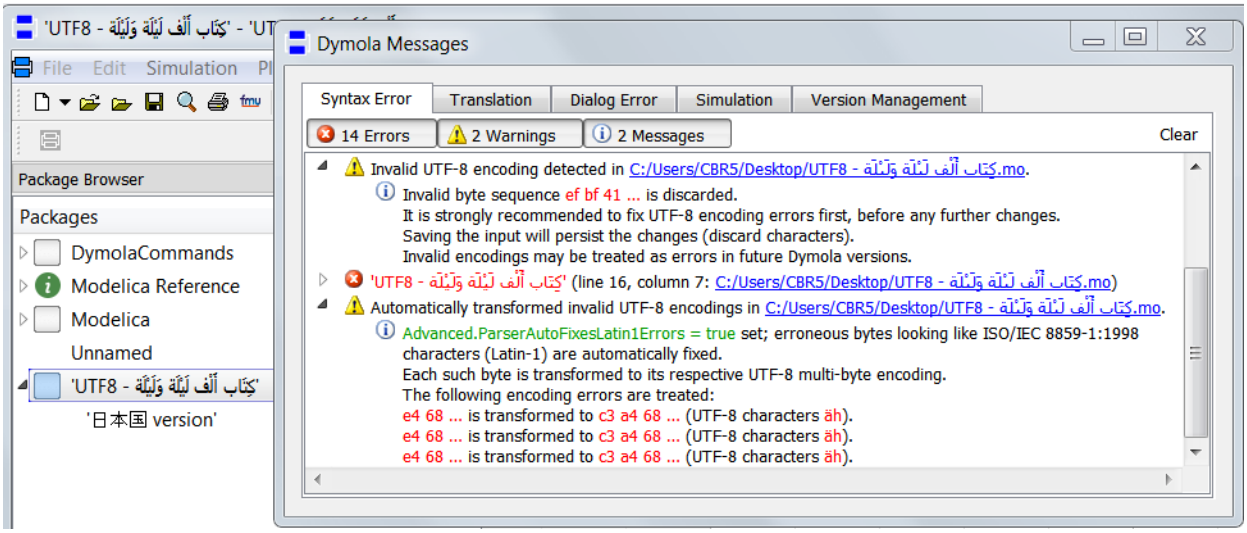
The alternative is to remove all erroneous byte sequences instead of trying to convert them. This can be done by setting the flag

```
Advanced.ParserAutoFixesLatin1Errors = false;
```

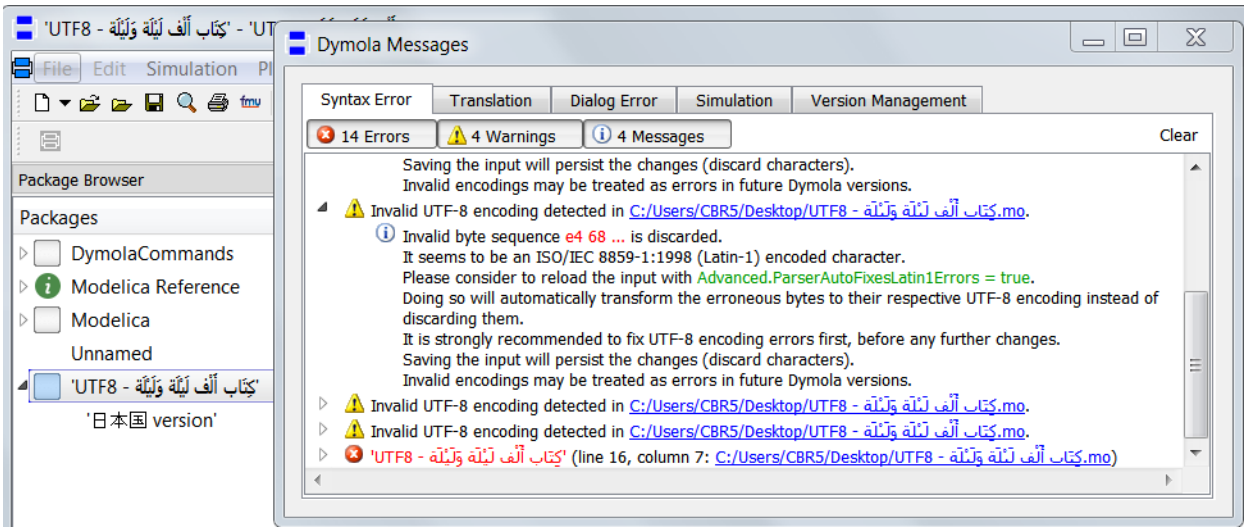
(The flag is by default `true`.)

Note! Since saving a model containing encoding errors will change it according to the alternatives above, it is important to investigate the encoding errors first, before doing any further model changes.

A typical session with auto-corrected and non-correctable encoding errors is:



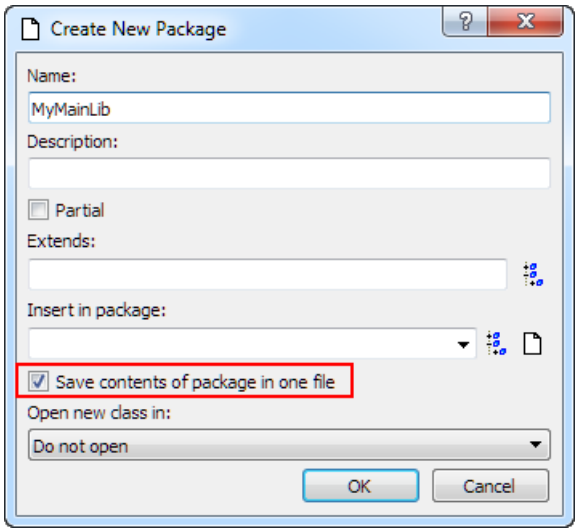
The same session with the flag `Advanced.ParserAutoFixesLatin1Errors` set to `false` is:



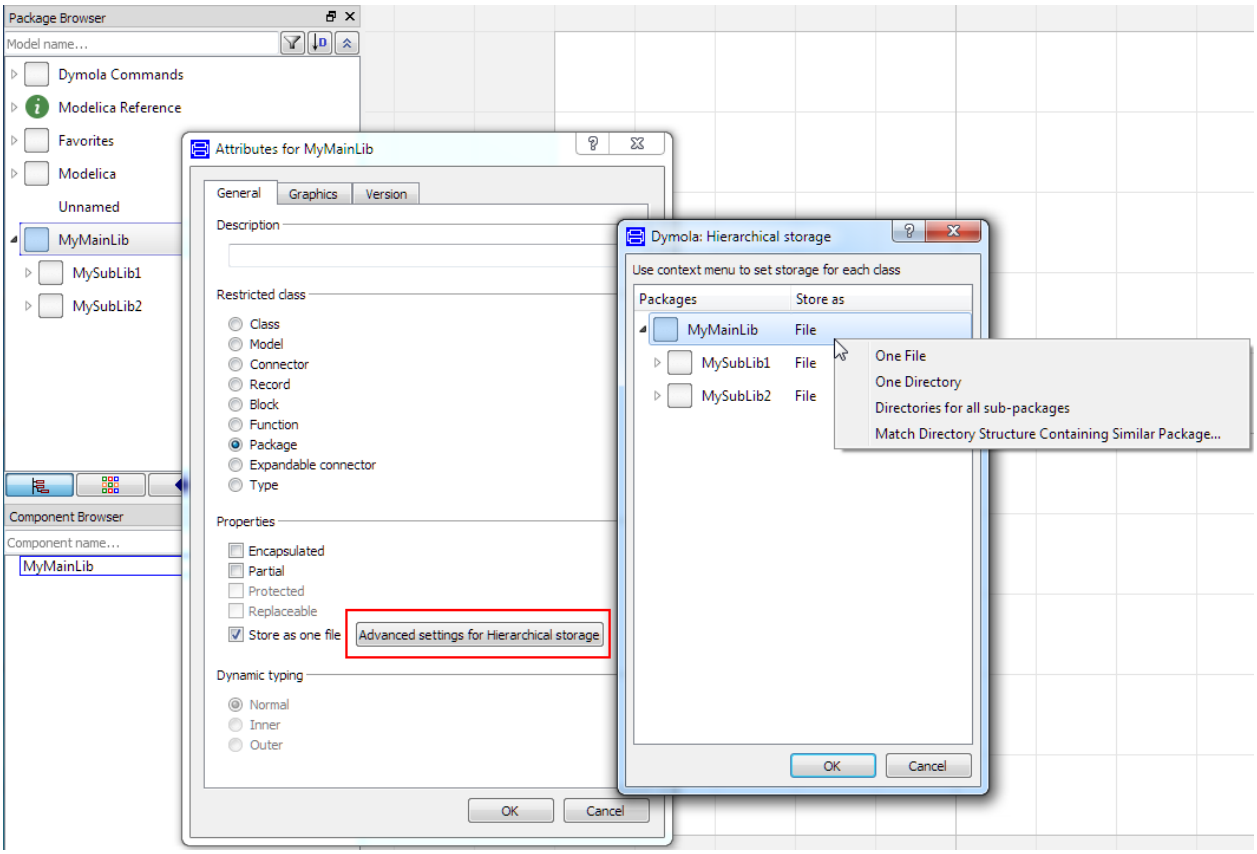
4.3 Advanced model editing

4.3.1 Storing packages hierarchically

When you create a hierarchy of packages, the default setting when creating any package is to save the content of the package as a file, and not as a hierarchy:



However, you can later right-click the package in the package browser and select **Attributes** and then select **Advanced settings for Hierarchical storage** to configure the storage of the package hierarchy. In the dialog that appears, you can right-click any package in the hierarchy and select how it should be saved. An example:



The alternatives are:

- **One File** – This selection corresponds to the default selection of saving the content of the package in one file. **Note** that if a directory is selected to be stored as a file, sub-directories of that directory are also always stored in that file.
- **One Directory** – This selection means to save this package as a directory.
- **Directories for All Sub-packages** – This selection means saving this package, and all sub-packages, as directories.
- **Match Directory Structure Containing Similar Package...** - Selecting this alternative gives you a browser where you can select a `package.mo` file. The idea is that you should select a package.mo file of a library similar to the one you are working with, to use the structure of that selected library for your own new library. The basic case is that your new library is a development of the one that is selected.

The column **Store as** in the dialog displays the current selection.

For more information about what storing as a file or as a directory means, see section “File > New > Package” on page 450.

4.3.2 State Machines

General

State machines according to Modelica Language version 3.3 are supported.

In order to write state machines using Modelica 3.3, please note the following properties:

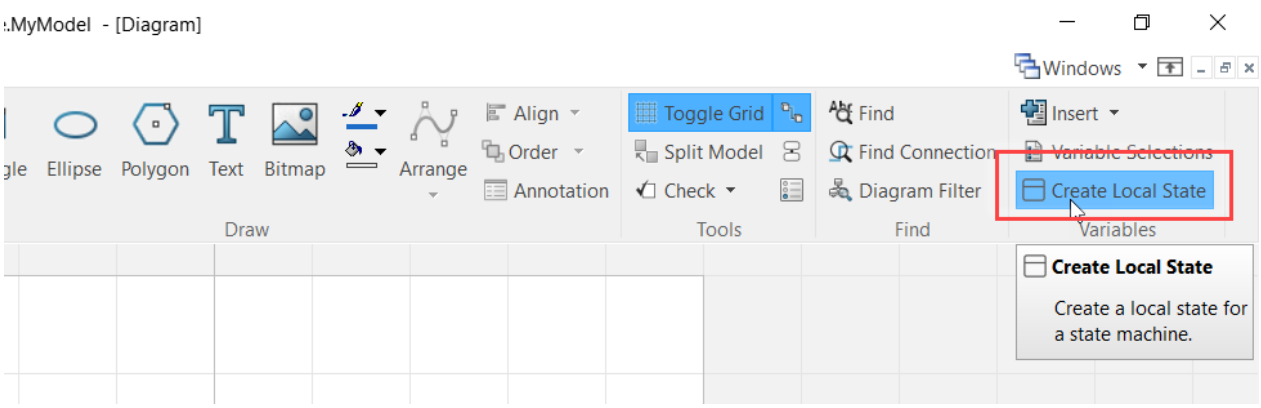
- For variables that are common between different states, the inner/outer concept has to be used, or alternatively graphical connections.
- State machines are clocked.

For more information, including examples what can be done when it comes to e.g. hierarchical state machines, conditional data flow and applications like cruise control, please see the relevant papers available by the command **Tools > Help Documents**, and the tutorial available by the command **File > Demos > Modelica Synchronous Tutorial**.

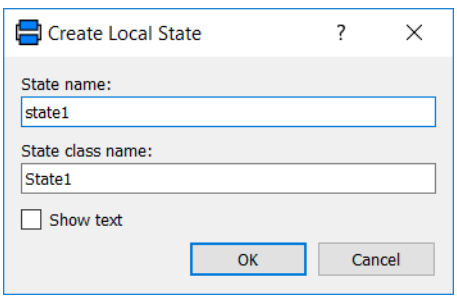
Creating and simulating a simple state machine

The below is a short description of the basic state machine editor features in Dymola; for more information, see above.

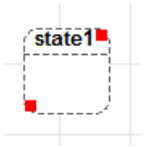
To create states, use, in the **Graphics** tab, the command **Create Local State**:



The command will display a menu:



The state name and state class name can be entered. **Show text** can be selected if the equations in the state should be visible in the step, without having to enter the Modelica Text layer of that step. Pressing **OK** will create the state, but with dashed outline, since it must have a transition to be defined as a state:



It can be dragged and resized like any other object. The step can now be opened by double-clicking it in the package browser, and code can be inserted using the Modelica Text editor.

The name of the state can be changed afterwards by double-clicking the state. To change the **Show text** setting afterwards to display equations, the corresponding annotation in the code

```
annotation(Diagram(graphics={Text(textString="%stateName")}))
```

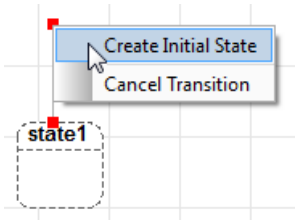
can be changed to

```
annotation(Diagram(graphics={Text(textString="%stateText")}))
```

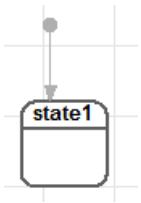


When the state is created, the **Transition Mode** is activated automatically; enabling creation of transitions by dragging, like dragging connections. You start dragging from the border of the state.

The initial state of a state machine is defined by creating a specific transition; dragging a transition from the step above, and right-click after having dragged just as small distance (without connecting to anything) will give:

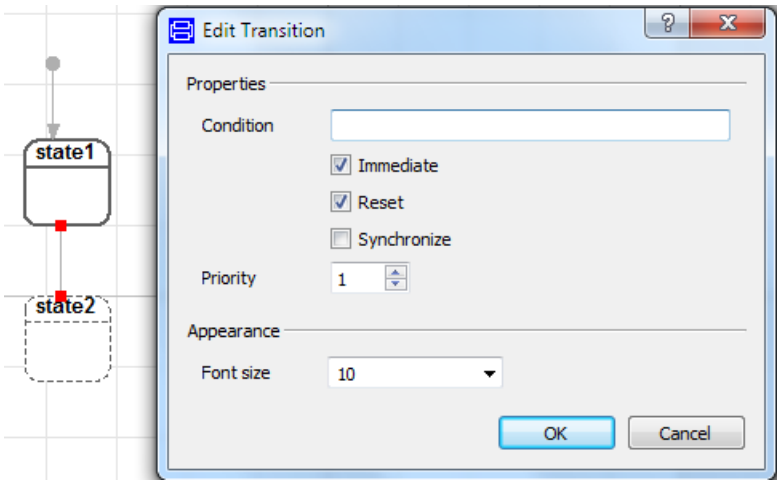


Selecting **Create Initial State** will give:



This state is now initial state, and since it has a transition it is presented as a valid state.

Drawing another state and dragging a transition connection between them, the following menu will be displayed:



The **Condition** is the condition that should be valid for the transition; it must be a Boolean expression.

Immediate should be selected if the transition should fire when `condition=true` (this is called "immediate transition"). If deselected, the transition will fire when `previous(condition)=true` (this is called "delayed transition").

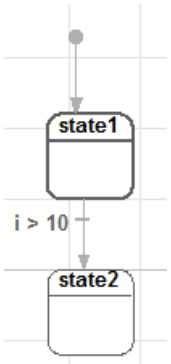
Reset should be selected if the target state (and states included in the target states, if any) should be reinitialized; i.e. included state machines are restarted in initial state and state variables are reset to their start values.

Synchronize should be selected if all state machines within the source state must have reached their final state before the transition condition is tested for.

Priority can be set to an integer corresponding to the priority; 1 is the highest priority.

The font size can be set.

Selecting `i > 10` as condition, and deactivate **Immediate** will give when selecting **OK**:

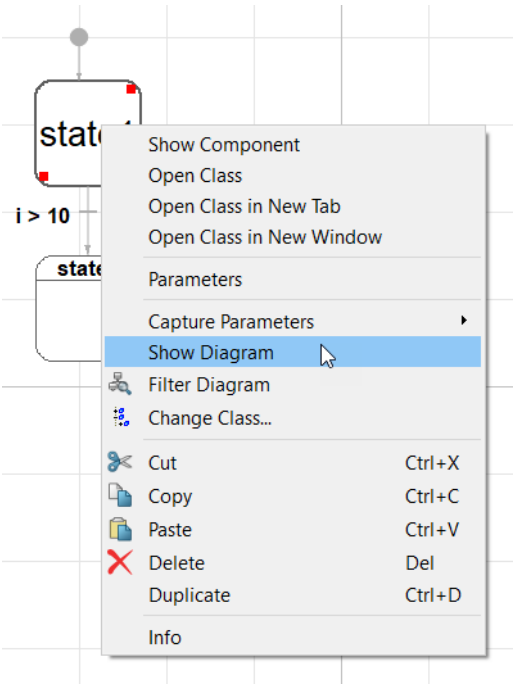


The transition is created as an arrow between the two states. It indicates the above selections

- The arrow direction is from the source state to the destination state.

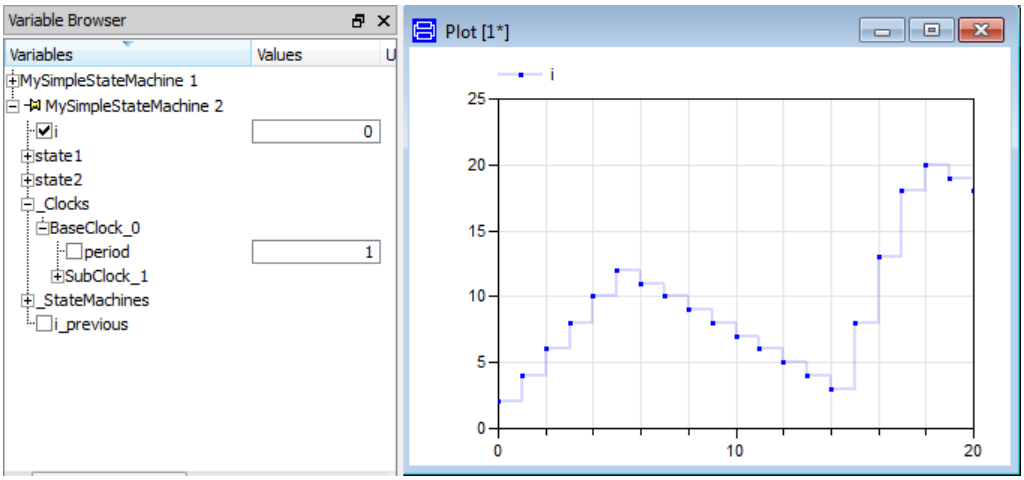
- The perpendicular bar of the transition is close to the destination state for an immediate transition, and close to the source state for a delayed transition.
- The arrow is filled for a reset transition, otherwise non-filled.
- A synchronize transition has an “inverted fork” at the source state.
- Priority is shown preceding the condition if not equal to 1 (e.g. `2: i>1`).
- The transition can be created like any line, with corners that can be dragged; corners can be added and removed etc.
- Double-clicking the transition will display the menu above.

Hierarchical state machines can easily be created by creating state machines in states. Having hierarchical state machines, the annotation `annotation(showDiagram=true)` that is set by default in new states is of importance. The default value is to display state machines inside a step. It is possible to select if enclosed steps should be displayed or not by right-clicking a step and ticking/unticking **Show Diagram** for each individual step:



Important: To enable the above selection, the mentioned annotation must be deleted in all states where this selection should be possible. (It is not sufficient to set the annotation to false, it overrides the menu anyway; the annotation must be deleted.)

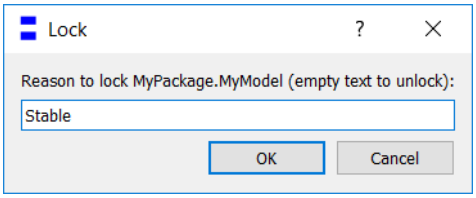
When simulating this model, a message will be displayed that since a clock is not defined; a default base clock with the period 1 second will be used; the period can be set using the variable browser:



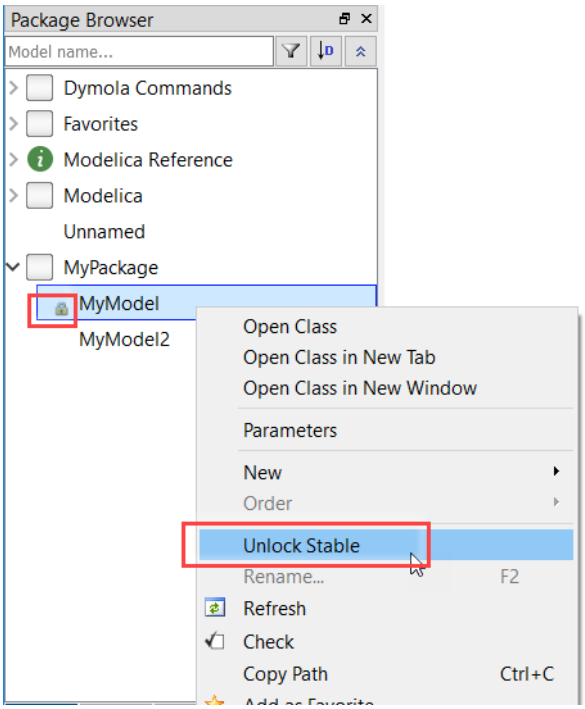
4.3.3 Locking and unlocking classes

You can lock/unlock classes in the package browser; the idea is that locked classes need to be explicitly unlocked for editing.

To lock a class, right-click it in the package browser and select **Lock...**. A dialog where you can specify the reason for locking appears; in this example the text `Stable` is entered:



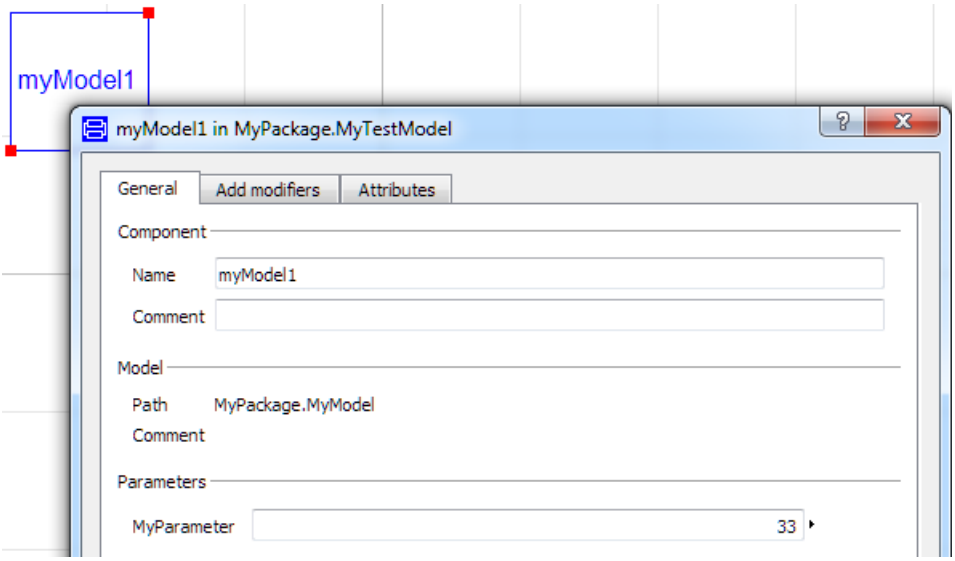
Clicking OK makes the class read-only; this is indicated in the window heading of the class and with a hanglock on the class in the package browser (see image below). Right-clicking the class again gives:



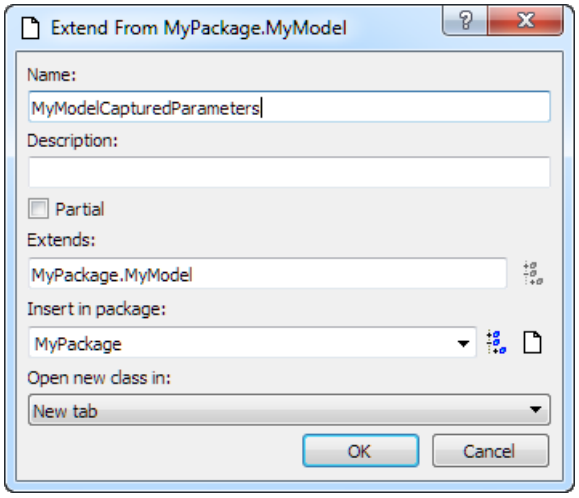
4.3.4 Creating a new model from a component with parameter values included

You can create a new model from a component by the command **Capture Parameters > As Model**. This command is available both when right-clicking a component in the diagram layer and right-clicking the component in the component browser. An important feature when using this command is that the parameter values given to the component by the parameter dialog, as well as other modifiers, are included in the favorite model that is created by extending. (For more information about extending in general, see “Creating a model by extending from an existing model” on page 232.)

As a minor example, consider the following model MyModel with just one parameter MyParameter with no default value, however given the value 33 by using the parameter dialog:



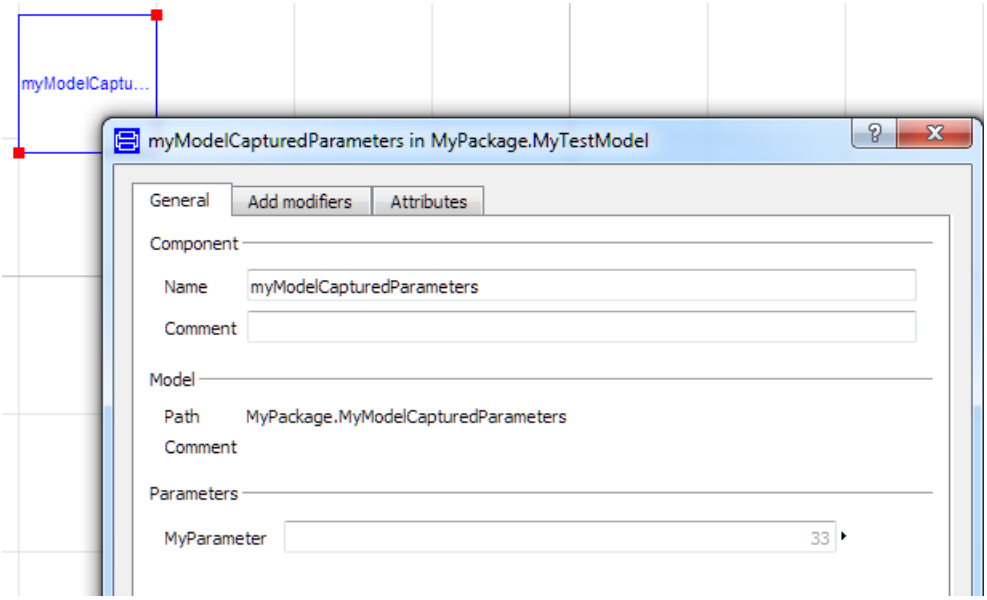
Right-clicking this model in the diagram or the component browser and selecting **Capture Parameters > As Model**, gives the dialog for extending the model of the selected component. Giving the new model the name MyModelCapturedParameters in the dialog:



The code of the model will be:

```
model MyModelCapturedParameters
  extends MyModel(MyParameter=33);
end MyModelCapturedParameters;
```

Instantiating the new model in the diagram and looking at the parameter dialog gives:



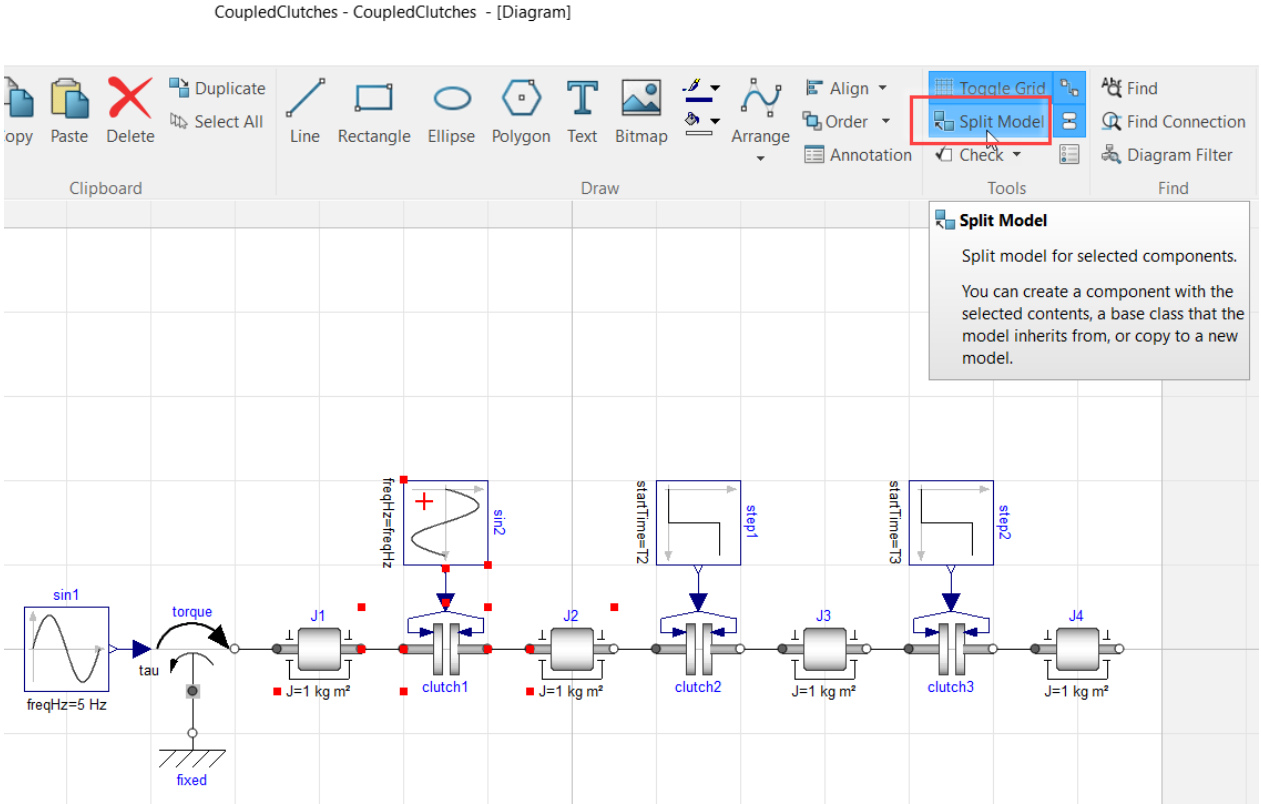
The value 33 is now a default value in the new model.

4.3.5 Splitting models

A submodel, base class, or model can be created by the **Split Model** command.

Submodel creation

A submodel can be created by selecting components in a model, and, on the **Graphics** tab, in the **Tools** group, clicking the command **Split Model**. (To be able to repeat the following example, open the demo Coupled Clutches, right-click the model in the package browser and select **New > Duplicate Class...**, and click **OK** in the dialog. This will create an editable model.)



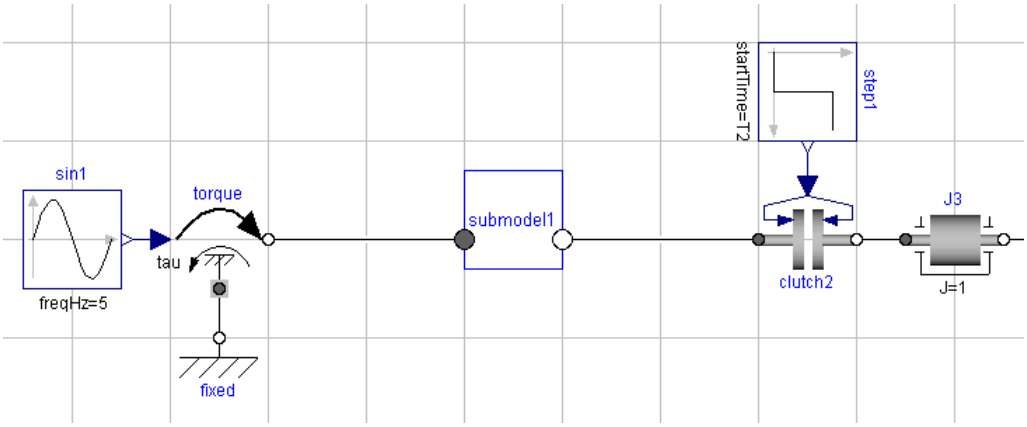
The selected components will be moved to a new class that is created automatically. That class will then be instantiated in the place of the selected components in the original model.

Connections between components in the new instantiated class and the other components in the initial model are restored. (Connectors are automatically added to the new class.)

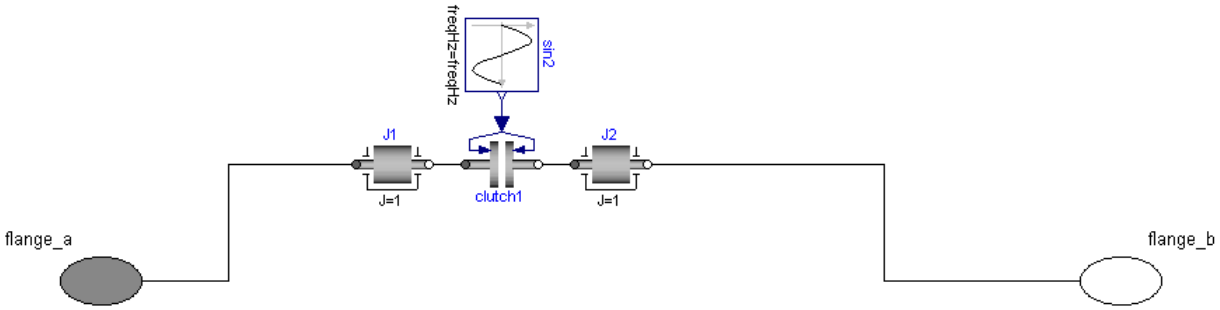
Parameters used in the components are copied to the subsystem when they (or components of them) are used ("=p" and "=p.x"), as well as when used in arrays/matrices/and simple expressions.

Inner/outer components are copied to the subsystem, this is similar to the parameter handling. However, for this case of creating submodels, inner/outer components are, if necessary, changed to outer components without modifiers.

The visual result (keeping the default settings) will be:



Right-clicking the submodel and using the context command **Show Component** will display:



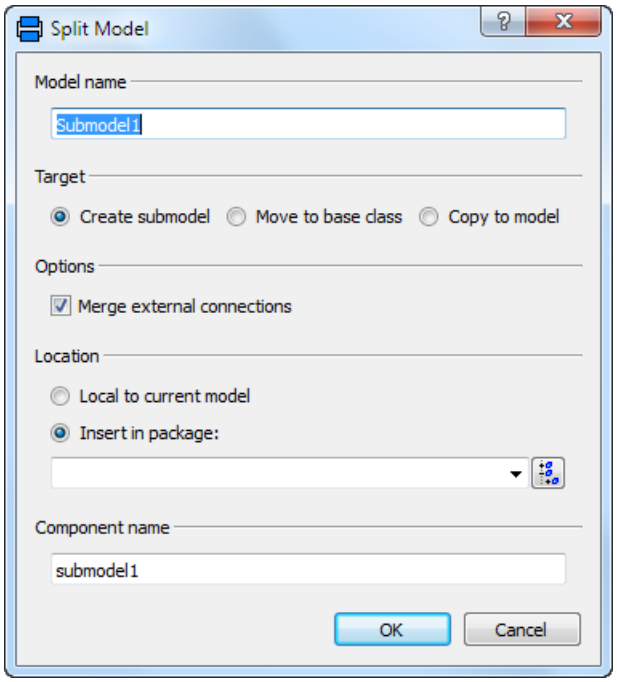
Connections between selected components are automatically included in the submodel.

Options when creating the submodel

There are two uses of submodels:

- Visual simplification of the original model.
- Reuse of the submodel in another model.

The dialog box that is the result of the **Split Model** command is:

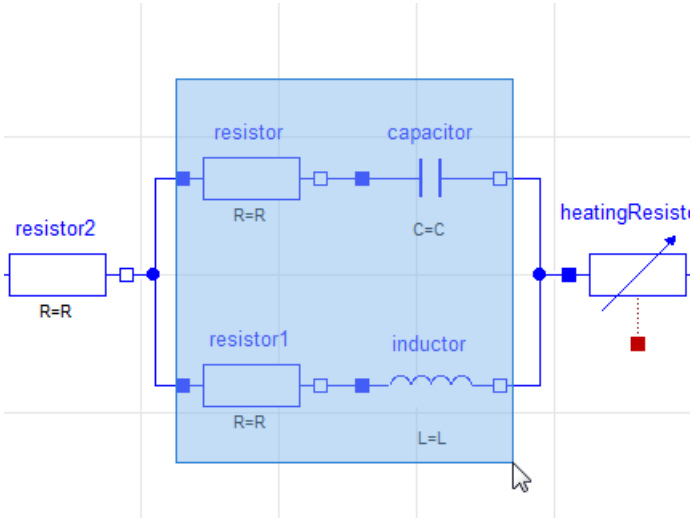


By default the submodel (class) created is public, and the user should select a package where the new submodel will be stored. Selecting **Local to current model** will store the submodel as a protected class in the package of the original model.

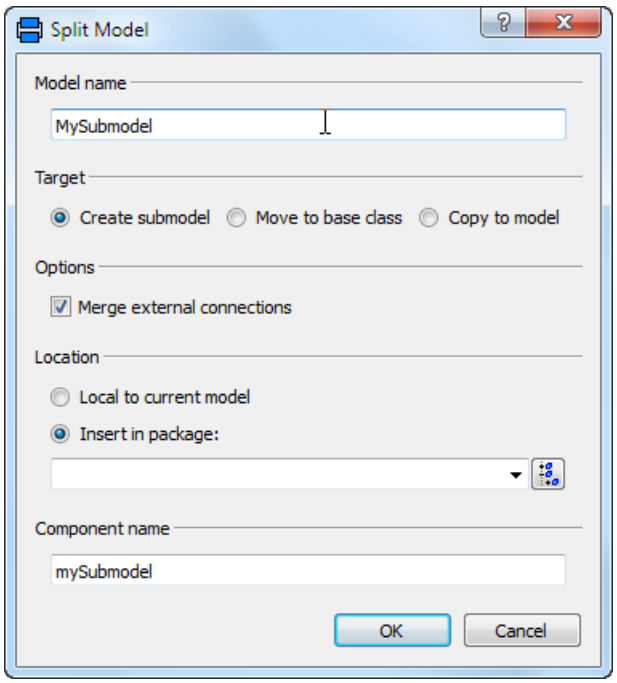
If the user changes the **Model name**, the **Component name** will automatically be changed simultaneously; the difference being that the component name will start with a lower-case letter. If the user changes the Component name, there is no automatic change of the Model name.

It is possible to merge external connections if a submodel is created. This increases the simplification of a diagram.

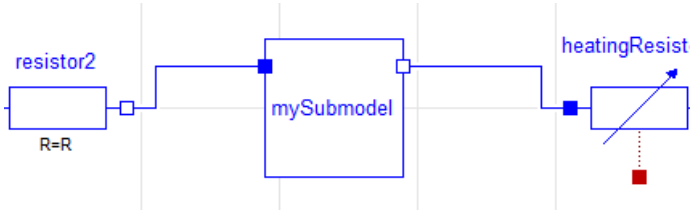
Having as a starting point selected the following:



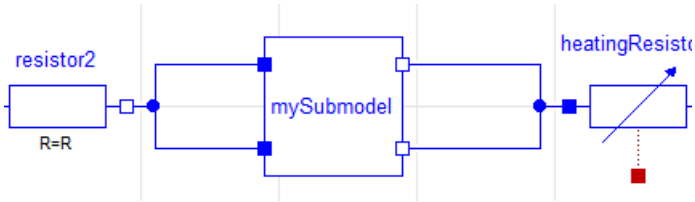
Selecting **Split Model** on the **Graphics** tab, the following menu will appear:



Keeping the default setting of **Merge external connections** will give the result:

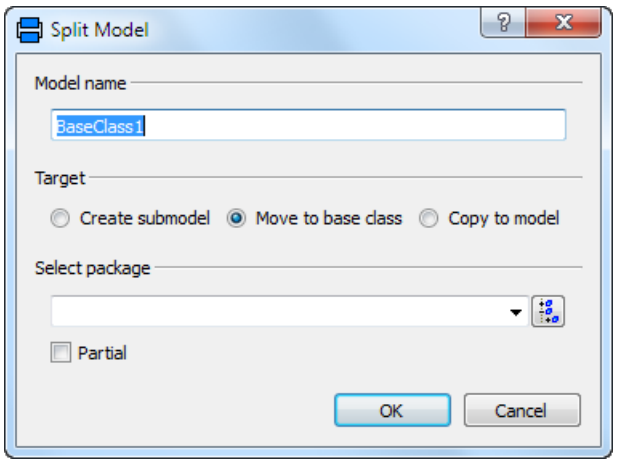


Unticking the setting will give:



Base class creation

If **Move to base class** is selected in the basic dialog above, the dialog will change to:



The selected components will be moved to a new base class that is created automatically. That class will then be extended to the original model. There will be no visual change of the original model. Connections between the extended base class and the other components in the initial model are restored. (Connectors are automatically added.)

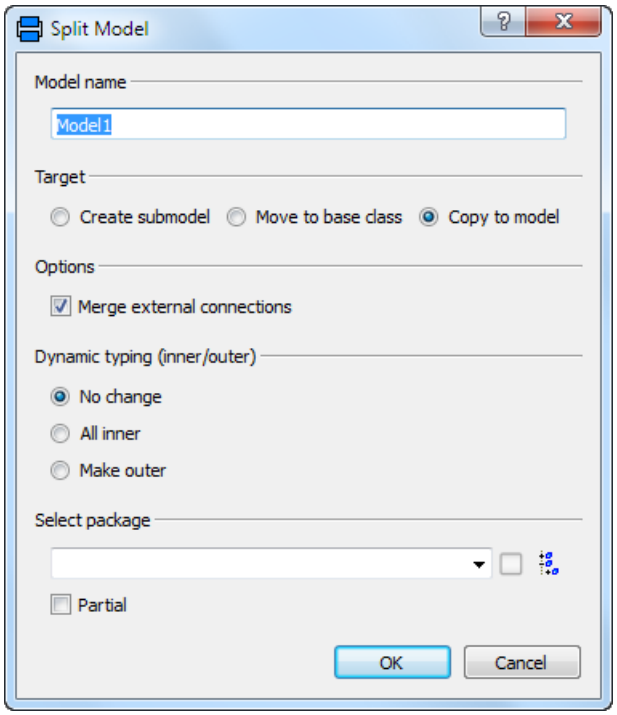
Parameters used in the components are copied to the subsystem when they (or components of them) are used (“=p” and “=p.x”), as well as when used in arrays/matrices/and simple expressions.

Inner/outer components are copied to the subsystem, this is similar to the parameter handling.

You can select the destination and if the base class should be partial.

Model creation

If **Copy to model** is selected in the basic dialog, the dialog will change to:



The selected components will be copied to a new model, in the selected package. The model can be made partial. There will be no change at all the original model – note that is this case the components are copied, not moved.

Parameters used in the components are copied to the new model when they (or components of them) are used ("=p" and "=p.x"), as well as when used in arrays/matrices/and simple expressions.

The **Dynamic Typing (inner/outer)** alternatives are:

- **No change** (default). No special treatment of inner/outer components. They are copied to the new model; this is similar to the parameter handling.
- **All inner**. Intended to make a model that can be run directly. All inner components visible from the selected subcomponents are included.
- **Make outer**. Intended to make a reusable subsystem. Any used inner component is replaced by a corresponding outer component.

The dynamic typing choice is remembered between calls, and is also available as the flag `Advanced.SplitModelDynamicVariant`.

Concerning moving external connections, see the section about creating submodels above.

4.3.6 Components and connectors

Nested connectors, overlapping (stacked) connectors, expandable connectors and array of connectors are supported.

Inserting a component/connector by dragging it to the diagram layer of an edit window is presented in corresponding sub-section in the section “Basic model editing”.

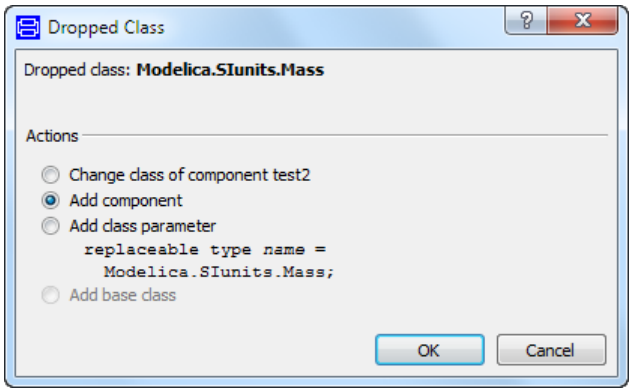
Inserting by dragging it to the Modelica Text layer of an edit window

It is also possible to insert a name into the Modelica text editor by dragging from the package browser or a library window. This can for example be used to declare variables of types from Modelica.Units.SI or functions from Modelica.Math.

Inserting by dragging it to the component browser

Sometimes it is convenient to insert a component by dragging it from either the package browser or a library window to the component browser. Components of type, block, class and model can be added this way.

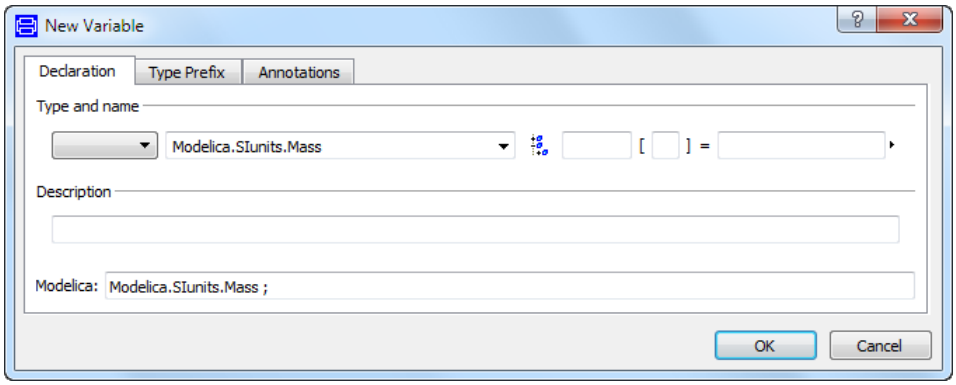
When dragging a component from the package browser on top of a component in the component browser, a **Change class of component** action (see below) is performed by default if possible. By dragging it between components, or if the default action is not possible, the following menu appears:



The following choices might be possible:

Change class of component *componentname* By dropping the component on top of a top-level component name, the class of that component will be changed to the class of the component that is dropped on it. (In this case, the component `test2` will be of class `Modelica.Units.SI.Mass` if this command is executed). If modifiers or connections no longer match, you will be asked to confirm the change, and in that case, they are automatically removed. (Of course, also the graphics is affected when replacing a class.)

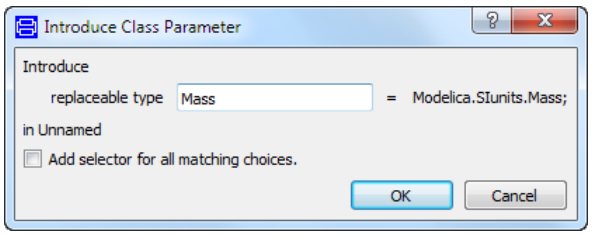
Add component Components of type, block, class and model can be added this way. If this alternative is selected, a new menu where variables can be declared will follow when **OK** is pressed. In the case of “mass” it will look the following:



Here the variable type and name can be declared. Also a description can be entered. For more information about declaring parameters etc (and this menu) please see section “Parameters, variables and constants” starting on page 376.

Please note that this way of inserting components is *not* suitable for components containing graphics, since the graphics is not transferred!

Add class parameter Class parameters are local replaceable classes. They can be changed for some components. If that is possible, this entry will produce the following menu:



An application example could be building a car model, where the class `Wheel` is suitable to implement as a local replaceable class (class parameter).

By ticking the setting **Add selector for all matching choices** a list of all matching choices is automatically created when the replaceable component is displayed as a modifier in the parameter dialog of an enclosing class. This list is available as a drop-down list in the modifier field.

Add base class. This will add a new base class. When this selection is possible to select, multiple base-classes are legal.

Working with replaceable components



A replaceable component is a component declared replaceable (by, for example, selecting it and, on the Graphics tab, in the Tools group, clicking the command **Attributes**.).

Presentation in the diagram layer

For information on how replaceable and redeclared components are presented in the diagram layer, see section “Replaceable, redeclared and inherited components” starting on page 364.

Redeclaring (replacing) one replaceable component

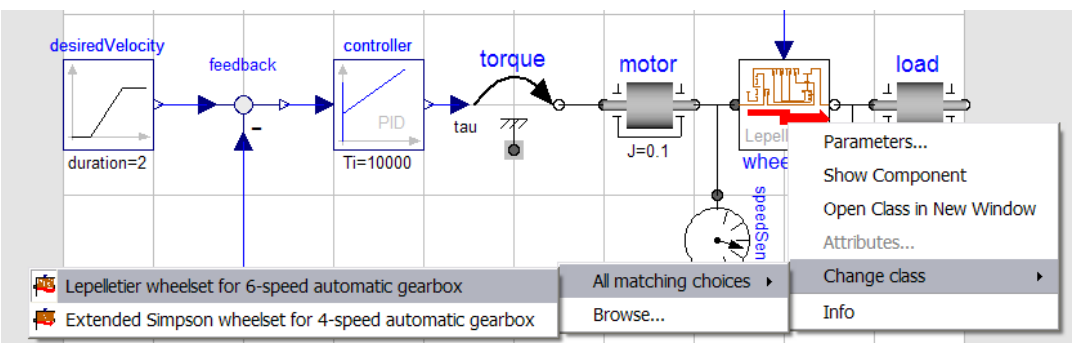
Please note that this section is closely related to the section “Building templates using replaceable components” on page 369. Please also see that section.

There are two ways of replacing **one** replaceable component.

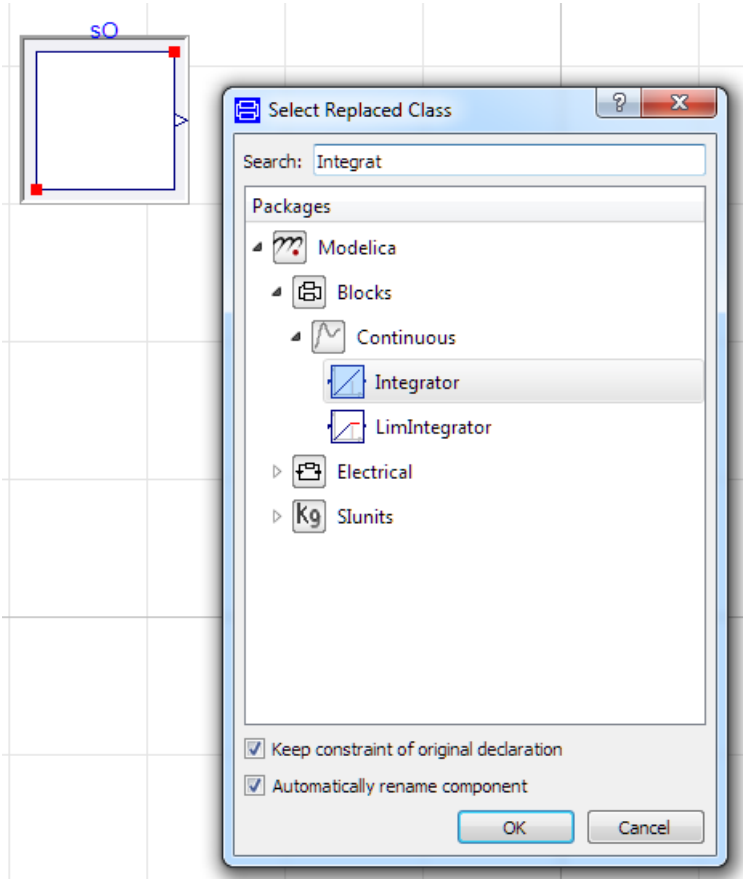
One way is by dragging a component of another class from the package browser and placing it on top of the component that should be replaced in the component browser. Please see the command **Change class of component** above.

The other way is by using the context menu. This is done by selecting the component that should be replaced (redeclared), right-click and select **Change Class > All Matching Choices**. Selecting this alternative, the menu shows a list of matching choices, where the icon of the current choice is shown depressed.

The resulting window will show the parameter dialog of the redeclared class (below for Lepellitier) instead of the original class. The Modelica Text layer and the documentation layer show the text of the redeclared class.



Selecting the menu choice **Change Class > Browse...** instead displays a normal package browser. Let us look at a simple example of this. Drag in a `Modelica.Blocks.Interfaces.SO` to the diagram layer and accept it being replaceable. Right-click it and select **Change Class > Browse...** and search in this example for Integrator to replace it. The menu will look like this:



Note the two options:

- **Keep constraint of original declaration** In this case it means that the new declaration will have "constrained by Modelica.Blocks.Interfaces.SO" on the declaration.
- **Automatically rename component** This automatic renaming is intended for the common case when the component has the default name from when it was instantiated, and it is only activated in this case. The new name will be the default name of the class.

(This way of replacing a component can be used if the component is not replaceable as well; the **Change Class** context command will display the browser. However, in that case only the last one of the two options above can be seen.)

Replacing (redeclaring) several replaceable components at the same time

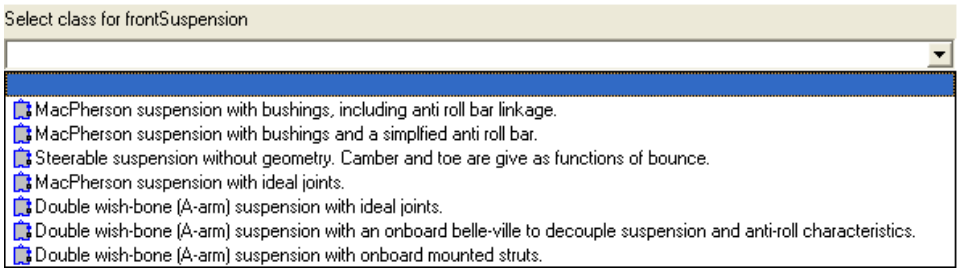
To replace (redeclare) more than one component at a time, you can select several components of the same class; right-click and use the context menu command **Change Class**.

Another way to replace (redeclare) a number of components in a convenient way in the same time, without having to select them one by one, is the following. Take up the context menu

in the enclosing component (“one level higher”). In the figure above one way is to take up the context menu with no component selected.

In such case, the context menu entry **Parameters…** should first be selected. The parameter dialog will be displayed, containing among other things all replaceable (sub-)components. Now each replaceable component can be replaced by taking up the context menu for the line corresponding to the component and selecting **Select Class.** This will display a class selector for redeclared components available. All matching classes are listed.

Choices for select class.



Replacing (redeclaring) a component already replaced (redeclared)

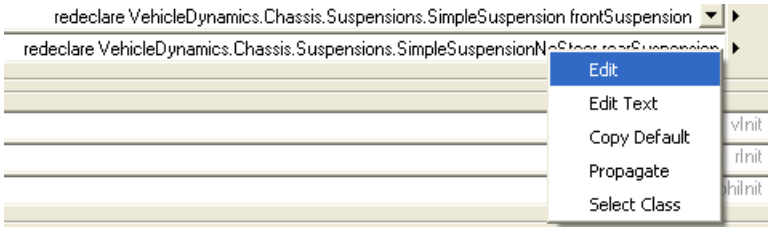
Please note (advanced users) that in the normal case a component that has been replaced (redeclared) cannot be replaced again. (That is, when the model in which the already replaced (redeclared) component is located, is being extended from or being used as a sub-component. As an example, a template can be extended from and a replaceable component can then be replaced, but another extent will make that component impossible to replace.) In some cases, the user might want another behavior. In that case, it is possible to set that all choices generated by `choicesAllMatching` and `choicesFromPackage` are replaceable:

```
Advanced.AutomaticChoicesAreReplaceable=true;
```

Editing of the re-declaration modifier in the parameter dialog

The context menu for a parameter in the parameter dialog for a replaceable component allows editing of the re-declaration modifier.

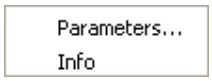
Editing parameters of re-declaration.



This gives the parameter dialog for `frontSuspension`.

Modifying and testing parameter values of a replaceable read-only component

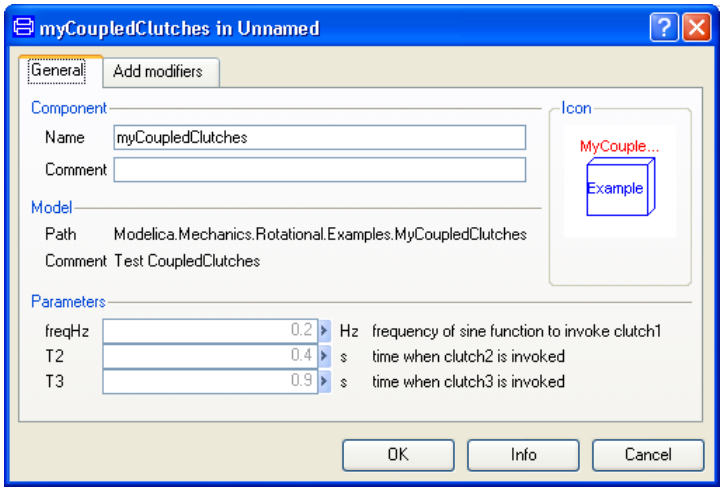
To modify and test the parameter values of a component of a read-only class the class can be extended. A component of the extended class can be manipulated using the parameter dialog.



The parameter dialog is also available for **inherited parameters** of the shown model using the context menu in the diagram layer if no component is selected.

This is also enabled at root model, provided the model solely extends from other models and contains no declarations of its own. For instance if extending from Modelica.Mechanics.Rotational.Examples.CoupledClutches (Please see item 1. below for menu choice):

Inherited parameters.



Semantics: Parameter changes of the root model are stored as modifiers on the corresponding extends-clause(s). Parameter changes for non-root models work in the same way if you use **Show component** and then **Parameters** (with no selected component) to bring up the parameters for the component or directly use **Parameters** for the component.

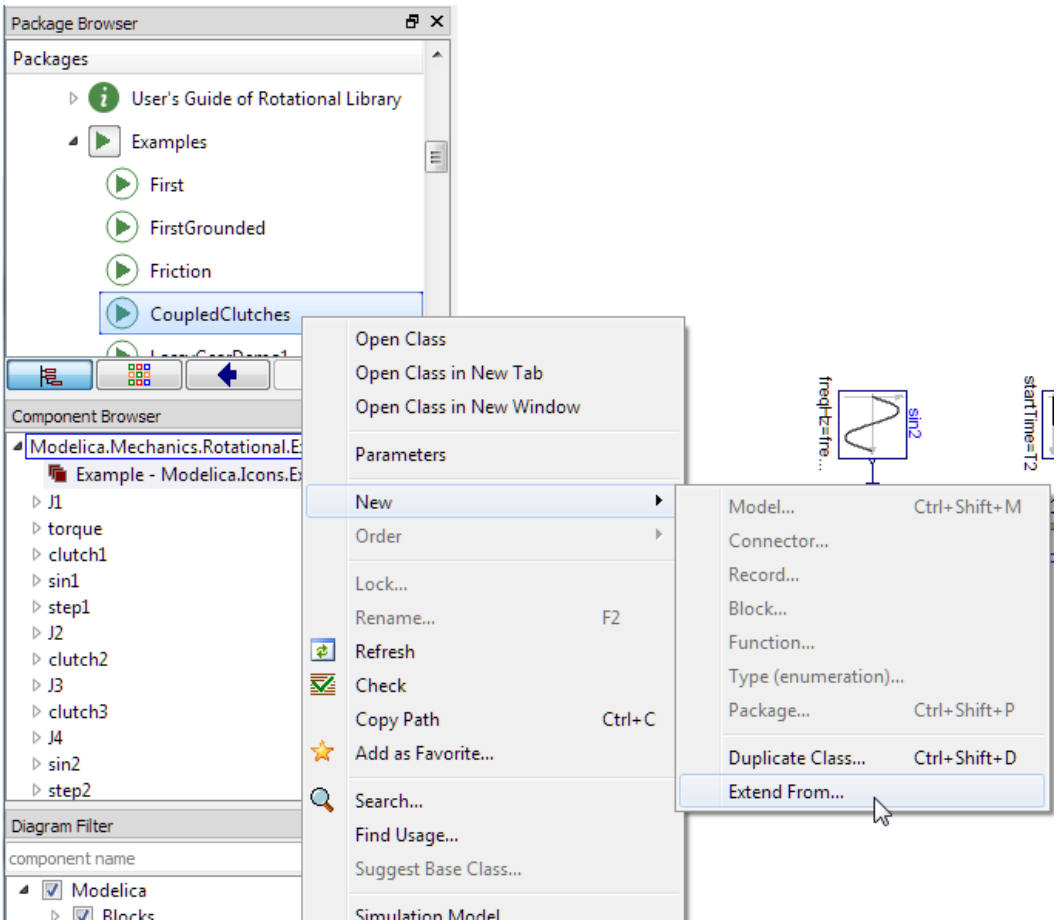
To experiment with a model such as CoupledClutches you can follow these steps. The experimental changes are then stored in a model.

Note: You can also edit parameters after translation in the variable browser. Such changes are not stored in the model, but you can store them in a script using **Script Editor > Generate Script** and then ticking the check-box **Variables**.

Example:

1. Extend from the model CoupledClutches using the context menu on CoupledClutches in the package browser, and give it a name.

Extending a model.



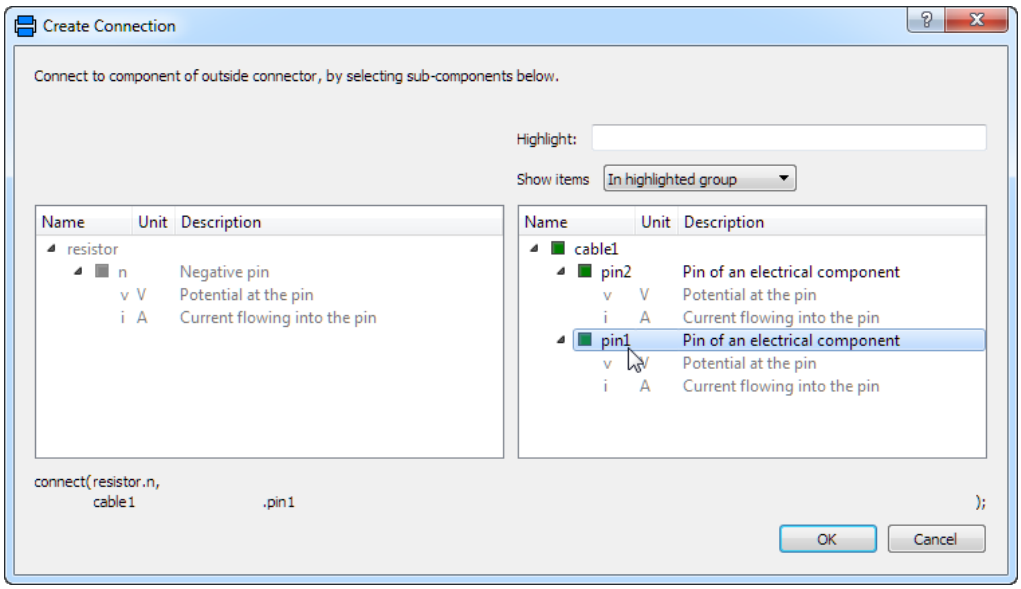
2. Instantiate the class by e.g. dragging a component from the package browser to the diagram layer of an edit window.
3. In the diagram layer select a component, several components, or no component and right-click to bring up the context menu, select **Parameters…** and modify the parameter values using the parameter dialog (see next section).
4. Simulate. The model is translated if necessary.
5. Repeat from step 3.

Handling of nested connectors, overlapping connectors, expandable connectors and array of connectors etc.

Nested connectors

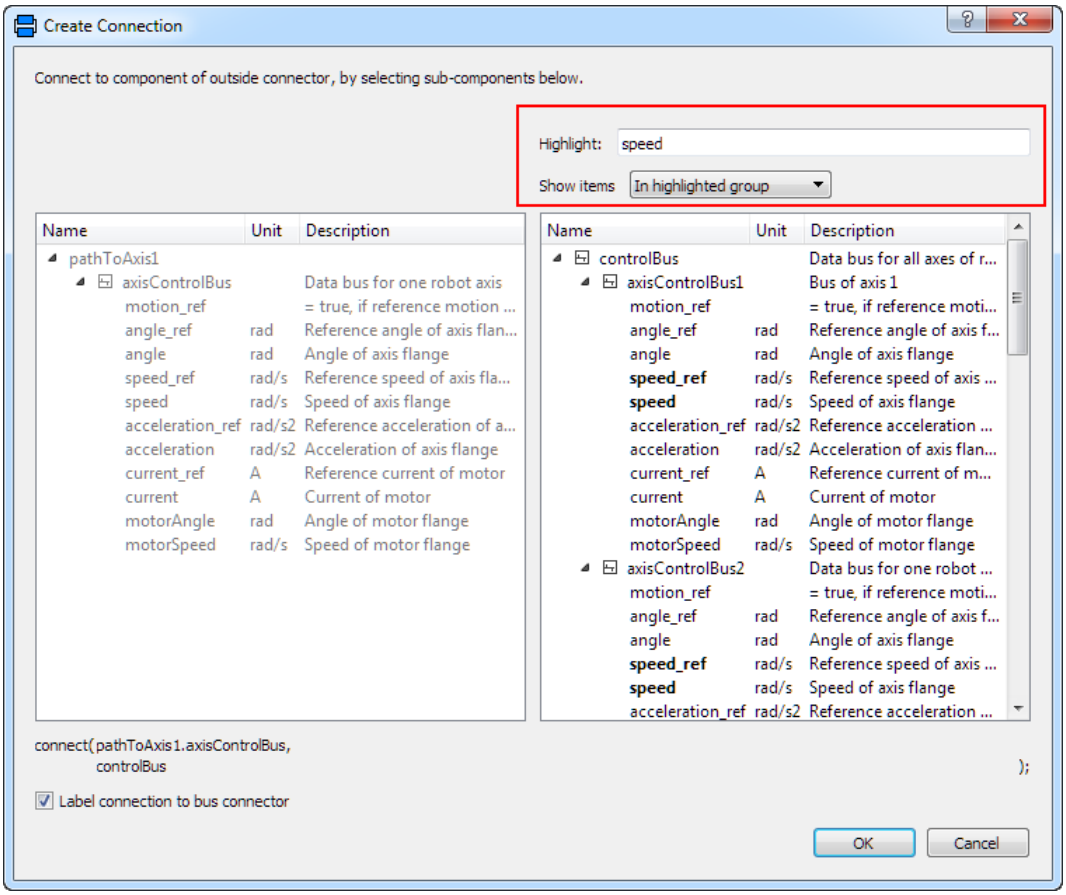
When a connection between incompatible connectors is attempted, it is checked if one of them has nested connectors. If so, a dialog for selecting a sub-connector is presented.

Nested connector dialog.



In this case, a connection from the connector resistor.n to any of the two nested connectors (sub-connectors) in cable1 is possible. The one that should be used is selected by clicking on it. The resulting connection that will be the result if **OK** is pressed is shown in the lower part of the window.

Note that search option is added in the connection dialog; looking at a larger example:



What is entered in the **Highlight** field is searched for in the pane to the right, and hits are highlighted. Nodes are expanded to show the found occurrences.

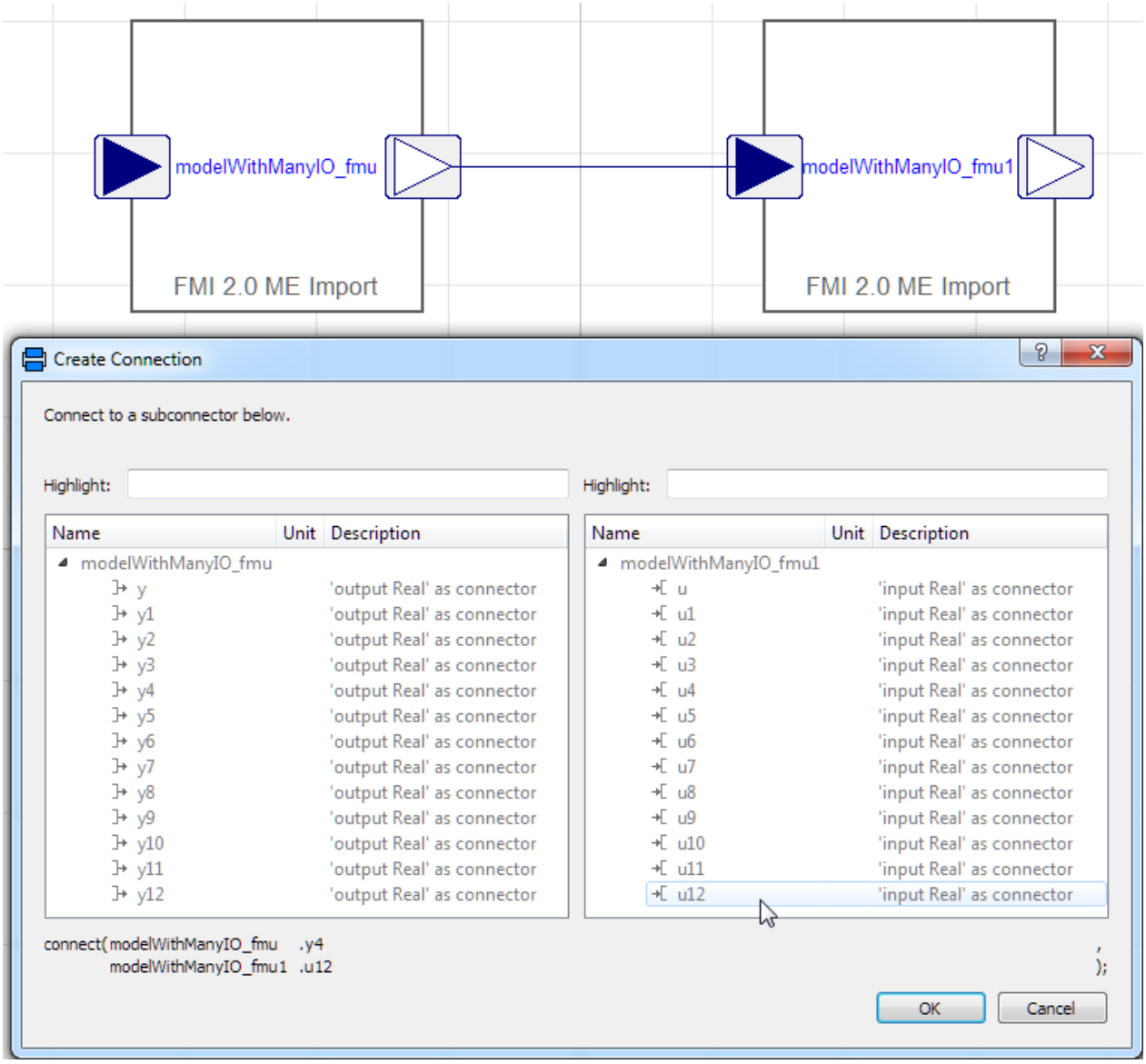
You can select what should be displayed by using the **Show items** dropdown list. The alternatives are:

- **All** displays all items.
- **In highlighted group** displays only the variables in the groups where the hits are.
- **Only highlighted** displays only the hits and the path to them.
- **Filter instead of highlight** is similar to the above, but filters instead of highlights.

Overlapping (stacked) connectors

The connection dialog support overlapping (stacked) connectors.

Below is an example how to use this feature when connecting FMUs with stacked connectors.



The figure shows the dialog that is shown when dragging a connection from the Real output group of one instance of the FMU to the Real input group of another instance of the FMU, and having selected (by the drop-down list) to connect the connector y4 in the Real output group of the FMU to the left, to connector u12 in the Real input group of FMU to the right.

The connection is done when clicking **OK**.

Note 1: Only overlapping connectors of the same parent component are supported by this feature.

Note 2: The feature “Smart Connect” is not supported when using this feature.

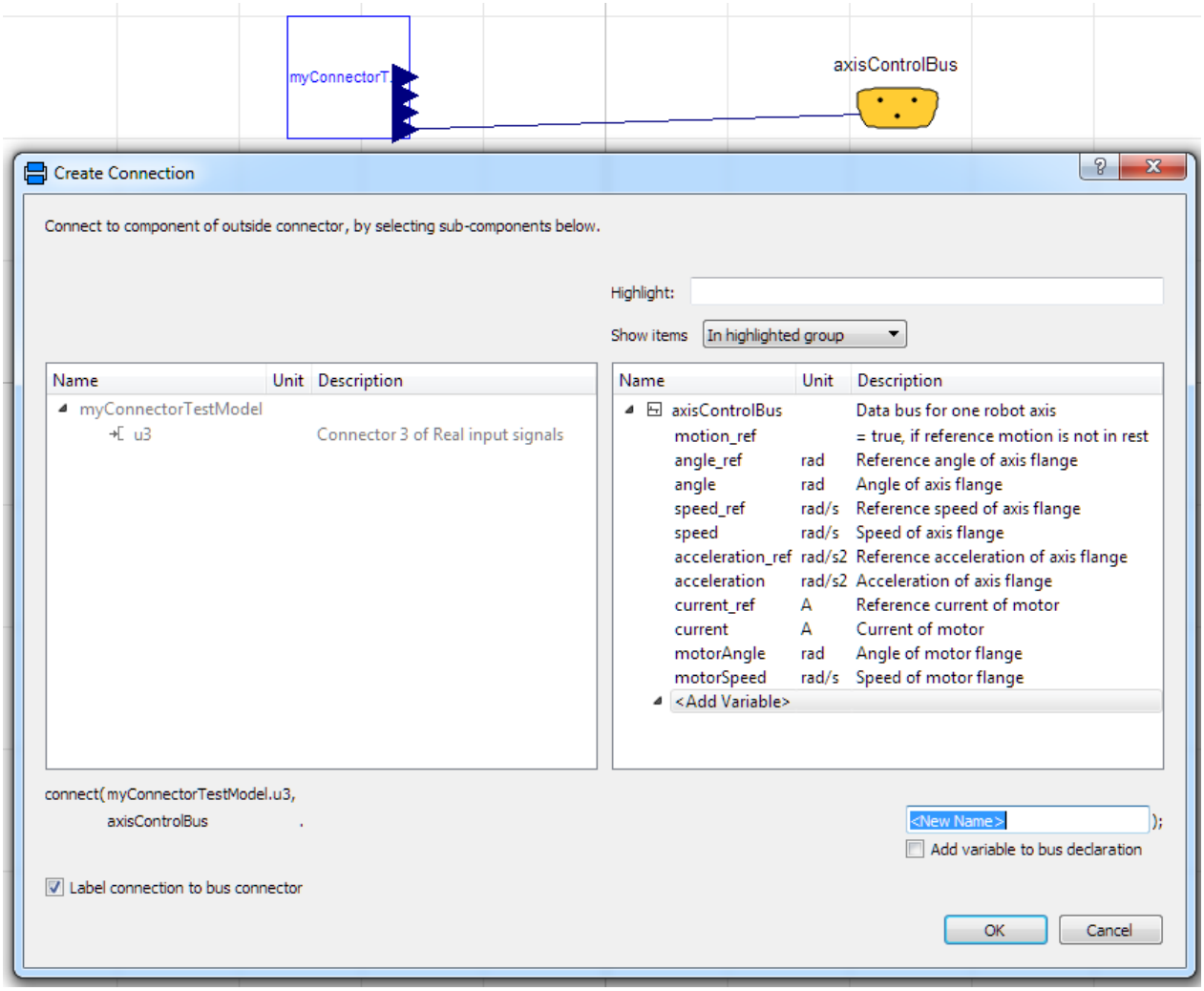
Note 3: The example is simplified, usually more groups are present.

Note 4: Search option is available for both panes, for description see previous section.

Expandable connectors

Dymola has support for expandable connectors as defined in Modelica Language. Components of expandable connectors are automatically treated as connectors.

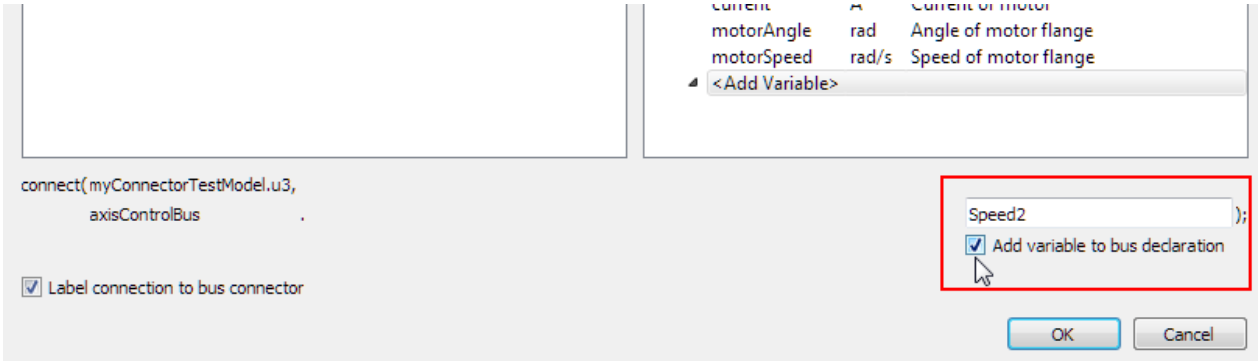
In the example below we have connected from the output u3 of a test model to the expandable connector axisControlBus. We are then given a choice of what variable in the axisControlBus to connect to, but we can also use the **<Add Variable>** alternative to either browse for existing connections to axisControlBus in other models (by clicking the triangle in front of **<Add Variable>**), or construct a new variable by clicking **<Add Variable>** and enter a new name in the box to the lower right. If we use the later, the menu will look like:



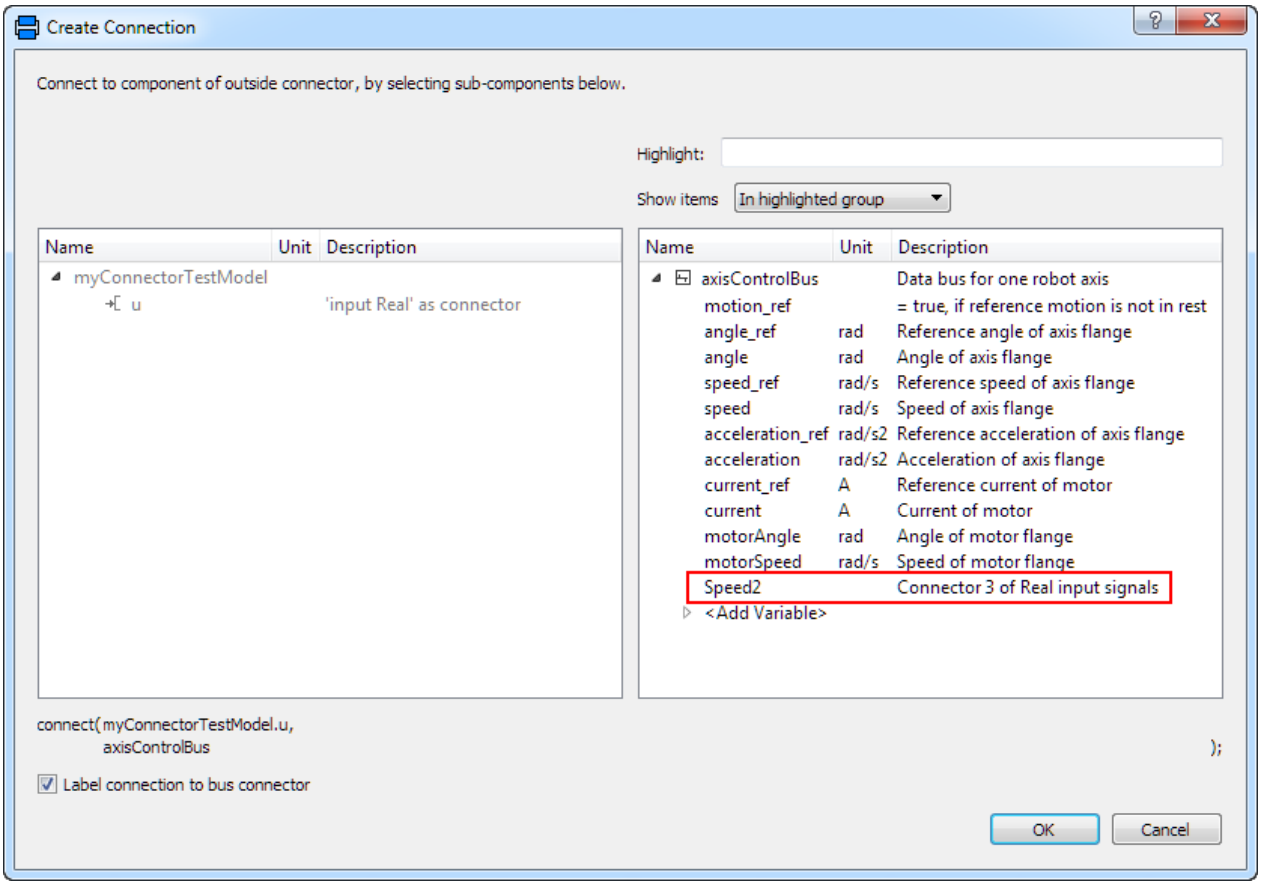
Note in general that additional information might be given in the dialog (not present in the example above):

- Items in italics are already connected (it is unlikely that you want to connect to them again).
- Struck out items will generate errors if you attempt to connect to them.

The option **Add variable to bus declaration** enables you to add the new variable to the general bus declaration. As an example, enter Speed2 and enable this option:



The next time this dialog is used, for example, when connecting to another instance, the new variable is presented in the bus:

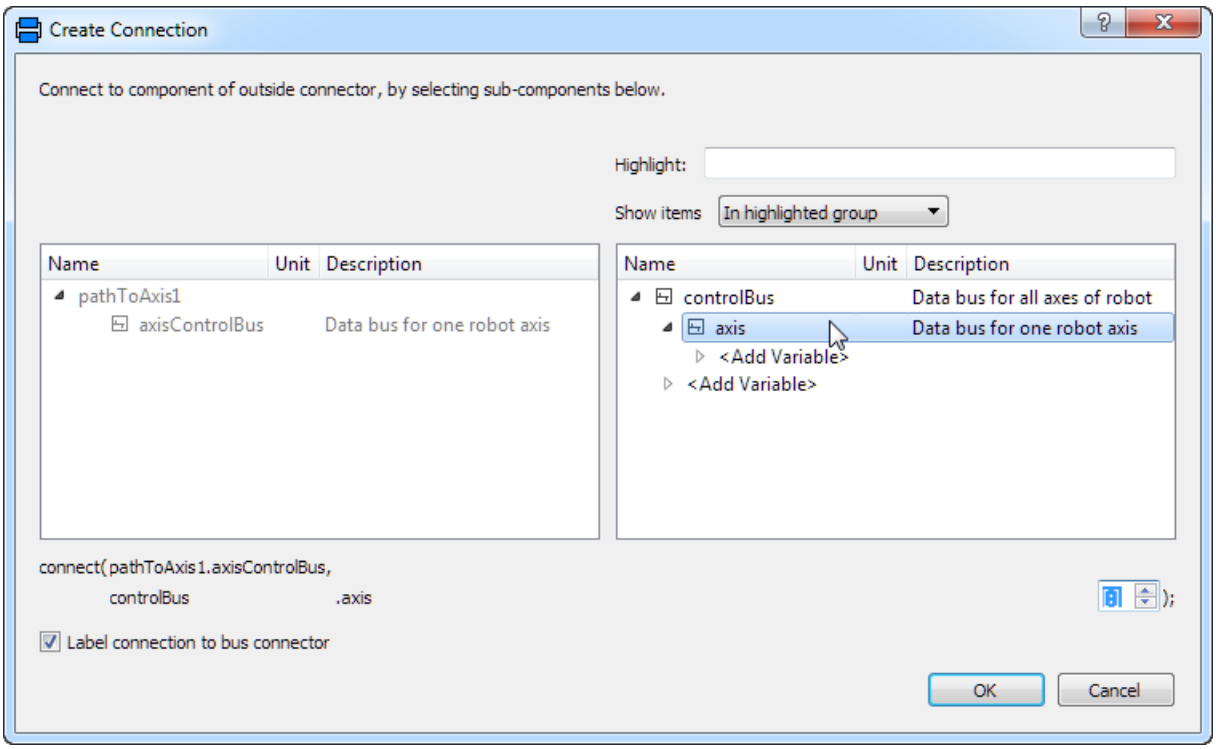


Note that the checkbox for adding a variable to the bus declaration is present also for variables that are visible under <Add Variable> when clicking it; ticking the check box for such a

variable will “push” it above the <Add Variable> entry in the pane and add it to the bus declaration.

By default the variables added to bus declarations this way are stored between sessions, this is controlled by the option **Added variables in bus declarations** that can be reached by the command **Tools > Options**, the **Settings** tab. See “Settings options tab” on page 522.

The handling of expandable connectors also supports connection to a specific array slot in the expandable connector dialog.

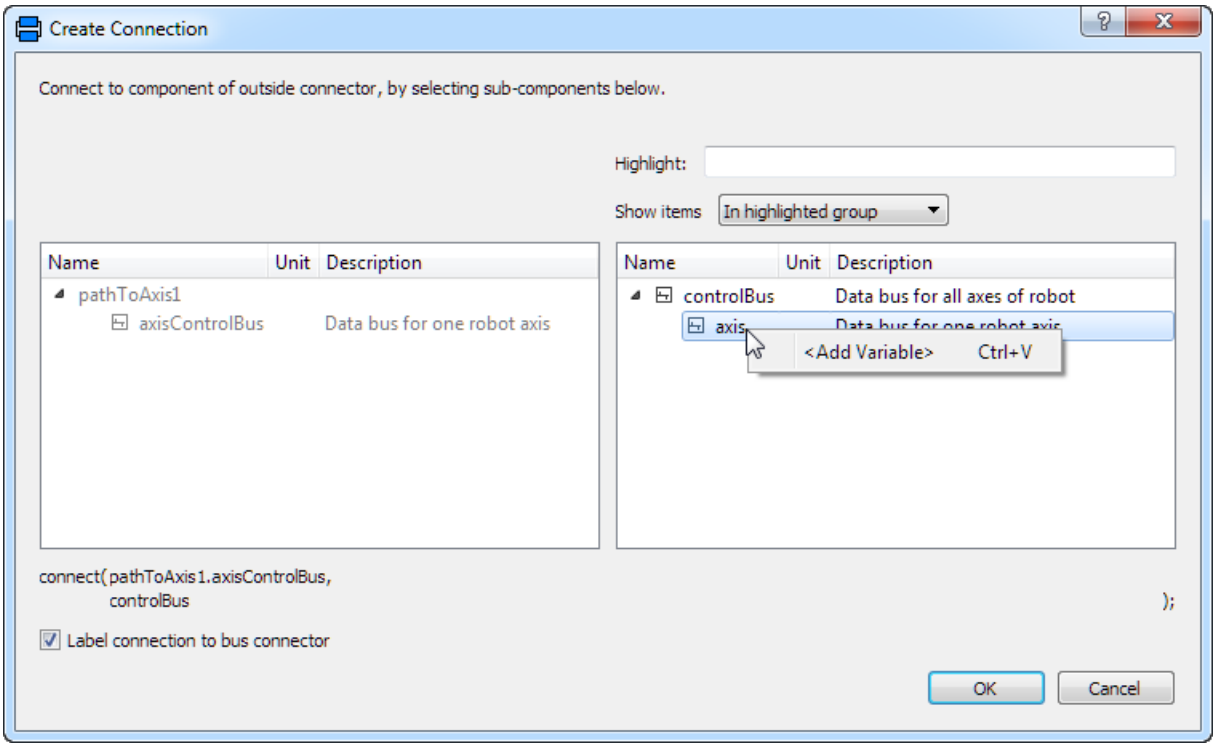


It is possible to use parameter expressions inside the subscripts as well (even though there is a spinbox); and when introducing a new variable e.g. axis[1] can be used as well.

Note that you can trace signals in expandable connectors (bus signals) from the Translation tab in the log window. See next chapter, section “Log window”.

Note also that you have search options in the window, see section “Nested connectors” on page 352 for an example of such search.

There is an option to not display the **<Add Variable>** alternatives in the dialogs, but instead use the context menu to display such an alternative when you want to use it. To activate this option, set the flag `Advanced.AddVariableRequireContext=true;` (the flag is by default `false`). If this option is activated, you must right-click the relevant node in the tree to display the **<Add Variable>** alternative, as an example, to do this for the axis (compare the image above)



Array of connectors

Handling of array of connectors is supported.

- Vectors of connectors can be defined and connected to in the graphical editor.
- The graphical representation is currently as one large connector, not as "n" individual connectors. For that reason, the extent (e.g. height) of the connector should be increased so it can accommodate a reasonable number of connections.
- When connecting to a connector array, Dymola will present a dialog that lets the user select index.
- Connection lines are evenly distributed on the connector, according to index number.
- The orientation of the connector determines if connections start at the top or the bottom. The connections start from the first extent point of the connector. If a different order is desired, select the connector, and on the **Graphics** tab, in the **Draw** group, click the arrow next to the **Arrange** command, and select **Flip Vertical** or **Flip Horizontal**.
- The array dimension can either be a number or a simple parameter:

```
ConnType a[3];
parameter Real n = 2;
ConnType b[n];
```

Array of unary connectors

When creating unary connections the annotation `connectorSizing` can be used. It handles automatic calculation of array indices.

Create an integer parameter with the annotation:

```
parameter Integer nIn=0 annotation(Dialog(connectorSizing=true));
```

Then use the parameter as array index for a connector array:

```
StepIn nPorts[nIn];
```

When creating connections involving the connector array the array index for the connection will automatically be calculated. The first connection will have index 1, the second index 2, and so on.

This feature is very useful for state machines and fluid libraries.

Bus modeling

It is possible to draw connections from a connector back to itself. This allows signals on a bus to be re-routed.

Conditional declarations

Conditional declarations are supported according to the Modelica semantics. If the condition is false, the component is removed when translated (including modifiers and connections to it):

```
model C2
  extends
Modelica.Mechanics.Rotational.Examples.CoupledClutches;
  parameter Boolean addFriction=true;
  Modelica.Mechanics.Rotational.Components.BearingFriction
BearingFriction1(tau_pos=[0,2]) if addFriction annotation
(extent=[62,-62; 82,-42]);
equation
  connect(BearingFriction1.flange_a, J3.flange_a) annotation
(points=[62,-52;
        50,-52; 50,0; 35,0], style(color=0, rgbcolor={0,0,0}));
end C2;
```

(If comments are present, the “if-condition” must be placed before the comment.)

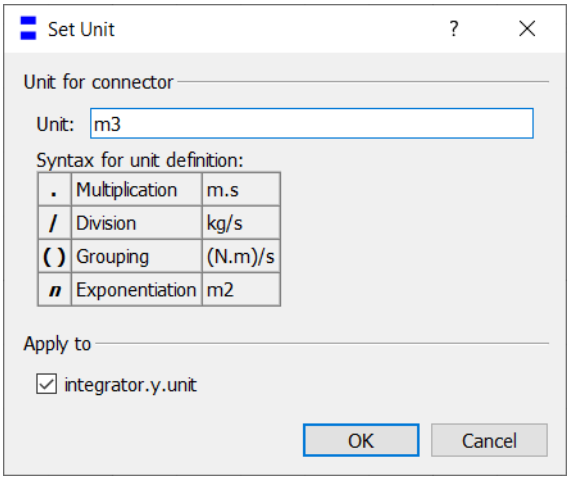
Adding unit to output of block

You can add unit to the output of a block either by the context menu of a connector or by the context menu of a connection line.

Adding unit by using the context menu of the connector

You can add unit to the output of a block by the context menu of a connector. As an example, from the package browser, drag an instance of Modelica.Blocks.Continuous.Integrator to the diagram layer, and right-click the output connector of it and select **Set Unit**. (Be sure to right-

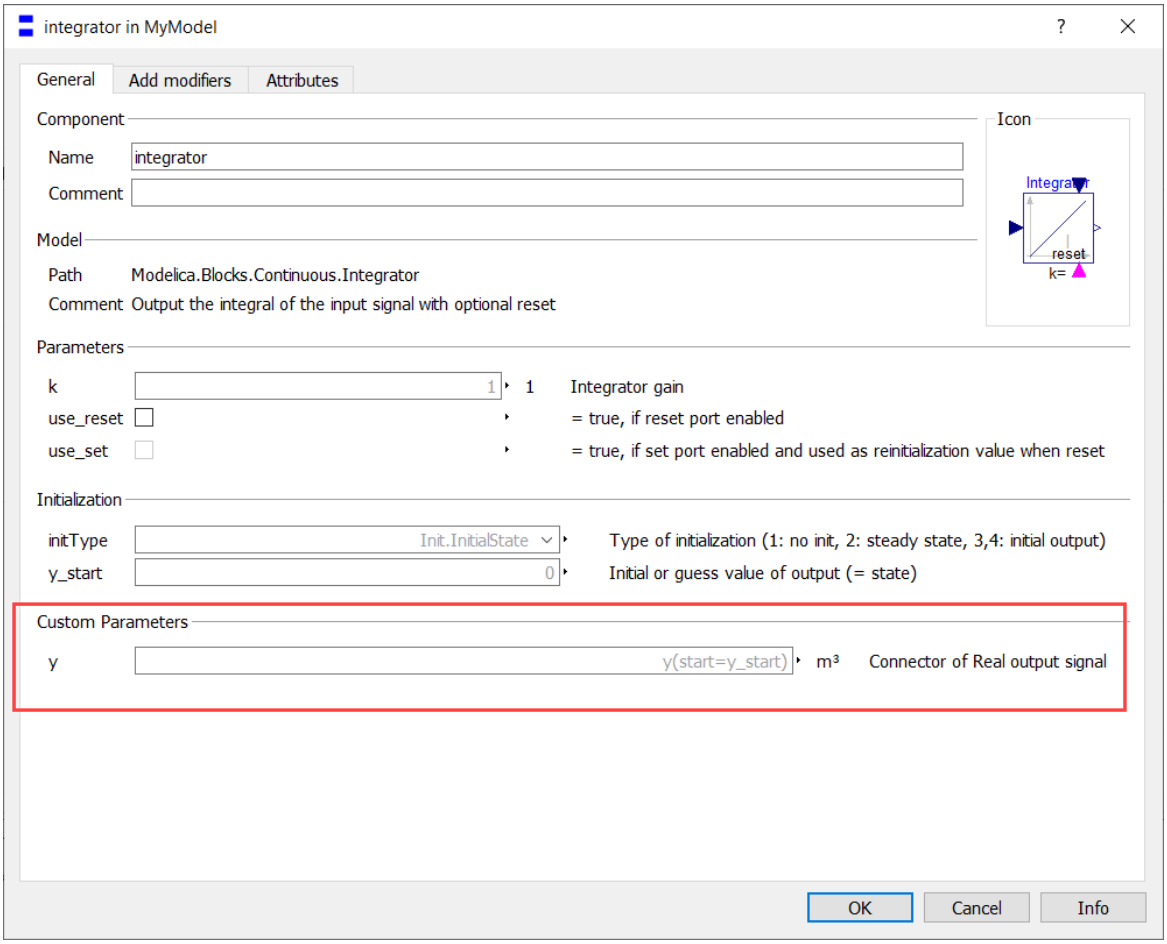
click the connector and not the block – the handles will look the same in both cases.) You will get a dialog like:



The selected checkbox in the menu indicates that the unit will be saved for the connector. (In this case, the checkbox really has no meaning; please see the next example for a better usage of it.)

As an example, enter, as in the figure above, `m3` as unit and click **OK**. The results will be:

- The **Set Unit** entry in the context menu of the connector disappears.
- The unit is visible in the parameter dialog of the integrator as a modifier of a custom parameter:



- The unit is visible as a modifier to the output y of the integrator (looking at the Modelica Text layer of the model):

```
model MyModel
  Modelica.Blocks.Continuous.Integrator integrator(y(unit="m3"))
    ;
  ;
end MyModel;
```

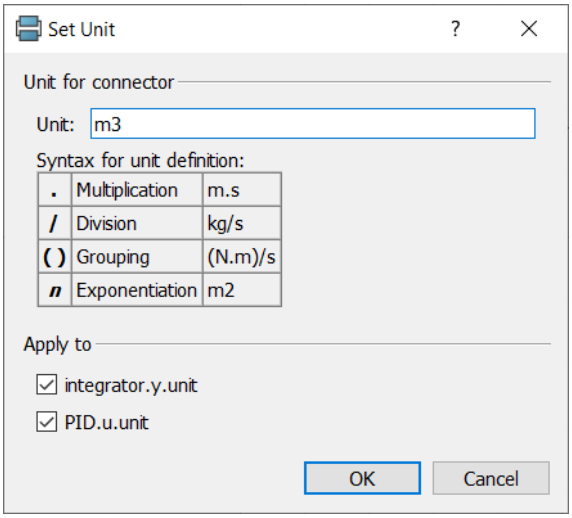
- The unit will be propagated, when possible, when connecting the integrator output to anything (this includes propagating the unit to an input if possible).

Note that a block with a unit like the one above can be saved as favorite, including the unit, by using the context command **Capture Parameters > As Favorite**.

Adding unit by using the context menu of the connection line

You can add unit to the connector of block(s) by the context menu of a connection line. As an example, do the following:

- From the package browser, drag an instance of Modelica.Blocks.Continuous.Integrator and an instance of Modelica.Blocks.Continuous.PID to the diagram layer.
- Connect the two components by a graphical connection.
- Right-click the connection line and select **Set Unit**. The result is (compare with previous example):



In this case, if you enter a unit and keep the checkboxes selected, the unit will be saved to both the y output of the integrator block and the u input of the PID block. If you, for example, clear the checkbox of the PID.u.unit, the unit will be propagated from the integrator block when translating, but it will not be saved to the PID.u.unit connector.

Presentation of components and connectors in the diagram layer

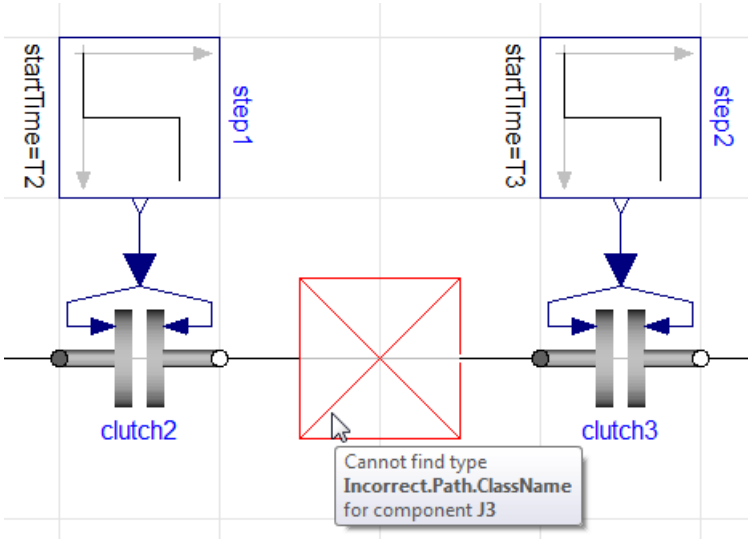
Conditional components/connectors

In a top-level view, conditional connectors and components (see previous section) are always shown regardless if they are enabled or disabled, since all cases must be taken into account by the model creator. When using the context command **Show Component**, disabled conditional components and connectors are rendered with a dotted rectangle. When the icon of a model is shown in the diagram layer, disabled connectors of that model are not rendered. These features help avoiding creation of dangling connections to the disabled components and provide information about what connectors and components are enabled.

Conditions for conditional components/connectors are shown in the tooltip information of the component/connector.

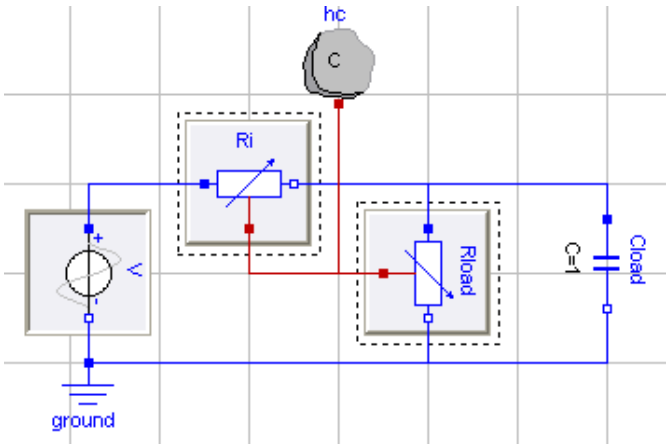
Components of a non-existing class

Components of a non-existing class are marked with a big red cross in the graphical editor. This makes it easier to find model inconsistencies. The tooltip will display the type when there is a type error:



Replaceable, redeclared and inherited components

The highlighting indication makes it easier to visualize model structure in the diagram layer of the edit window.



Replaceable and redeclared components:

- Highlights components and connectors that can be replaced. Replaceable components with the default value are shown sunken, as a socket. Redeclared components are shown raised, as mounted in the socket. Components of a replaceable/redeclared *class* are

outlined with a dashed line. Note that redeclared components are only highlighted if they are *modifiable*, either because they are replaceable, or because the redeclare is in the current model (whether the current model is editable is ignored).

Inherited components: (off by default)

- Highlights components and connections inherited from a base class with gray background color. Such components cannot be deleted, moved or replaced.

Highlighting can be turned on or off by the option **Highlight replaceable and redeclared components** in **Tools > Options**, the **Graphical Editor** tab, for example when copying to clipboard or exporting images. It is always off when using the command for exporting to HTML; **Tools > HTML**. Redeclared components are always shown, if they fulfil the conditons in the first item above.

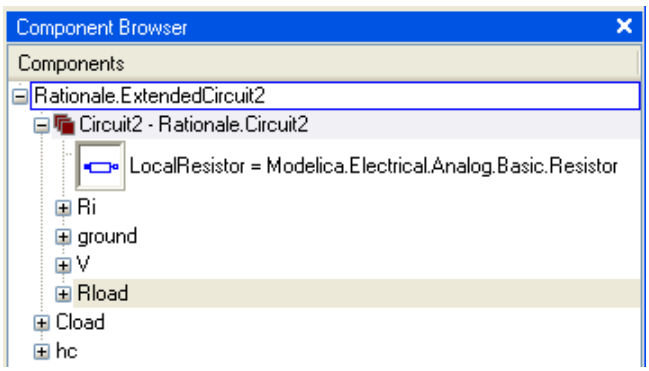
You can also use flags to control indication. The indication of replaceable *classes* (which also includes classes in replaceable packages) in the diagram can be controlled by the flag `Advanced.Editor.Highlight.ReplaceableClass`. The default value of the flag is `false`, meaning that replaceable classes in the diagram are not highlighted. The indication of replaceable or redeclared *components* in the diagram can be controlled by the flag `Advanced.Editor.Highlight.Replaceable`. The default value is `true`, menaing that the replaceable or redeclared components in the diagram are highlighted. This flag corresponds to the opton option **Highlight replaceable and redeclared components** above.

Presentation of components and connectors in the package browser

- Changed but not saved classes are shown with a red background in the package browser.
- Replaceable classes are shown with a gray background in the package browser.

Presentation of components and connectors in the component browser

- Replaceable classes are shown in the component browser. Redeclared classes are highlighted as in the diagram layer (raised or sunken).



Presentation of diagram layer instead of icon

The diagram layer can be shown instead of the icon layer of an instantiated class by setting the annotation `__Dymola_openIconLayer=true` for that class.

4.3.7 Media propagation using components from Modelica.Fluid

A flag facilitates media propagation when using components from the Modelica.Fluid library. The flag is

```
Advanced.MediaPropagation
```

The flag can have the following values:

- 0 no media propagation by this feature (default value)
- 1 media propagation with dialogs
- 2….media propagation, but with fewer/no dialogs

Important! The flag is not saved between sessions.

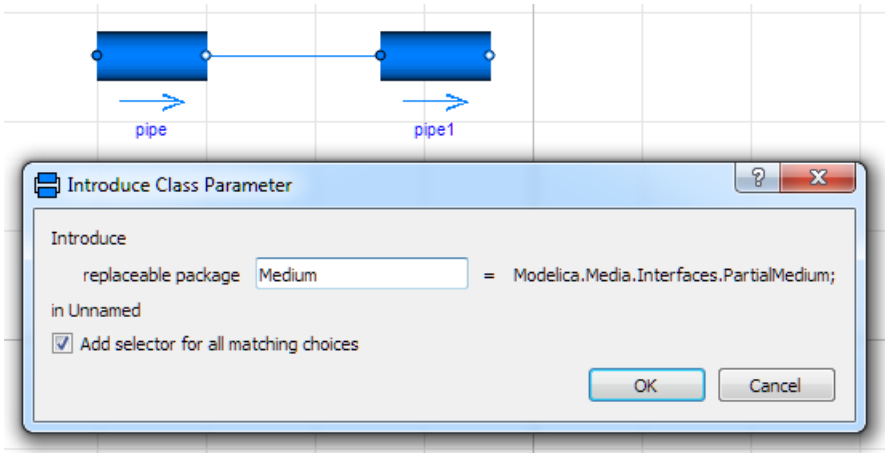
The feature works both with smart connect and normal connect. Note that the feature handles propagation with different names, facilitating building heat exchangers etc.

Note also the possibility to automatically have the propagations with attribute final by setting the flag

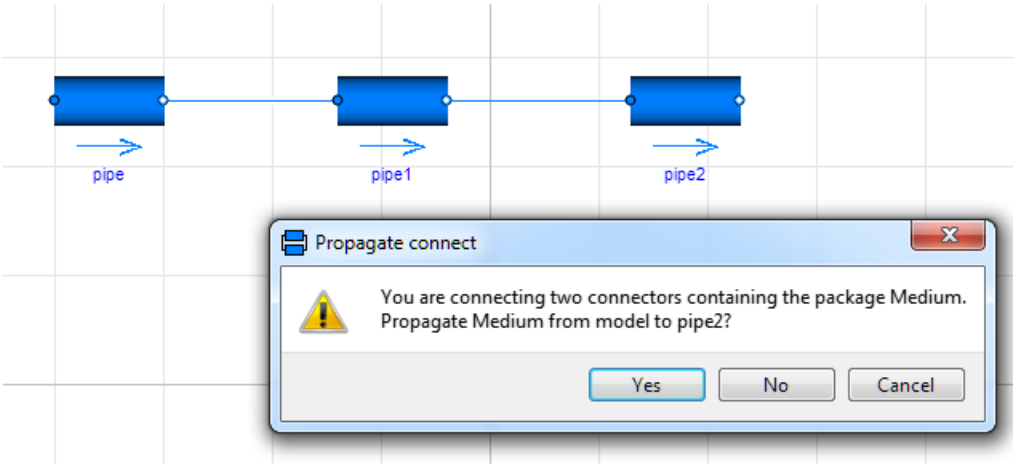
```
Advanced.Editor.FinalPropagate=true
```

(The default value of the flag is `false`.) For more information about this flag, and propagation in general, see the description of the parameter **propagate *<parameterName>*** in the section "Context menu: Parameter dialog; parameter input field" starting at page 572.

As an example, consider setting `Advanced.MediaPropagation=1` and then instantiating and connecting two StaticPipe components from Modelica.Fluid.Pipes. The following dialog appears:

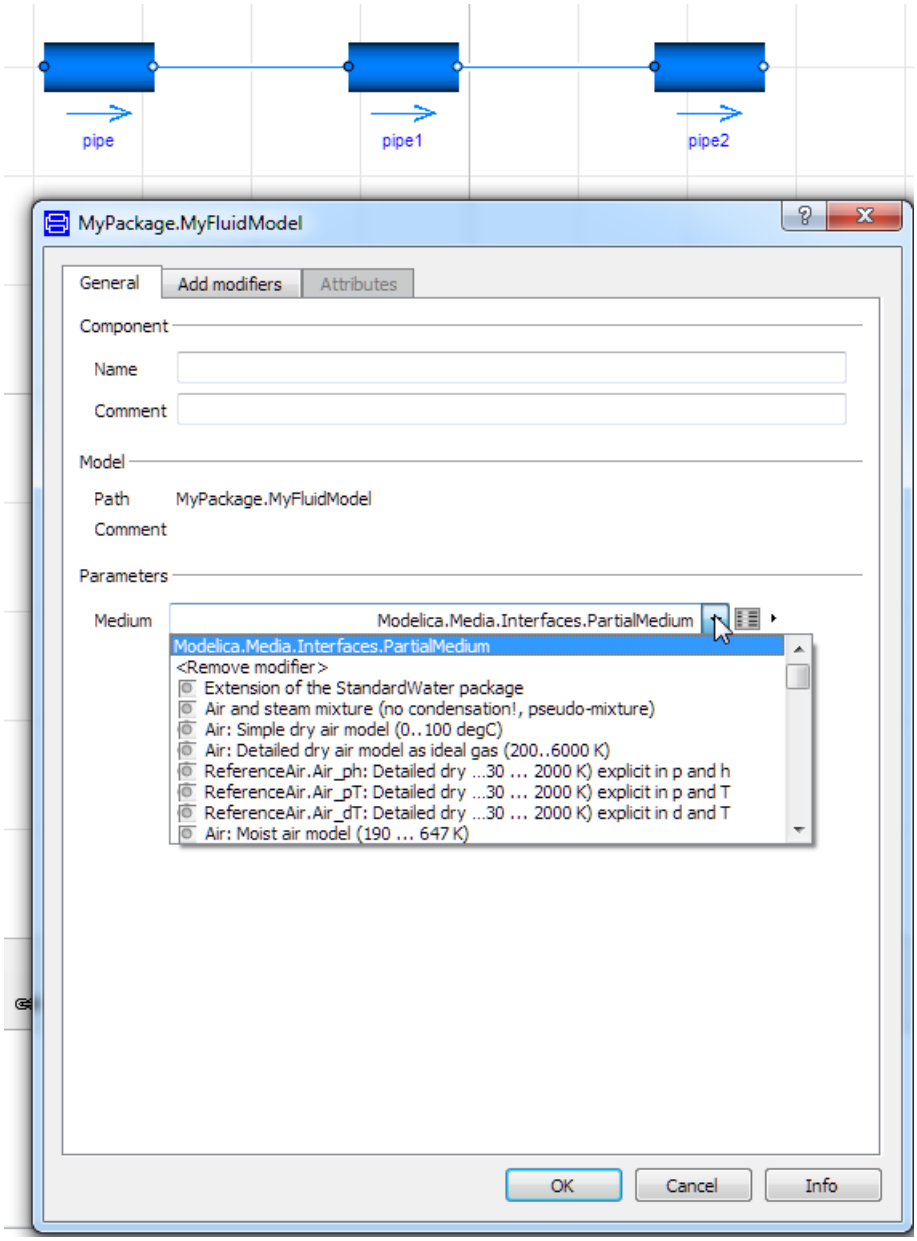


Clicking OK and connecting yet another StaticPipe gives the dialog:



Clicking Yes propagates Medium to the new StaticPipe.

If you want to change the default for the Medium you have to right-click an empty area in the model and select Parameters. Top-level parameters, including the Medium, are present in this dialog:



You can use the arrow down for the Medium, like in the figure above, to see possible selections.

4.3.8 Building templates using replaceable components

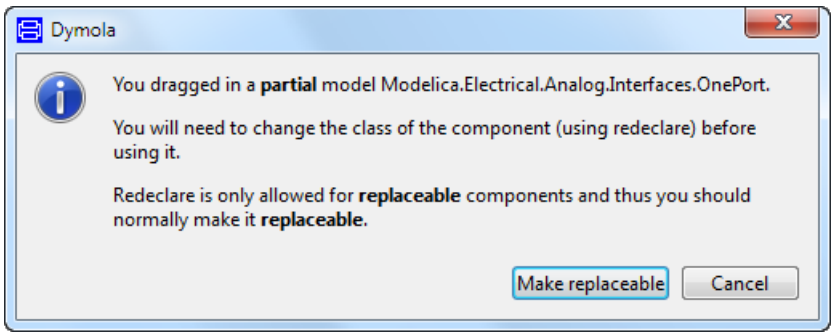
Building templates

It is easy to build model templates using replaceable components:

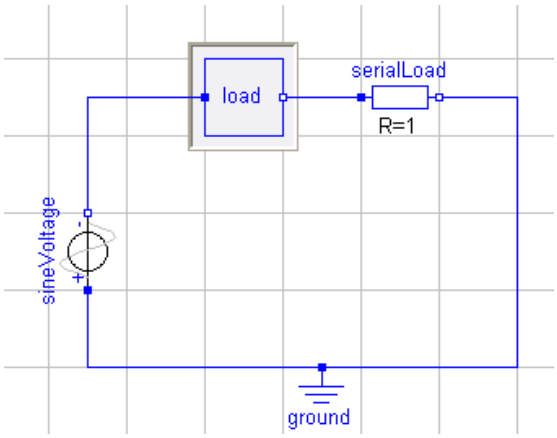
Note. Compare building test-driven template models by using part of a complete system, see next section.

- Drag and drop of partial models construct replaceable components.
- For a replaceable component, it is possible to modify the default class with the context menu entry **Change Class**.
- The **Attributes** tab in the parameter dialog for the component makes it possible to set and modify the constraining class (and replaceable attribute), both for the original declaration and a redeclare.
- Local replaceable classes (class parameters) can be used; e.g. to implement a Wheel class in a car model. For more information about class parameters, see the chapter "Introduction to Modelica", section "Advanced modeling features", sub-section "Class parameters".

As a simple example consider building an electrical test-template for Resistor, Capacitor, and Inductor. This can be done by dragging in an OnePort from Modelica.Electrical.Analog.Interfaces. You will get a message

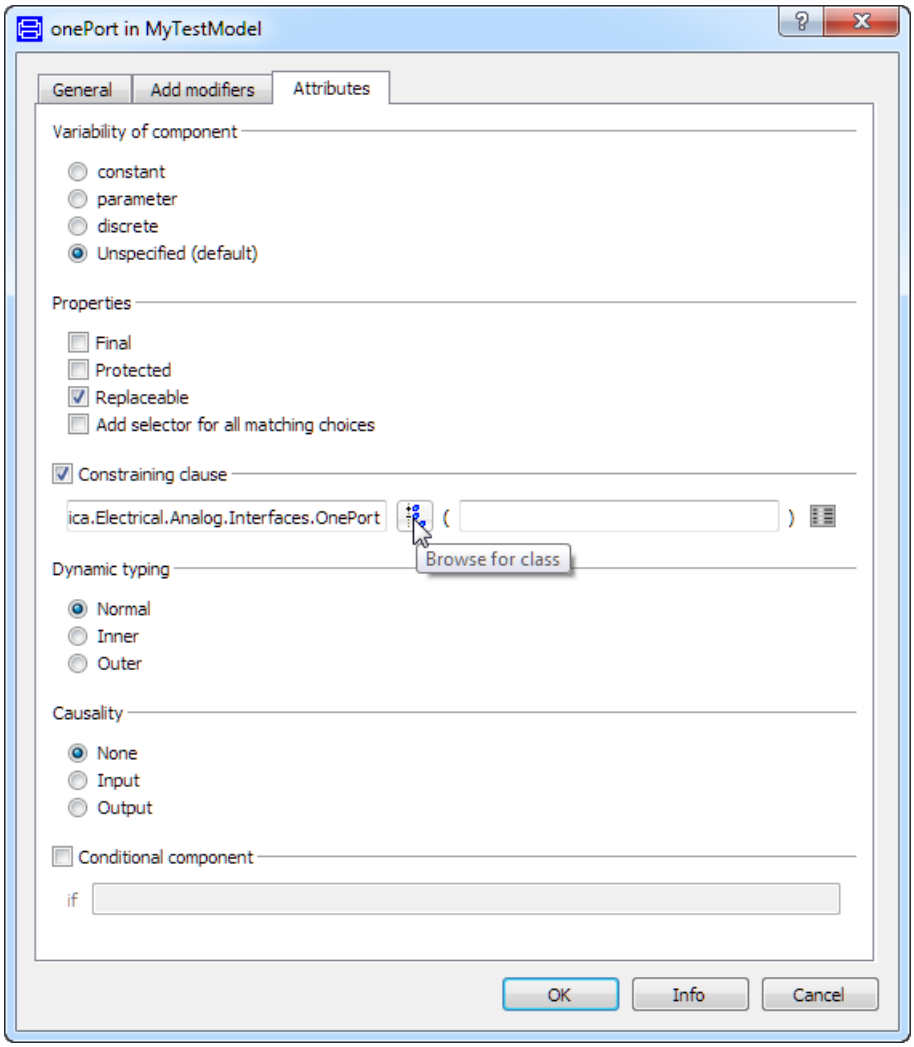


By building the circuit in the usual way one then gets



and can then change the default to Resistor using **Change class > All matching choices** in the context-menu of load.

Alternatively, one can modify replaceable and constraining class using the **Attributes** tab in the parameter dialog. (The parameter dialog is reached by right-clicking the component and selecting **Parameters**, or by default by double-clicking the component.)



For more information about the context menu, please see section “Context menu: Components” on page 543.

Use the tool  to browse for the constraining class. This icon is used in general to activate a class-browser.

Both alternatives can give the same model as result. The difference is whether you find it more convenient to build the template directly, or want to start by building an actual model, test it, and then turn it into a template with replaceable components.

Using templates

The menus are adapted for easy use of templates:

- The list in the context-menu **Change Class** has a separator between top-level packages, and a status-message indicating the full path of each class. (The full path is also available as a text by pressing **Shift+F1** on the entry.)
- The list in the context-menu **Change Class** is hierarchical (in one level) to combine models that are parameter sets (i.e. comprised solely of an extends-clause with modifiers for parameter changes).

The usual use of a template is to extend from it, or use it as a sub-component.

4.3.9 Building test-driven template models

The structured way of building template models (by finding a suitable base model, e.g. a base interface model, adding that base class, making it replaceable, making a subsystem, redeclare and test the complete system) is well supported, see the previous section.

However, it is also possible to build a test-driven template model, by:

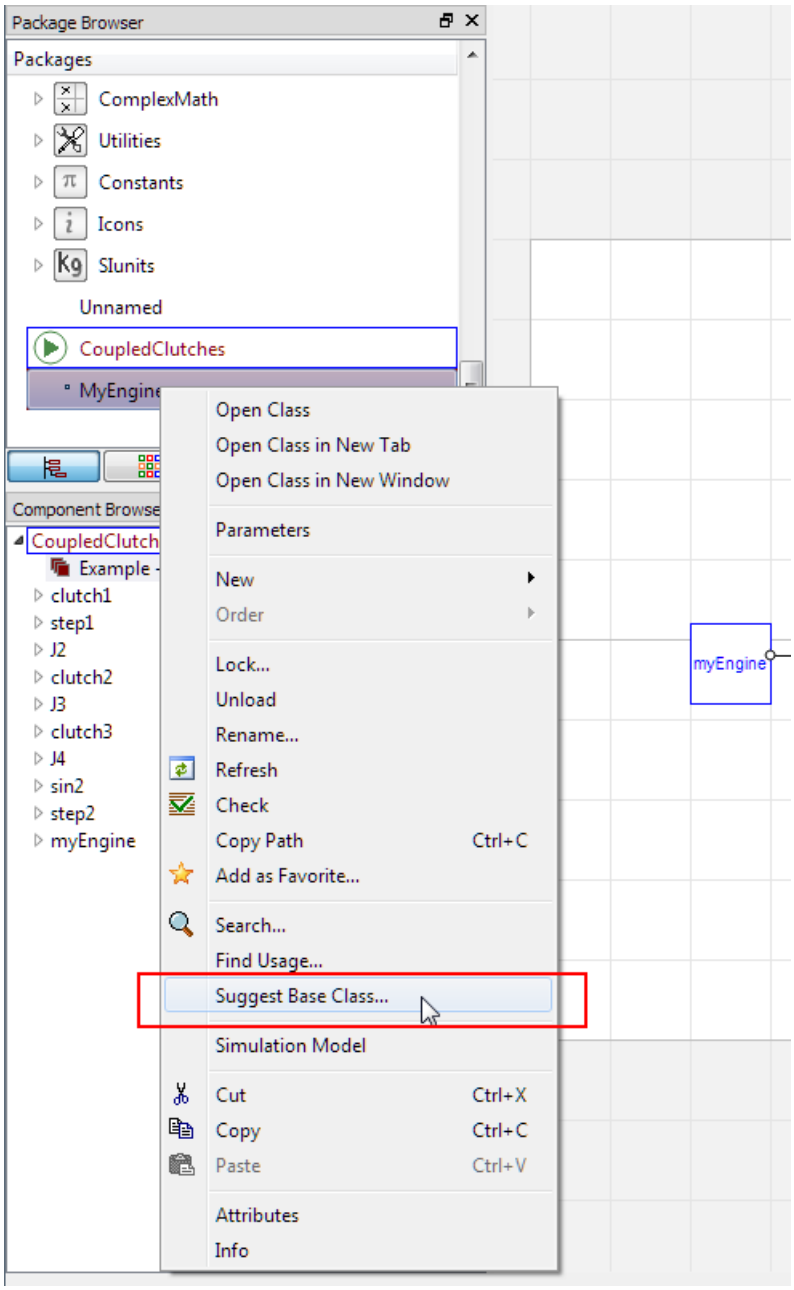
1. Testing a complete system.
2. Making a subsystem (using the context menu command **Split Model**).
3. Making sure
   a. Finding suggestions for base classes.
   b. Adding the selected base class.
4. Making the subsystem component replaceable, including defining the constraining clause of the template model component to be the selected base class.

(Note that minor cleanup is needed, e.g. adding icons, moving connectors.)

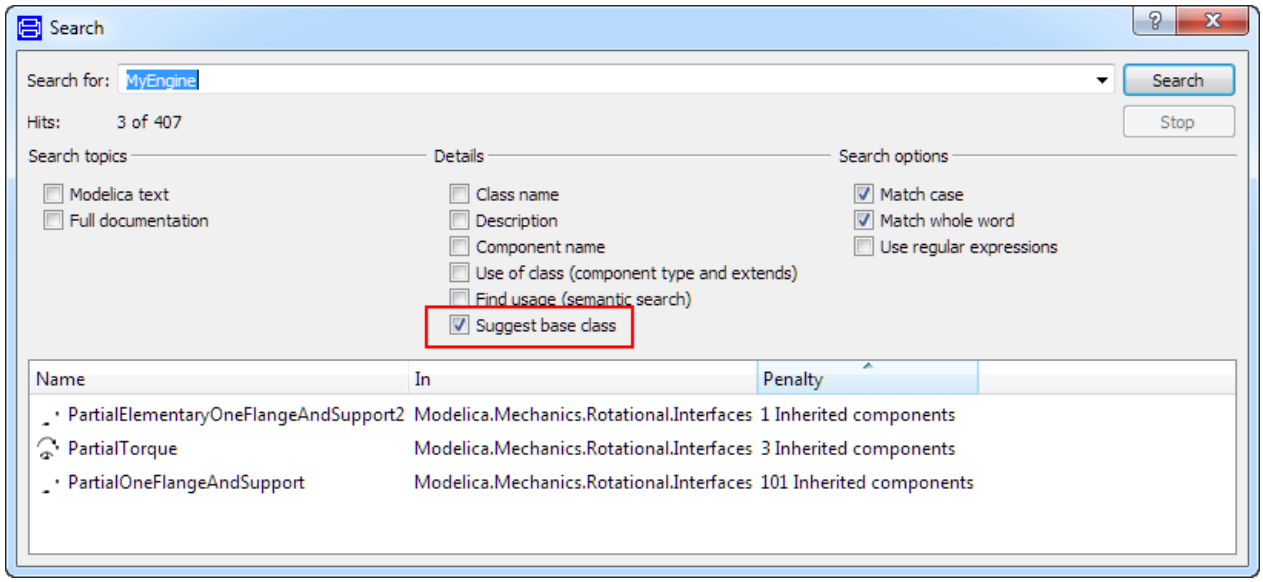
The item 3a, 3b, and 4 in the list above are described in the following sections.

Search for base class suggestions when creating a template model

As an example, assume we want to create a template from the three first components in the Couple clutches demo. To make this demo model editable, first duplicate it by, in the package browser, using the context command **New > Duplicate Class…**. Then select the three first components. A submodel can now be created by right-clicking and using the **Split Model** command, naming the submodel `MyEngine`, then right-clicking that submodel in the package browser, and selecting the command **Suggest Base Class…**,



This command opens the following dialog, with the option **Suggest base class** selected. When you have clicked **Search** the result is:

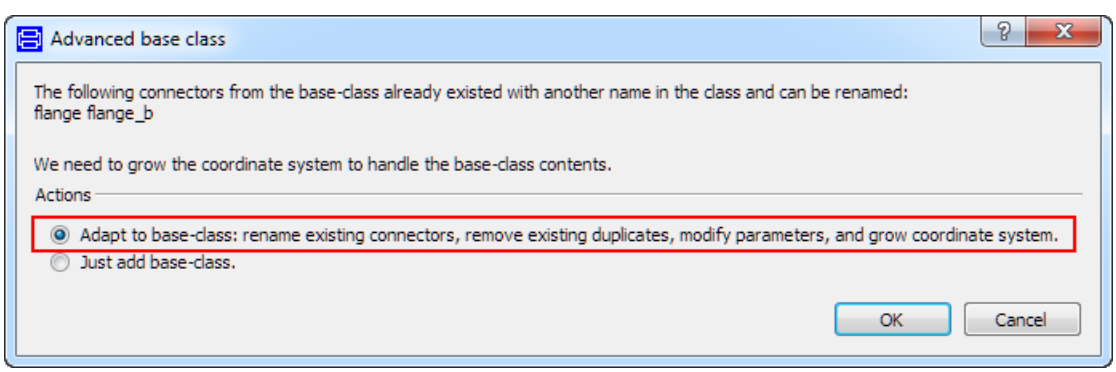


(It may be noted that the last found suggestion in the figure above creates a heavy penalty since the fixed is inherited and also exists locally.)

This section corresponds to 3a in the list above.

Adding a selected base class when creating a template model

The results in the dialog above can be double-clicked to get possibility adapt the template to the selected base class. Double-clicking any of the alternatives in the list above (we select the first one) gives:

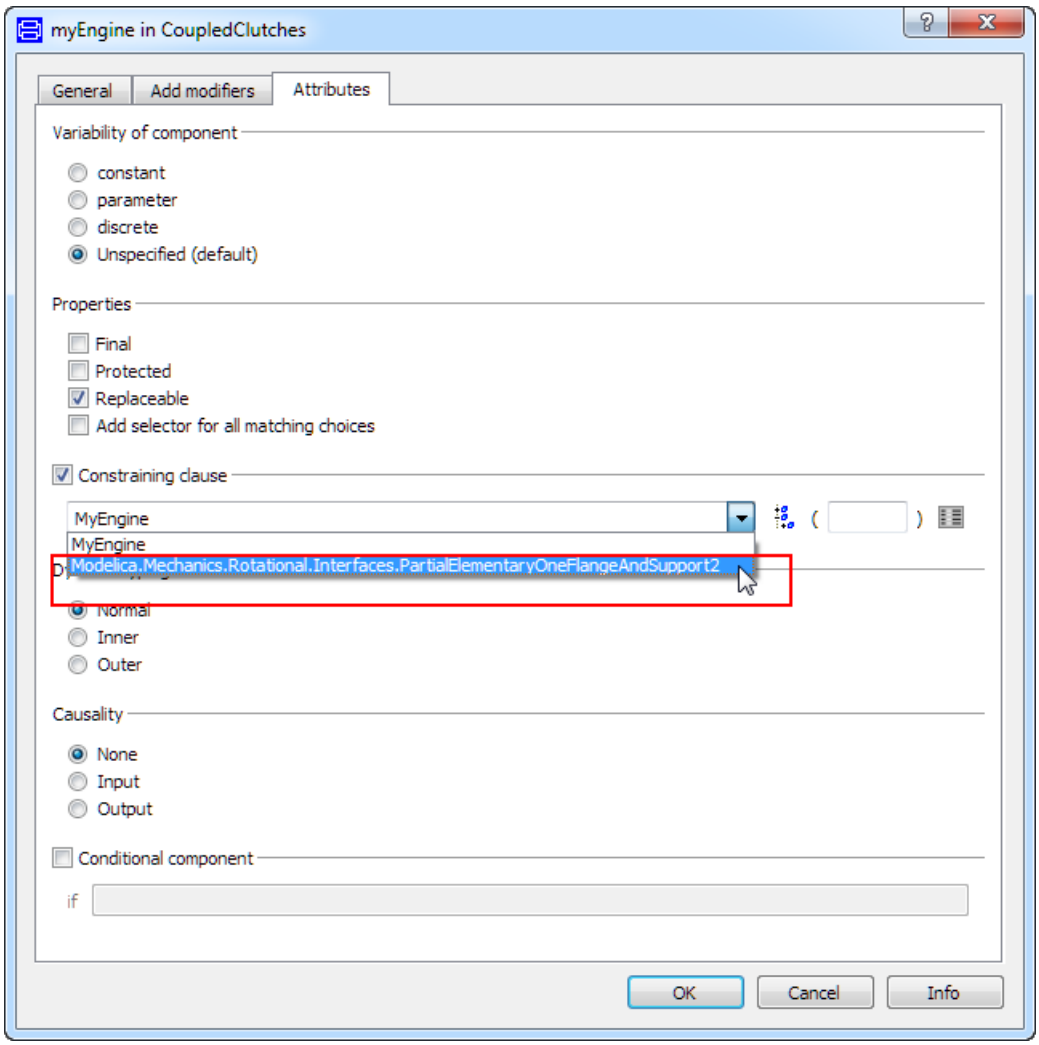


The default selection here is to perform the actions needed to adapt the model that should be the template to this base class. The selection **Just add base-class** gives no additional actions, it corresponds to dragging a class from the package browser to the component browser.

This section corresponds to 3b in the list above.

Defining the constraining clause of the template model components to be the selected base class

To make the template model component replaceable and define the constraining clause as the base class selected for the template model, right-click the template model component in the model where it was first defined (in our case a copy of the Coupled clutches demo) and select **Parameters**. In the **Attributes** tab, select **Replaceable**, and select the constraining clause to be the used class:



This section corresponds to item 4 in the list above.

4.3.10 Cleaning up a model developed step-by-step

It is not uncommon that a model has been developed step-by-step, working with different parameterizations and cumulative changes and additions. Such a model may have a number of inheritance and modifier levels. However, often you want a model that is similar to the model you started with in the first place, when it comes to inheritance levels and modifiers.

To cleanup such a model, right-click a suitable base class in the component browser and then select **Capture Parameters > As Model**. This will create a model that has a minimum of inheritance levels and modifier complexity, similar to the model you started with in the first place.

4.3.11 Parameters, variables and constants

General

Four *variability types* (or basic *type prefix*) can be selected in a drop-down menu. The choices **Constant**, **Parameter** and **Discrete** are specified in the menu; by selecting an empty field (default in the menu) the variable type **Variable** is defined.

Parameter values

For general information about parameter values in Modelica, please see the chapter "Introduction to Modelica", section "Initialization of models", sub-section "Parameter values".

Note the support for string parameters; see section "String parameters" on page 393.

Dymola issues an error for a parameter with fixed = true (this is the default) having neither value nor start value. This may be relaxed by setting the flag

```
Advanced.IssueErrorForUnassignedParameter = false
```

which turns the error into a warning and forces a zero (`false, ""`) value to be used for the parameter.

Dymola issues a warning for a parameter with fixed = true having no value but a start value. The idea is to hint a user that the parameter value is generic and that the user ought to set it.

The warning may be suppressed by setting the flag

```
WarnAboutParametersWithNoDefault = false
```

Concerning parameter evaluation, please see next chapter, section "Model simulation", sub-section "Simulation settings".

Declaration of parameters, variables and constants

There are two principal ways to work with declaration/editing of parameters, variables and constants.

- Using the variable declaration dialog, accessible by commands. In the diagram layer of the edit window, available on the **Graphics** tab, using commands.

- Directly editing Modelica text in the Modelica Text layer of the edit window.

There are some differences when editing variables in the diagram layer of the edit window, and doing it in the Modelica Text layer for the window. Of course the ways can be mixed, typically defining a variable using the variable dialog, but changing it by typing in the Modelica Text layer.

For deleting of parameters, variables and constants, see section "Deleting parameters, variables, and constants" starting on page 383.

Working in the diagram layer

Declarations/editing of parameters, variables and constants are made using two different menus, the variable declaration dialog and the parameter dialog. What menu that is used depends on the user action. Below are listed some examples how to reach these menus. The menus will later be presented in sections of their own.

Parameter dialog: Double-clicking on a component

Double-clicking on a component in the graphical editor (diagram layer of an edit window) by default displays the parameter dialog for the component. (This behavior might be changed by the **Tools > Options** command. If double-clicking does not open the parameter dialog, **Shift+double-click** will do it. See "Graphical Editor options tab" starting on page 516 for details.)

Parameter dialog: Using a context menu command

Displaying the context menu of components and extends-clauses in the component browser or in the edit window and selecting **Parameters** will display a parameter dialog. This makes it possible to directly change inherited parameters.

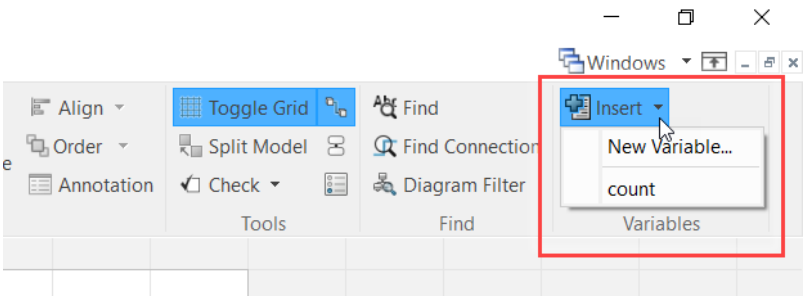
Selecting **Propagate…** or, in some cases, **Propagate <*parameterName*>** in the context menu of the parameter input field in the General tab in the Parameter dialog will display a variable declaration dialog.

In the component browser you can also use the context menu on parameters, variables, and constants of the top level in a model to display the variable declaration dialog, if you first select to present such top-level items in the component browser. To do this, use the command **Tools > Options**, and tick **Include non-graphical** in the **General** tab. (See "General options tab" on page 511.) Once they are visible, you can right-click any of them and select **Variable…** to display the variable declaration dialog.

Variable declaration dialog: Using a command



It is easy to access all variable declarations in a model by, on the **Graphics** tab, in the **Variables**. group, clicking the arrow next to **Insert**. This allows easy modification of attributes and common annotations. This command can be used to either modify an existing variable or to define a new one. Please also see section "Graphics > Insert [Variables]" starting on page 484 for more information.

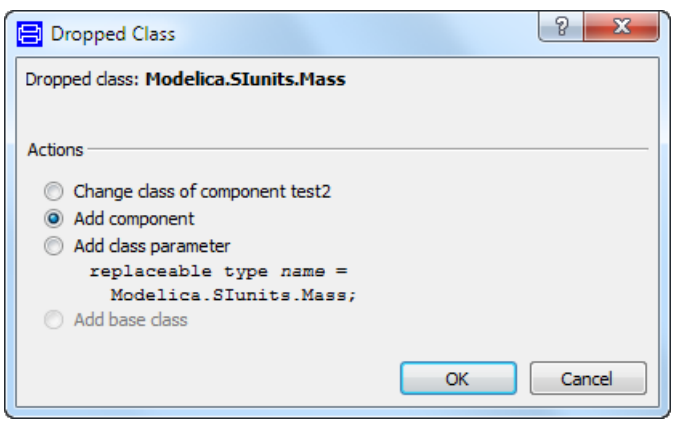


Drag-and-drop operation

Most variable declarations can be generated by a drag-and-drop operation from the package browser to the component browser.

To declare a new variable, drag a type from the package browser to the component browser or diagram. **Note!** To get the below menu and avoid the (in this case) unwanted default behavior of **Change class of component**, drag the component to between two components, not on top of any component.

Several operations are possible, but the default menu alternative is to add a new variable of the specified type:



Dymola will then show a variable declaration dialog.

Writing Modelica code

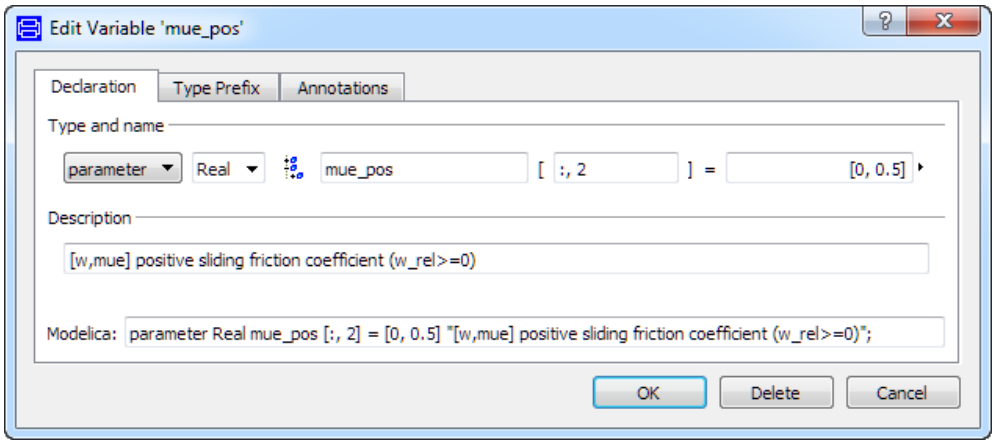
Parameters and variables can of course also be declared in the Modelica Text layer. Please see section “Programming in Modelica” on page 271.

Variable declaration dialog

The variable declaration dialog is presented to declare new variables or to edit existing ones. This means that the declaration dialog might have different headers: e.g. “New Variable” or “Edit Variable xxx”.

Declaration tab

Main declaration of component.



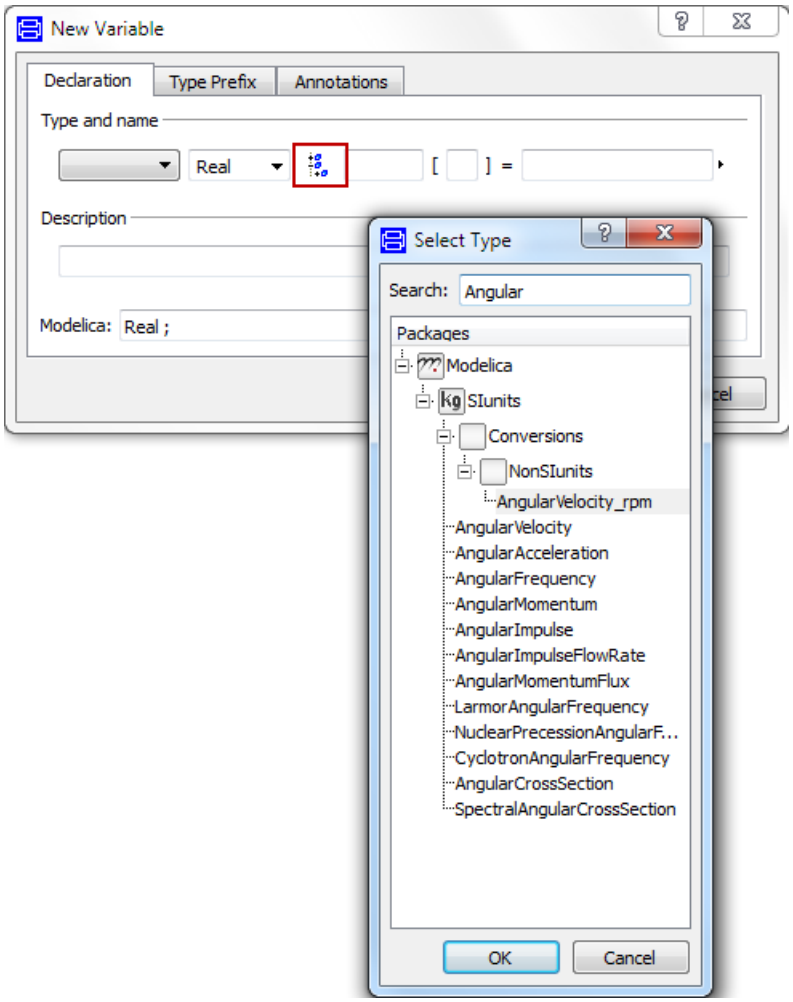
Depending on in which situation the variable declaration dialog is used, the available alternatives to select might differ slightly. The example above is from the parameter mue_pos in a clutch.

Type and name group

The type and name group consists of five input fields.

Basic **Type prefix** (or *variability type*) is selected in the first field. Keeping the field empty (default) defines a continuous Variable. If Constant, Parameter or Discrete is wanted instead, any of them can be selected using the pull-down list. For an explanation of the types, please see above.

Type name The type of the variable. The pull-down list contains the type, its base types and other types in the same package. If you want to browse for a type not in the list, you can click the icon  next to the field to get a browser. In that browser you can search for a type using **Search**. An example from defining a new variable:



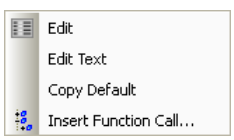
.

What is entered in the **Search** field will dynamically select what is shown in the Packages pane. The choice selected by default is marked with a light grey background. If the default selection is the wanted one, it is sufficient to click **OK** to get that type. Otherwise the wanted type can be selected by clicking on it and then clicking on **OK**.

Variable name – the name of the variable being declared.

Array dimensions for a non-scalar variable. For example, 2,4 declares a matrix with 2 rows and 4 columns.

Value – the default value of the variable, used for parameters and constants.



A context menu is available for the Value field. The menu can be used to e.g. edit the value. If the values should be given in a matrix form data for the table can be imported from external files in CSV format (Excel), as text files or as Matlab files. The context menu is reached by right-clicking in the field or by clicking on the arrow after the field. For more information

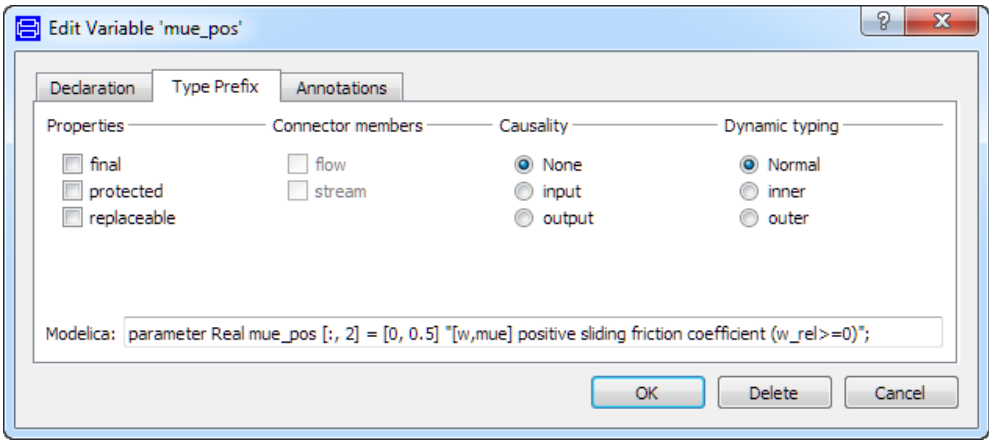
about this menu, please see the section “Context menu: Variable declaration dialog; Value input field” on page 559.

Description group

Here a description of the variable can be added/edited.

Type prefix tab

Attributes.



Additional type prefix is specified using this tab.

Properties group

Specifies how the variable can be accessed or changed in modifiers.

final – the value of the variable cannot be modified.

protected – the variable cannot be accessed from outside the model. This is the recommended way usually to protect a variable, rather than declare it as hidden (see next tab).

replaceable – the variable can be redeclared in a modifier list.

Connector members

The **flow** attribute specifies that this quantity should sum to zero when connected. Examples of such quantities are torque and current. The **stream** attribute is used for handling flows. See the Modelica Language Specification for more information.

These attribute only apply to variables in connectors.

Causality group

Specifies the causality of the variable, which is needed for public variables in blocks and functions.

None – the variable has unspecified causality (determined during translation).

input – the variable is an input to the model.

output – the variable is calculated by the model.

Dynamic typing group

This group specifies the type properties of the variable, and applies to more advanced users.

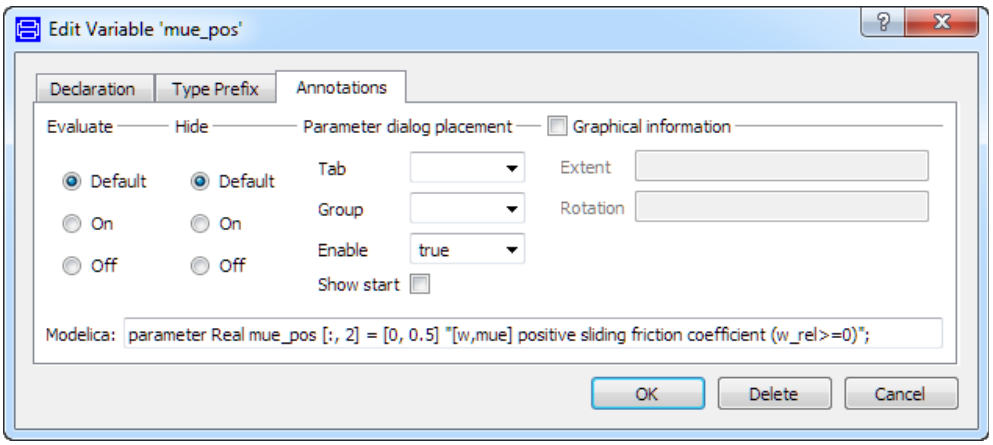
Normal – the variable has no dynamic type.

inner – the variable can be referenced by inner variables in nested components.

outer – the variable shall match an outer variable in an enclosing scope.

Annotations tab

Annotations.



In the annotations you can specify whether a parameter should be evaluated during translation, whether a variable should be hidden and in what tab and group of parameters the variable input field should be present in, and also make the input field conditionally enabled depending on other parameters. Display of a variable as a start value can be activated. The attributes of a graphical component can also be specified. These settings are stored as annotations.

Evaluate group

Specifies whether a parameter is evaluated during translation or not. Please note that a global setting for evaluation is also available, please see next chapter, section “Model simulation”, sub-section “Simulation settings” for more information. In this section also different log possibilities for evaluated parameters are presented.

Default – the evaluation is controlled by the settings in **Simulation > Setup…**.

On – the parameter is always evaluated during translation. (The annotation will be `annotation(Evaluate=true)`.) This setting will override the global setting.

Off – the parameter is never evaluated during translation. (The annotation will be `annotation(Evaluate=false)`.) This setting will override the global setting.

Notes:

- If you use the annotation `annotation(Evaluate=true)` on a record or model component all corresponding parameters will be evaluated even if some of them individually have the annotation `annotation(Evaluate=false)`. This means that you

do not have to add the annotation `annotation(Evaluate=true)` to each individual parameter of a record or model component to evaluate it. Note that this does not apply if you use the annotation `annotation(Evaluate=false)` on a record or model component; this annotation will not override any annotation on any corresponding individual parameter.

- If a parameter having annotation `annotation(Evaluate=true)` is bound to a parameter expression, the parameter expression will also be evaluated, unless it involves parameters with annotation `annotation(Evaluate=false)`.

Hide group

Specifies if a variable is shown in the variable browser. Please note that if a variable is internal, it is better to declare it protected using the **Type Prefix** tab (please see above).

Default – show variable if not declared as protected.

On – never show variable in variable browser. (The annotation will be `annotation(HideResult=true)`.) Please note that such a variable will not be stored during simulation.

Off – always show variable in variable browser. (The annotation will be `annotation(HideResult=false)`.)

Parameter dialog placement group

This group specifies the layout of the parameter dialog window. The parameter dialog can be structured at two levels. The first level consists of Tabs which are displayed on different pages of the dialog. The second level is a framed group on the page. Different variables with the same tab and group attributes are placed together. If empty, the default tab and group are used.

The **Enable** attribute specifies conditional parameters. If the expression is true, the corresponding input field in the parameter dialog is enabled.

The **Show start** attribute activates the dialog entry for a variable to be displayed as a start value.

The user can edit the look of the dialog in other ways as well – see following sections. See also the chapter "User-defined GUI".

Graphical information group

Position, size (extent) and rotation of graphical component.

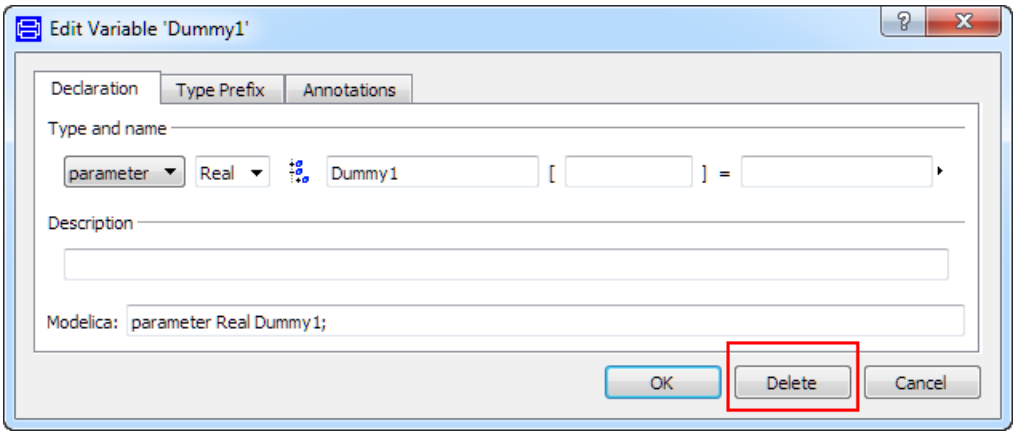
Deleting parameters, variables, and constants

You can delete a parameter, variable or constant by the corresponding variable declaration dialog, reached by any of:

- On the **Graphics** tab, in the **Variables** group, clicking the arrow next to **Insert**  and selecting variable.
- In the Modelica Text layer using the context command **Variables > variableName**. (You reach the Modelica Text layer by, on the **Text** tab, in the **Layer** group, clicking the arrow next to **Modelica** and selecting **Modelica Text**.)

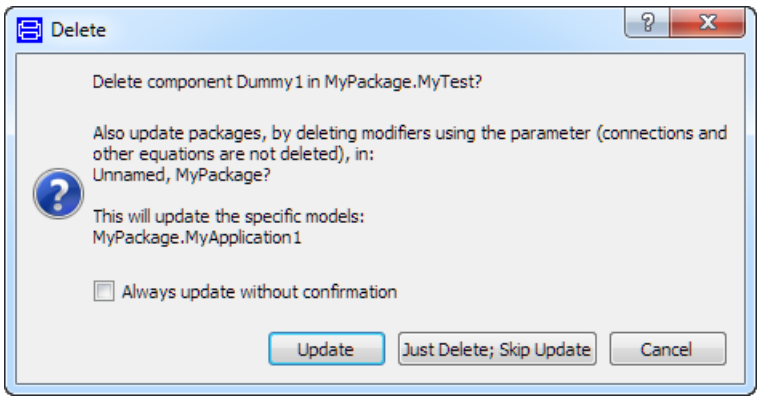
You can also delete by removing the corresponding text in the Modelica Text layer, but then the below is not performed.

In the variable declaration dialog, there is a **Delete** button available:



If the variable has modifiers, for example a start value, a warning message of hanging modifiers appears.

As an example, a simple model `MyTest` contains a parameter `Dummy1`. A component of `MyTest` is present in `MyApplication1`, where the parameter `Dummy1` is given the value 1 in the parameter dialog of the `MyTest` component. When deleting `Dummy1` in `MyTest` by clicking the **Delete** button in the dialog above, the following message appears:



The second part states all packages that have been investigated due to the possible delete: this is all open packages in the package browser.

The conclusion is that `MyApplication1` can be updated, due to deleting component `Dummy1` in `MyTest`.

The menu gives some choices:

- If you click **Update**, the update is performed according to the message.

- If you click **Just Delete; Skip Update**, Dummy1 is deleted, but only in MyTest; the code corresponding to setting Dummy1 to 1 in MyApplication1 is not removed. This will cause an error when testing MyApplication1.
- By selecting **Always update without confirmation**, you will not receive any more messages when deleting variables, the update will always be performed automatically.

There is a setting reached by the command **Tools > Options** related to the action and message. The setting is "Update other classes when deleting class/component", for details see this setting in the section "Package Browser options tab" starting on page 512.

There are some limitations for this functionality:

- Connections to the deleted variable are not deleted.
- Equations in the Modelica Text layer using the deleted variable are not updated.
- "Undo" cannot be applied for deletion of variables.

Parameter dialog – advanced

The basic handling of the parameter dialog is described in section "Editing parameters and variables" on page 237.

Possible modifiers

The modifiers that appear and can be set in the parameter dialog are:

- Normal parameters declared with the keyword parameter in the model. Please see section "Editing parameters and variables" on page 237.
- Replaceable classes in the model, which can be redeclared to another class (sometimes named "Class parameters"). Suitable choices are presented in form of a drop-down list with the icons of the classes. It is also possible to write the class name directly (e.g. by copying a name from the model-browser before opening the dialogue and then pasting it into this field). One can also add hierarchical modifiers to class name. The choices can be given using the annotation choices as described in the Modelica specification. Note that the declarations inside choices refer to global scope, i.e. must use full lexical path. For more information about class parameters, see the chapter "Introduction to Modelica", section "Advanced modeling features", sub-section "Class parameters".
- Hierarchical and other forms of modifiers, e.g. `x(start=2, unit="m")`. This is used as a catch-all for more complex cases, and can replace values for normal parameters/class parameters.
- Add modifiers. This is a comma-separated list of free form modifiers, and can be used to add free form modifiers for new variables. The next time the modifier dialogue is opened the list elements have been moved to the items. The field does not appear if the model is read-only. For an example, see below.

The General tab: Activation of the dialog entry for start values

This dialog entry for start values in the General tab is activated if one of following is true:

- The default start-value is a non-final scalar literal.
- The actual modifier sets the start-value.
- The variable declaration is annotated with `annotation(Dialog(showStartAttribute=true))`.

Advanced users: The popup-menu can also show the state-selection attribute. This is controlled with

```
annotation(Dialog(__Dymola_stateDialog=true,
showStartAttribute=true)).
```

The General tab: Presentation of enumeration values

If an enumeration value is used for a parameter, a combo box is presented in the parameter dialog. If only enumeration values should be used (and not parameter propagation) the annotation

```
annotation(__Dymola_editText=false)
```

can be used. In such case, only the selected value of the enumeration is shown in the input field, not the entire path including the name of the type.

The General tab: The Custom Parameters group

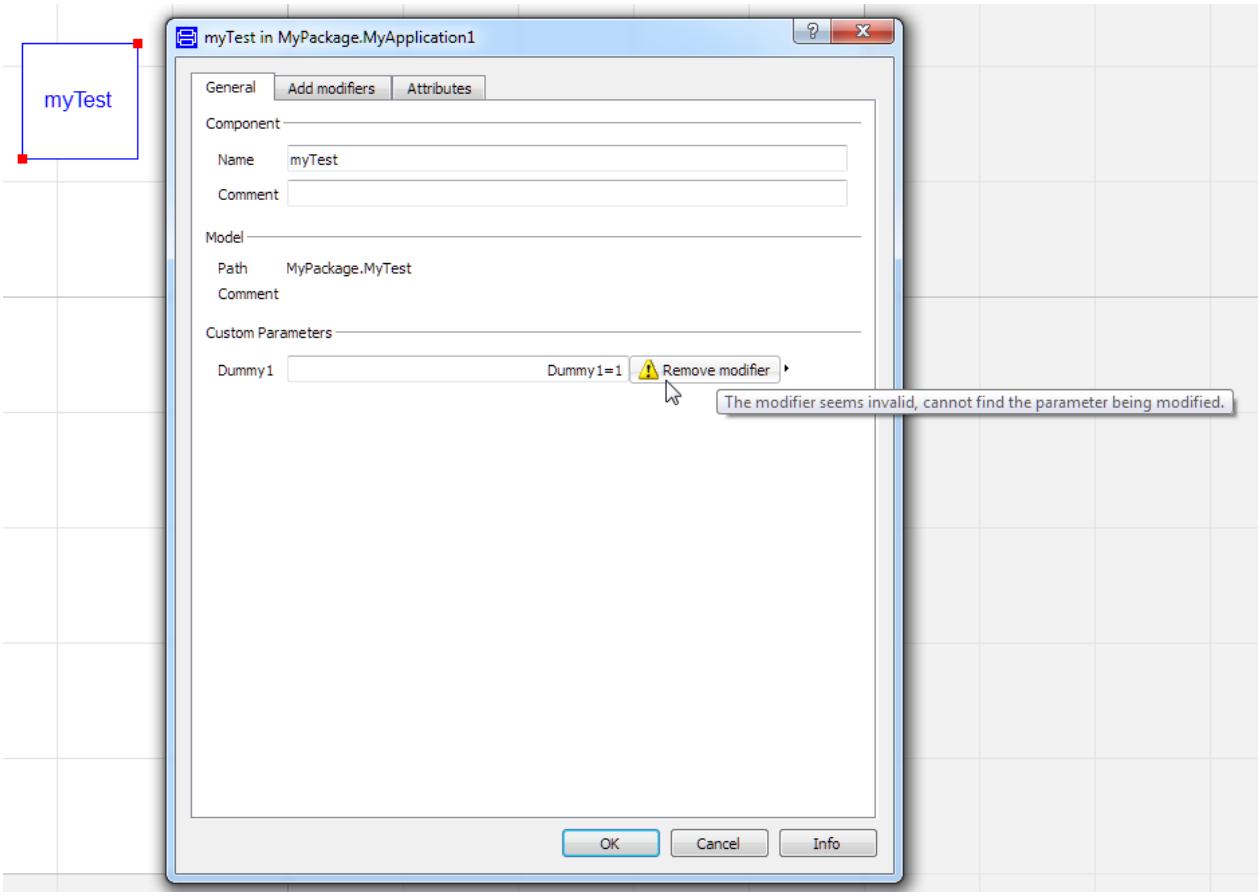
If certain modifiers are present, the group **Custom Parameters** appears after the Initialization group. The modifiers are:

- Modifiers of non-existing variables
- Modifiers modifying variables usually not displayed in the parameter dialog

Modifiers of non-existing variables

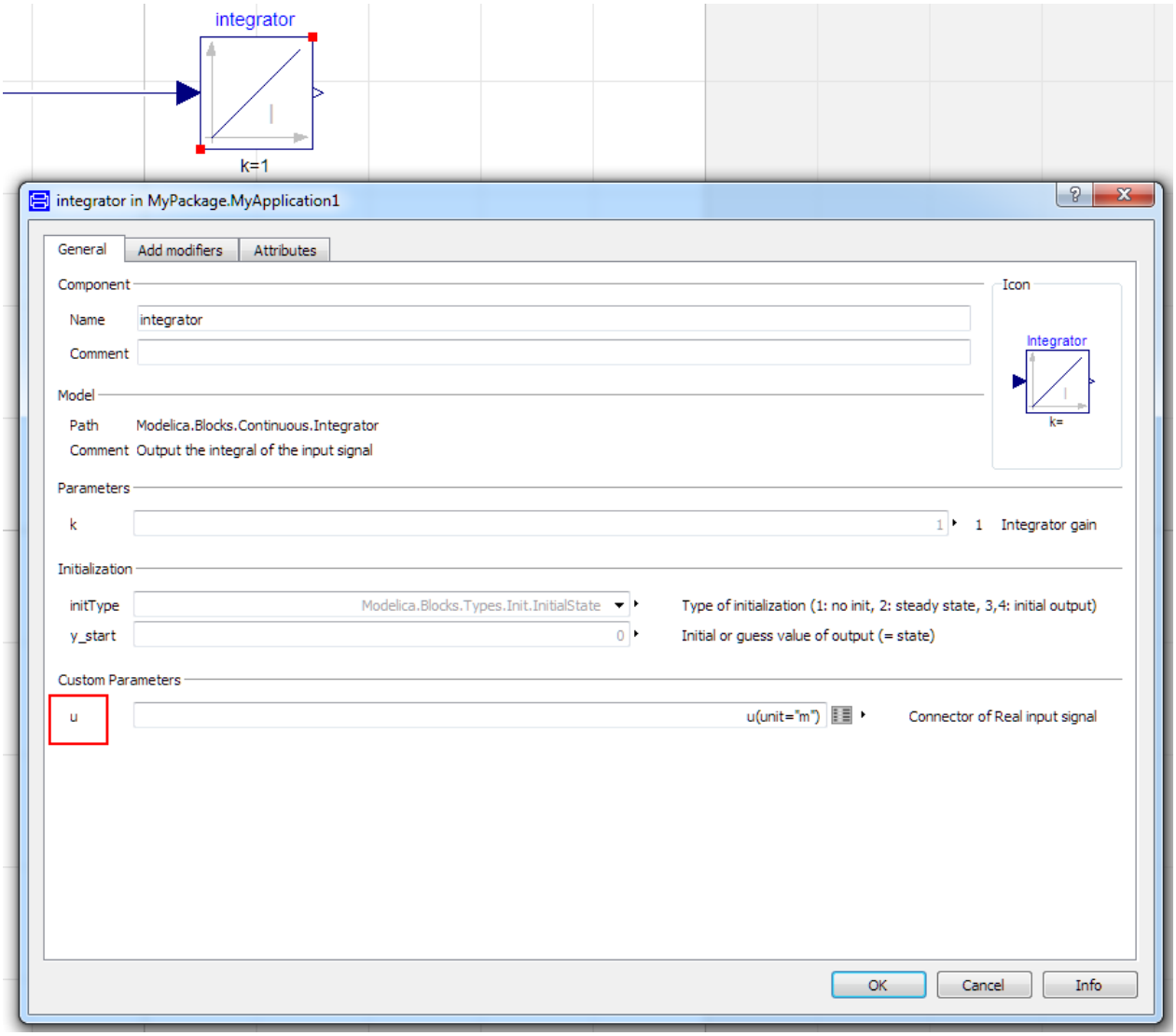
Modifiers of non-exiting variables, typically the result of deleting parameters without deleting the related modifiers, are displayed with a little warning sign and a button **Remove modifier** to delete them.

Below an example where the parameter `Dummy1` has been removed from `MyTest`, without removing the modifier corresponding to setting the value of `Dummy1` to `1` in `Application1`:



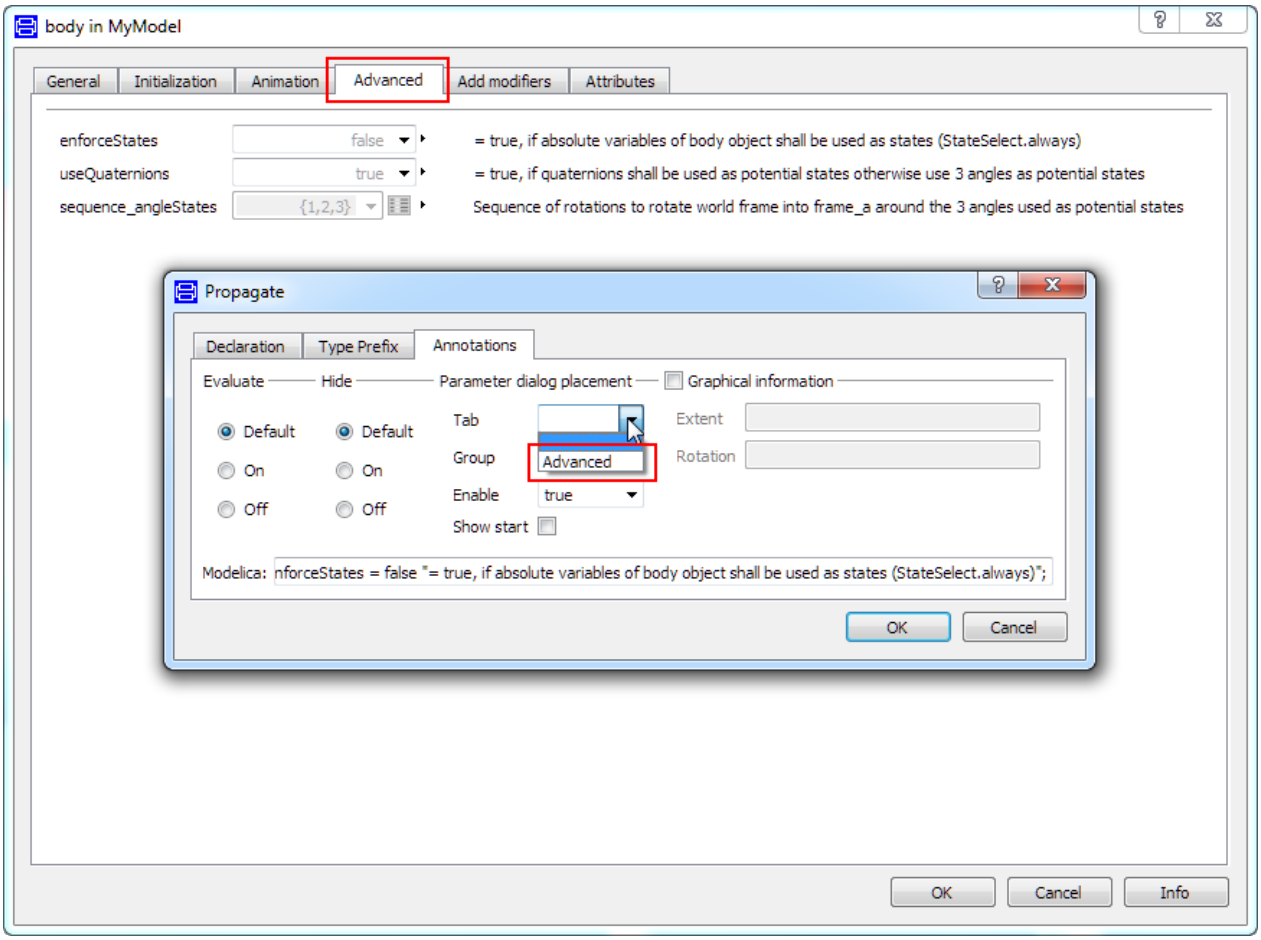
Modifiers modifying variables usually not displayed in the parameter dialog

An example is the input parameter of an integrator component, connected by graphical connection but also modified by adding a unit as a modifier:



Tab selection for propagated parameters

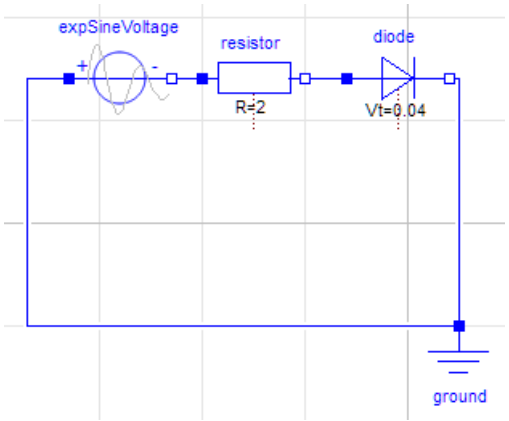
When propagating a parameter, you can select if the corresponding GUI should be located on the same tab as where the parameter is present in the parameter dialog or in the General tab in that dialog (default). As an example, consider propagating the parameter `enforceStates` for a component from the class Modelica.Mechanics.MultiBody.Parts.Body:


body in MyModel
General
Initialization
Animation
Advanced
Add modifiers
Attributes
enforceStates
false
= true, if absolute variables of body object shall be used as states (StateSelect.always)
useQuaternions
true
= true, if quaternions shall be used as potential states otherwise use 3 angles as potential states
sequence_angleStates
{1,2,3}
Sequence of rotations to rotate world frame into frame_a around the 3 angles used as potential states
Propagate
Declaration
Type Prefix
Annotations
Evaluate
Hide
Parameter dialog placement
Graphical information
Default
On
Off
Default
On
Off
Tab
Group
Advanced
Enable
true
Show start
Extent
Rotation
Modelica: nforceStates = false "= true, if absolute variables of body object shall be used as states (StateSelect.always)";
OK
Cancel
OK
Cancel
Info

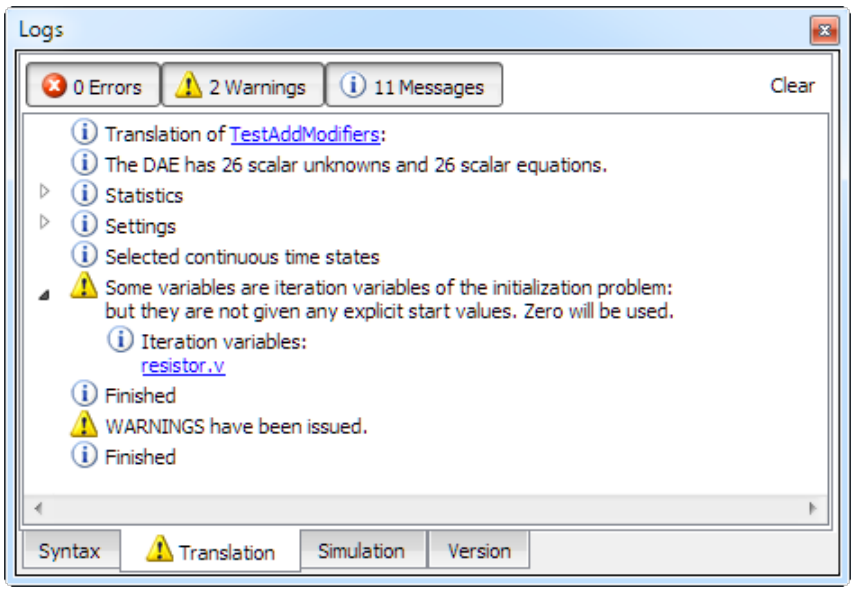
The Add modifiers tab: Using the Add modifiers tab to add modifiers

The basic idea is that the Add modifiers tab is an advanced feature, and should normally not be used; the component parameters in the General or Advanced tab should normally suffice. However, there are some cases where modifiers need to be added, and in such cases the tab is a better alternative than trying to add the modifiers directly in the Modelica code.

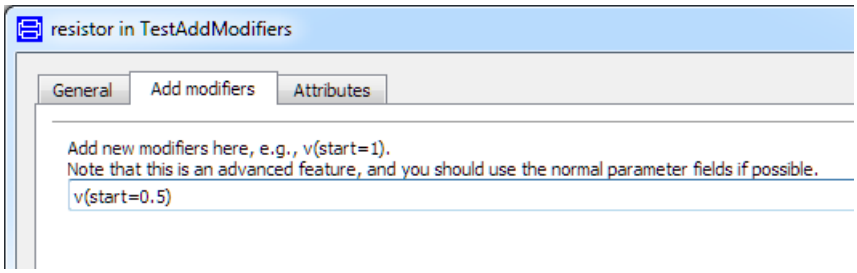
Consider the following simple electrical circuit, where the parameter dialog has been used to give all relevant parameters values for the components:



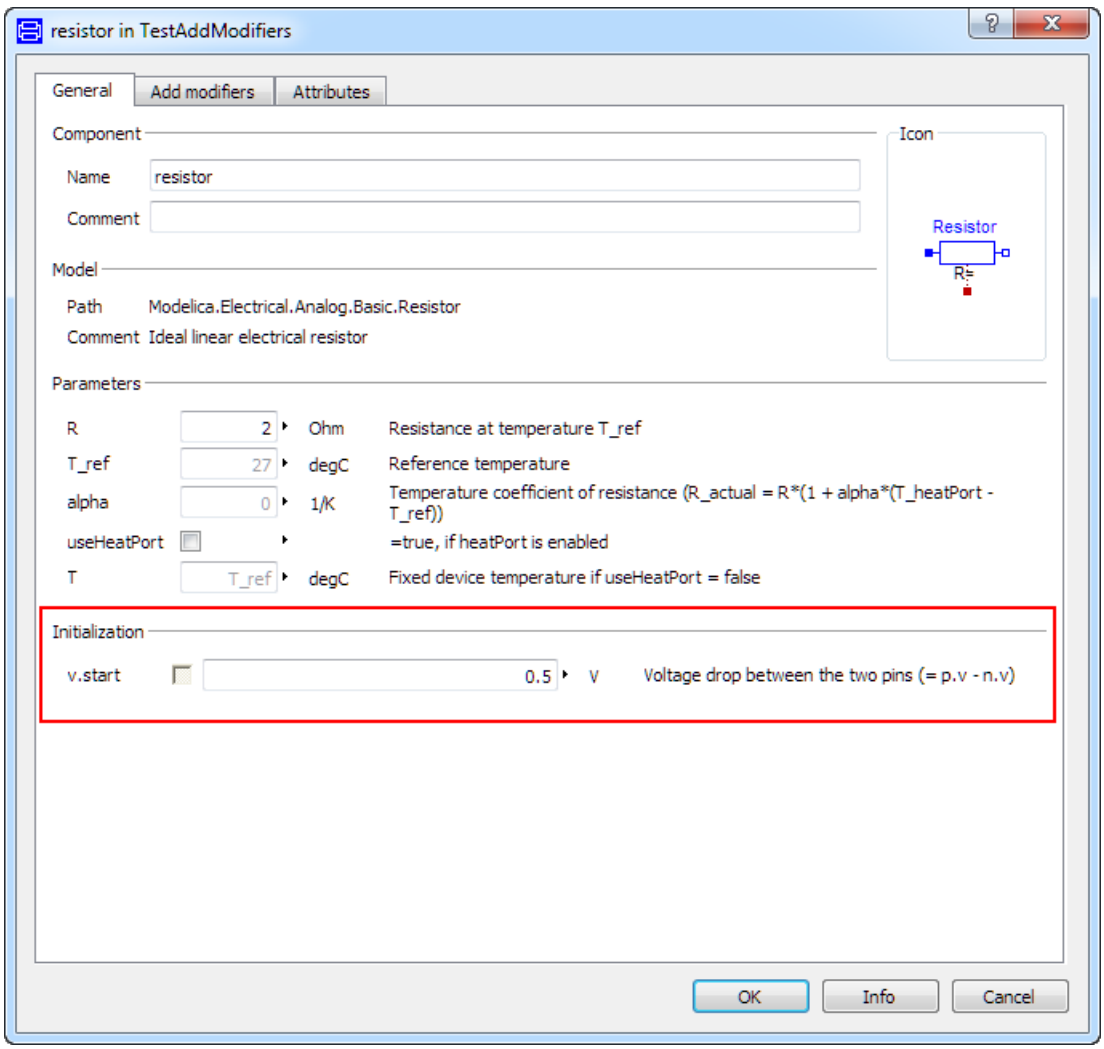
When simulating, the translation message however is:



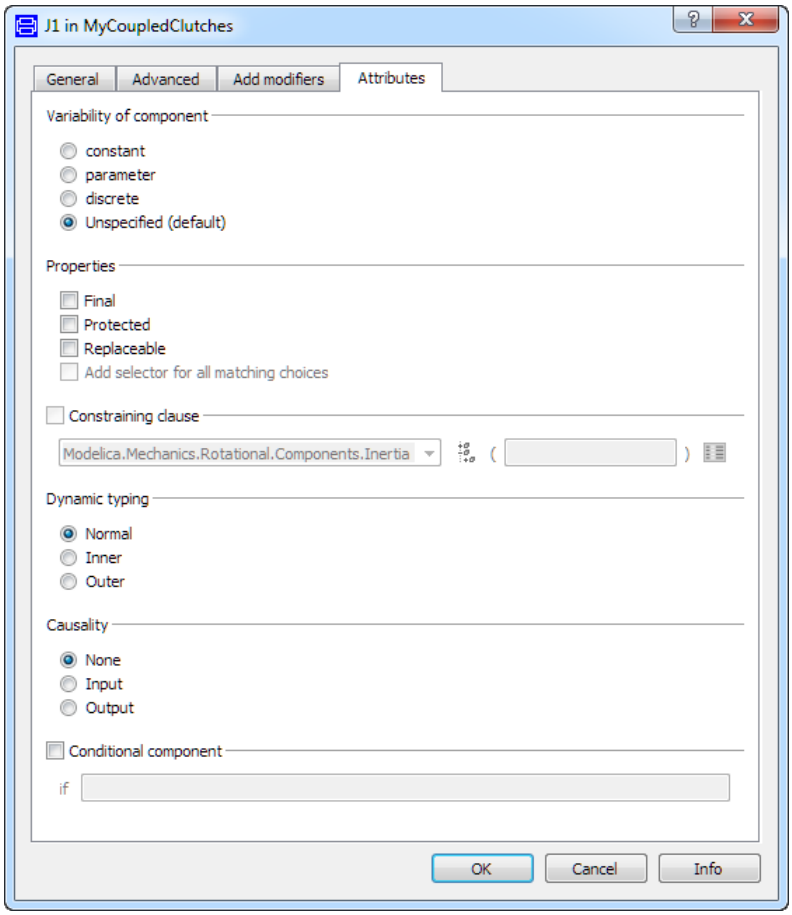
The start value of resistor.v is not available in the parameter dialog of the transistor by default; here a modifier needs to be added. You can use the Add modifier tab for the parameter dialog of the transistor to add the modifier:



When you have done this, the modifier will be added in the General tab of the parameter dialog of the transistor:



The Attributes tab



The attributes tab of the parameter dialog reflects the component properties specified in the Modelica language. The settings in the figure above are the default settings.

The group **Variability of component** is also described when it comes to declaring a variable. Please see the section “Declaration tab” on page 379.

For information about the most of the entries in the groups **Properties**, **Dynamic typing** and **Causality** please see section “Type prefix tab” on page 381. In the **Properties** group there are two additional settings:

- **Add selector for all matching choices** corresponds to the same option when dragging a replaceable component from the package browser to the component browser, adding it as a class parameter. It corresponds to the annotation annotation (`__Dymola_choicesAllMatching`). By ticking this setting a list of matching choices is automatically created when the replaceable component is displayed as a modifier in the parameter dialog of an enclosing class. This list is available as a drop-

down list in the modifier field. See also section "Working with replaceable components" starting on page 346. The setting can only be changed if the component is replaceable.

- **Constraining clause** makes it possible to define a constraining class if **Replaceable** is selected. If a constraining class is defined, only components from this class are possible to select when a redeclare is to be made. Default is the same class as for the component. Parameter settings are also possible to include in the constraints. Such parameter settings will be applied to all possible components. The dropdown list makes it possible to select the constraining clause to be a base class. This can be useful when creating test-driven template models, for an example of such a usage, see section "Defining the constraining clause of the template model components to be the selected base class" on page 375.

In the end of the tab, the setting **Conditional component** is available, together with the condition **if**. This option creates a conditional declaration for the component, as defined in the Modelica language. If the condition is false, the component is removed when translated (including modifiers and connections to it).

Parameter dialog – editing the dialog

The user can do some editing in the parameter dialog.

- Adding tabs.
- Adding groups.
- Hiding variables.
- Display variables as start values.
- Adding alternative form for input fields (combo boxes, radio buttons etc.).
- Adding images in groups.
- Formatting of dialogs (including HTML formatting).
- Adding specialized GUI widgets (for file handling, data input, color handling).
- Using conditional data.
- Checking of input data.

The two first items are briefly described in the section "Annotations tab" on page 382. For more information, and a description of the other features, please see the chapter "User-defined GUI". (Adding tabs and groups in tabs are also described.)

4.3.12 String parameters

String parameters are supported in Dymola. To use string parameters, the following flag must be set:

```
Advanced.AllowStringParameters=true
```

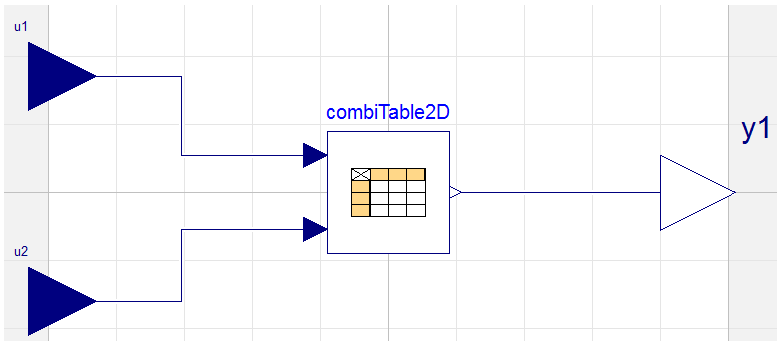
The flag is by default `false`.

(String variables are however presently not supported.)

Example

String parameter support can be illustrated by a simple example of changing tables for a model; consider creating a simple model for linearization.

Create a model; drag an instance of `Modelica.Blocks.Tables.CombiTable2D` into the model. Connect the two inputs and the output and create the corresponding connectors. The result is:



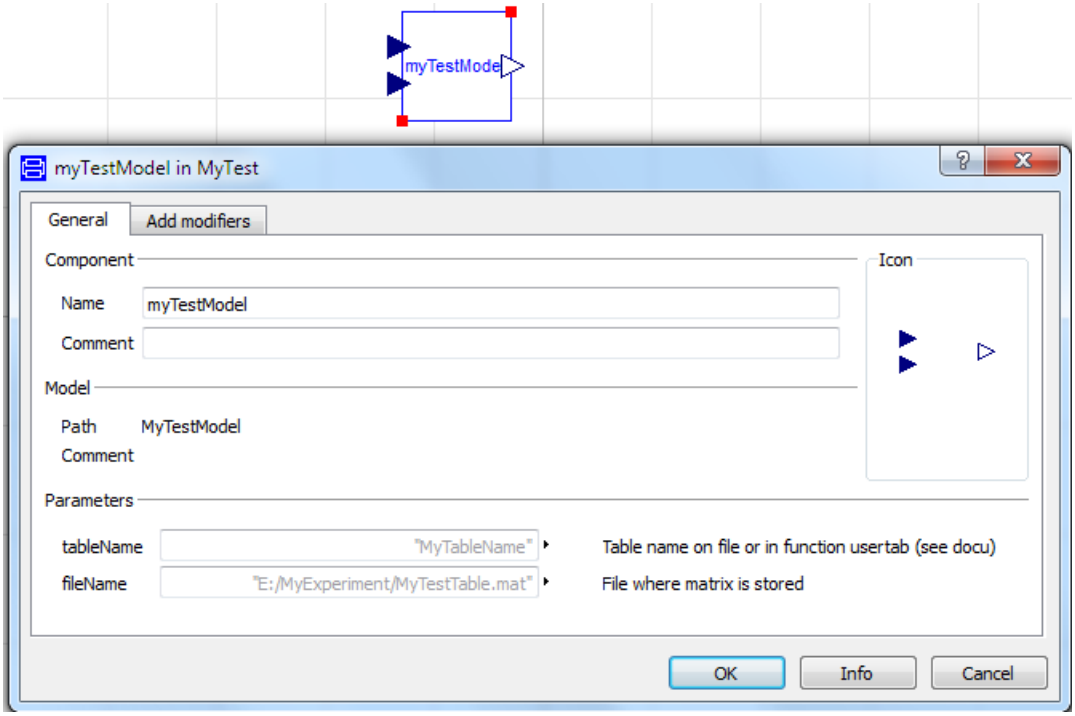
In the parameter dialog of combiTable2D, select **tableOnFile** to true, and propagate **tableName** and **fileName**. Give relevant default values for them. As an example, looking at the resulting Modelica code when having specified a table name and file name as default value, we find:

```
model MyTestModel

  parameter String tableName="MyTableName"
    "Table name on file or in function usertab (see docu)";
  parameter String fileName="E:/MyExperiment/MyTestTable.mat"
    "File where matrix is stored";
equation

  a
end MyTestModel;
```

Instantiating the model and displaying the parameter dialog, we find:

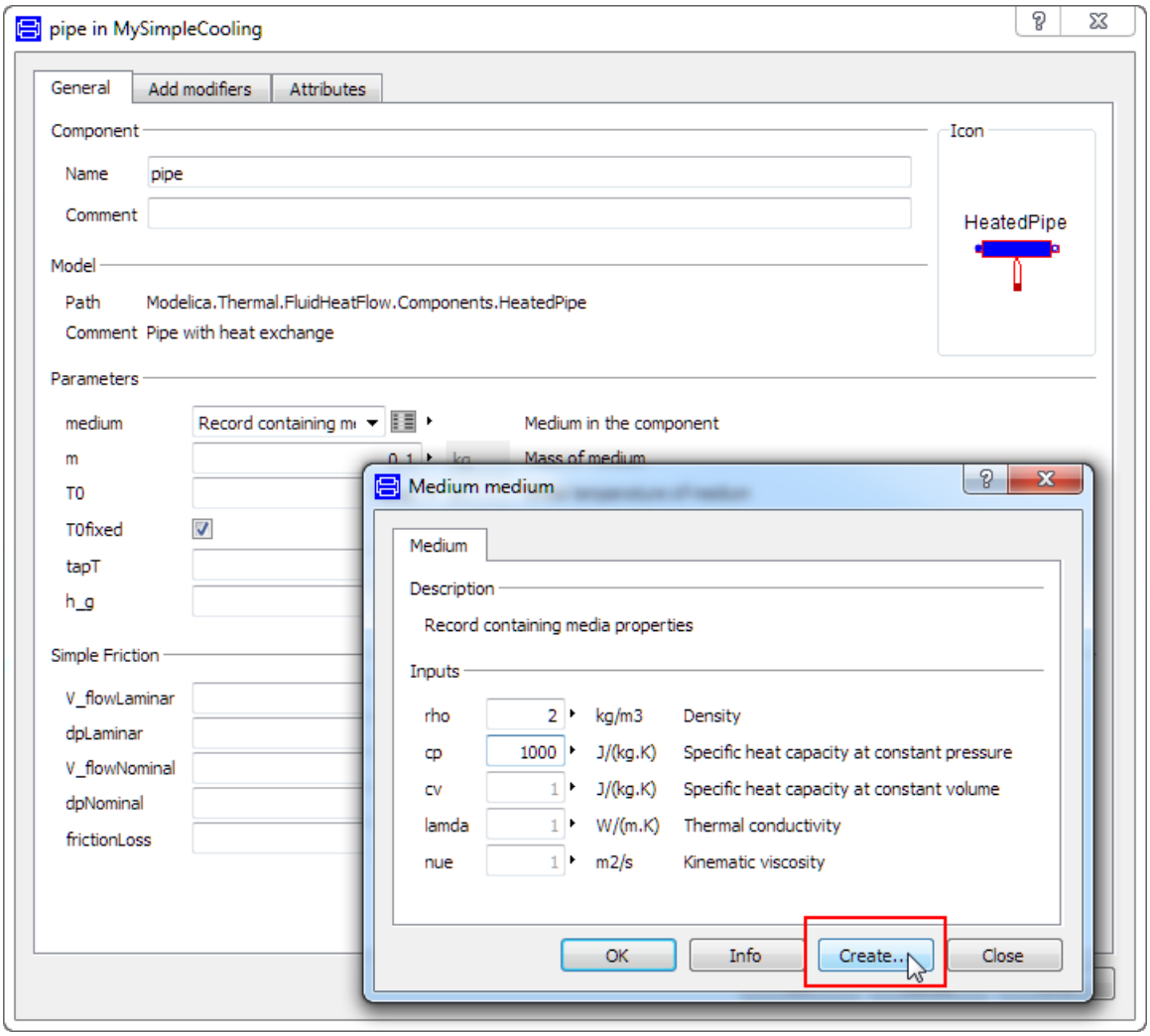


This component supports changing the table name and file name as string parameters.

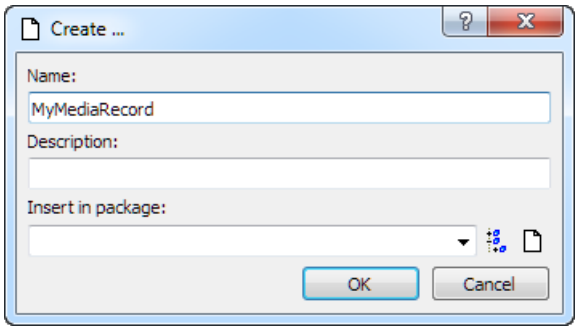
4.3.13 Saving of parameter records

Saving of parameter records can be done by the command Create… in the record constructor dialog. The option can be illustrated by the following example where a media record is saved from a demo.

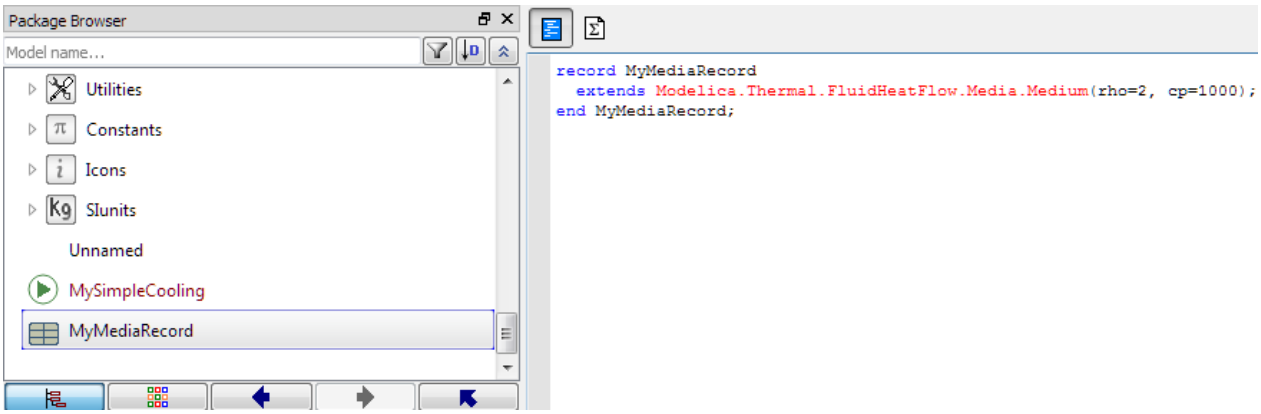
- Open the demo Modelica.Thermal.FluidHeatFlow.Examples.SimpleCooling.
- To be able to work with the records in it, create a new writable model by right-clicking it in the package browser and select **New > Duplicate Class…**. Change the name to MySimpleCooling.
- In the model, right-click the pipe and select **Parameters**. For the parameter **medium**, select **Record containing media properties**, and then click the **Edit** button after the box. The following dialog appear:



- Enter the values shown in the figure above for the first two record components in the media record. Now you can save the record with these values by clicking the command button **Create…** (framed in the figure above). A new dialog appears for saving the record:



- Enter MyMediaRecord as in the figure above. When clicking **OK** a new record is created. Opening it will display the following in the Modelica Text layer:



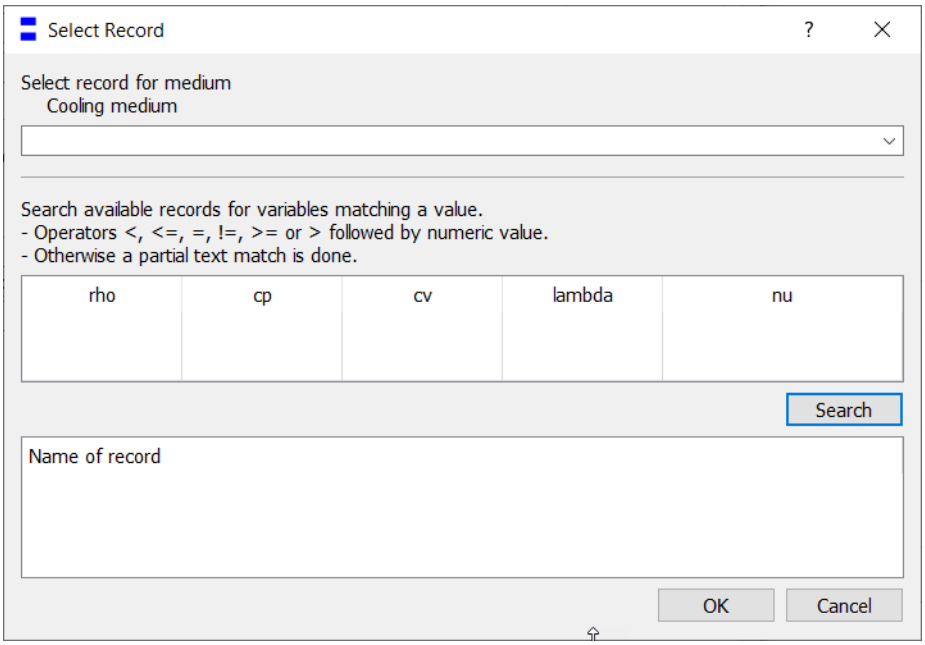
Note that you do not have to change the parameter values of the original record; you can close the first dialog without accepting the changes after you have saved the record.

4.3.14 Searching records

You can search records using a GUI. As an example, do the following:

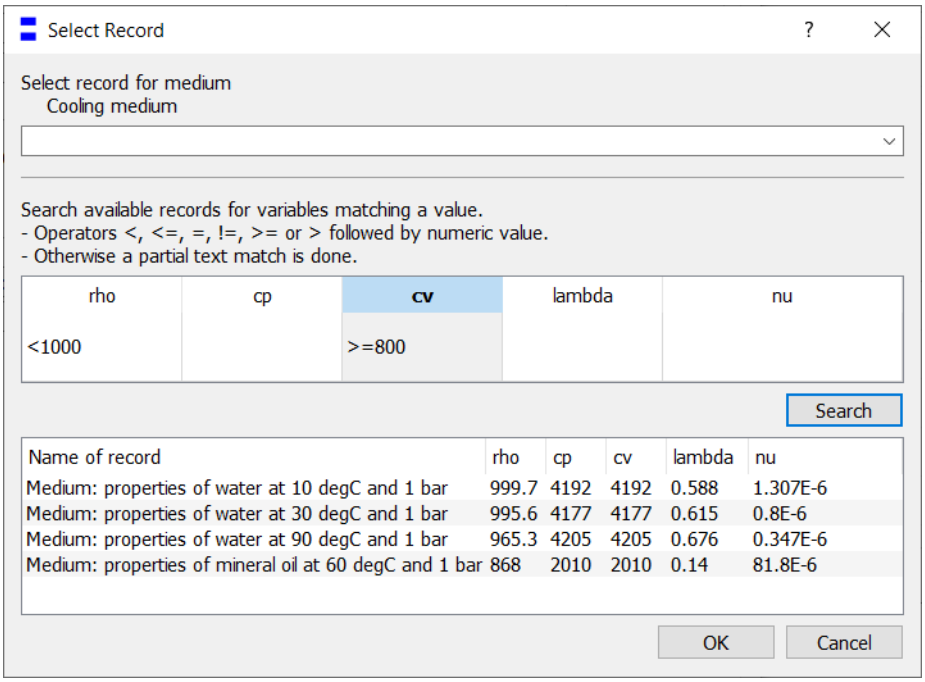
- Open `Modelica.Thermal.FluidHeatFlow.Examples.SimpleCooling`, and create an editable copy of it (by right-clicking in the package browser and selecting **New > Duplicate Class...** and finally clicking **OK** to confrm).
- Right-click the background and select **Parameters**.
- Right-click the **Medium** parameter value and select **Select Record...**.

The following dialog appears:

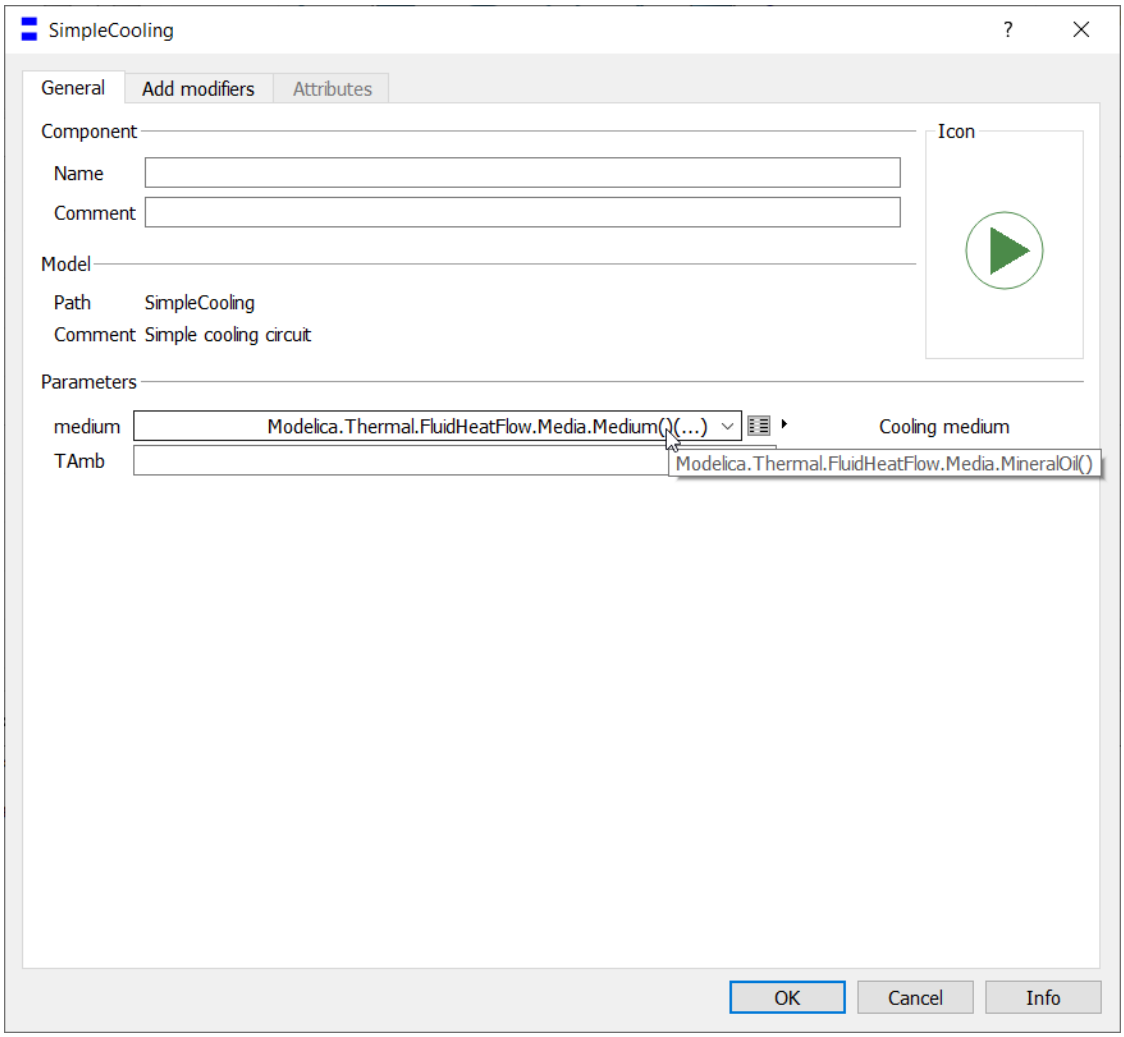


To enter your search criteria´s, double-click the criteria boxes and enter values, using the specified operators (“!=” means “not equal to”). If texts are searched for, a partial text match is done, that is, as an example, the string “dym” matches “Dymola”. All specified criteria must match (logical and).

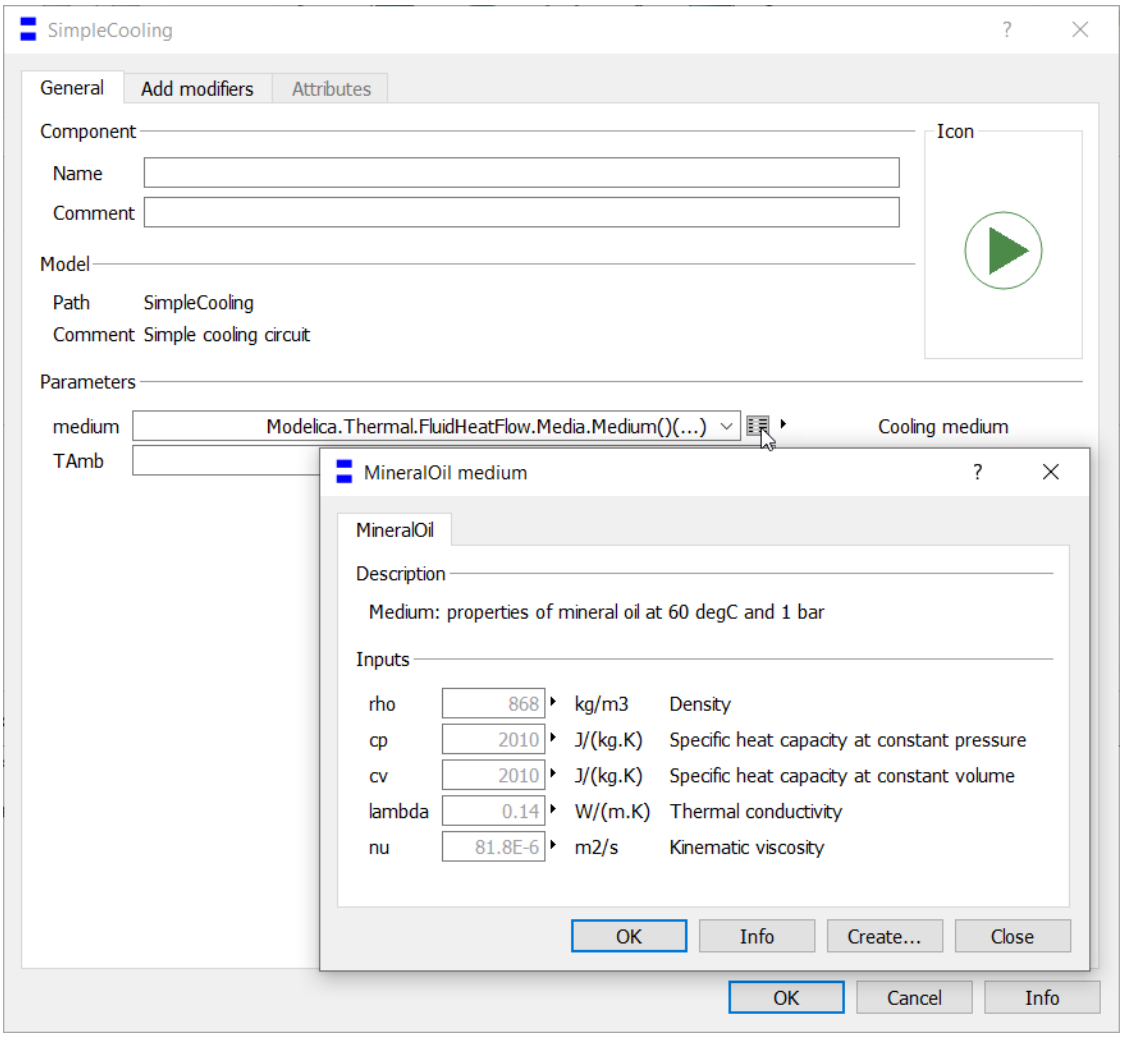
A search example, with **Search** clicked afterwards:



You can now select the wanted record and click **OK** (or double-click the wanted record) to select the record in the Parameter dialog. Assume that you select the last record; the result will be:

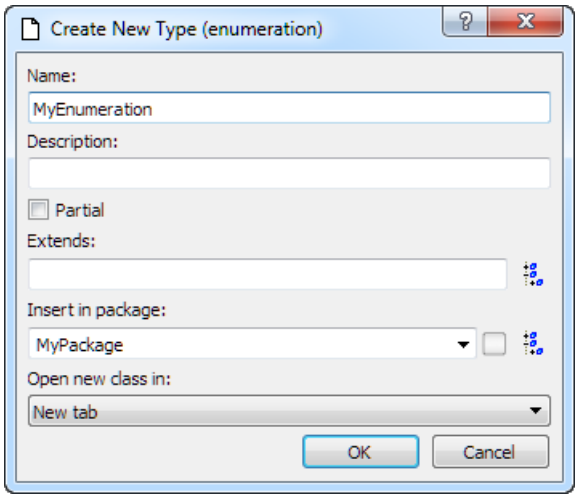


By clicking the button after the medum, you can see the properties:

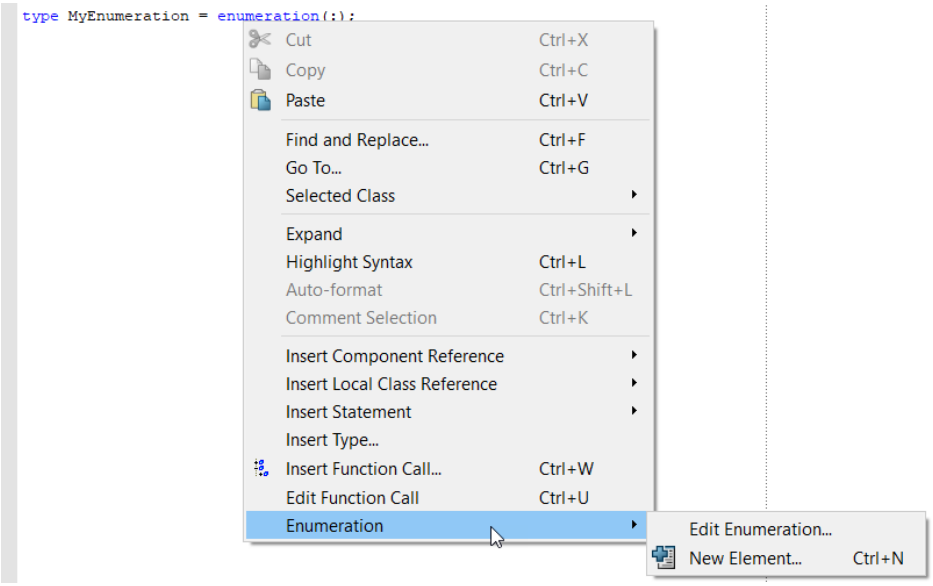


4.3.15 Defining and editing enumeration types

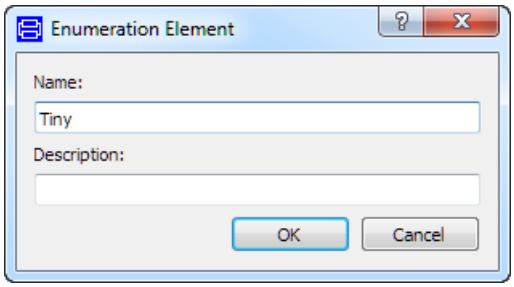
Enumeration types can be defined by the command **File > New > Type (enumeration)**. An example of the dialog:



Once defined, you can right-click the enumeration type in the Modelica text editor:

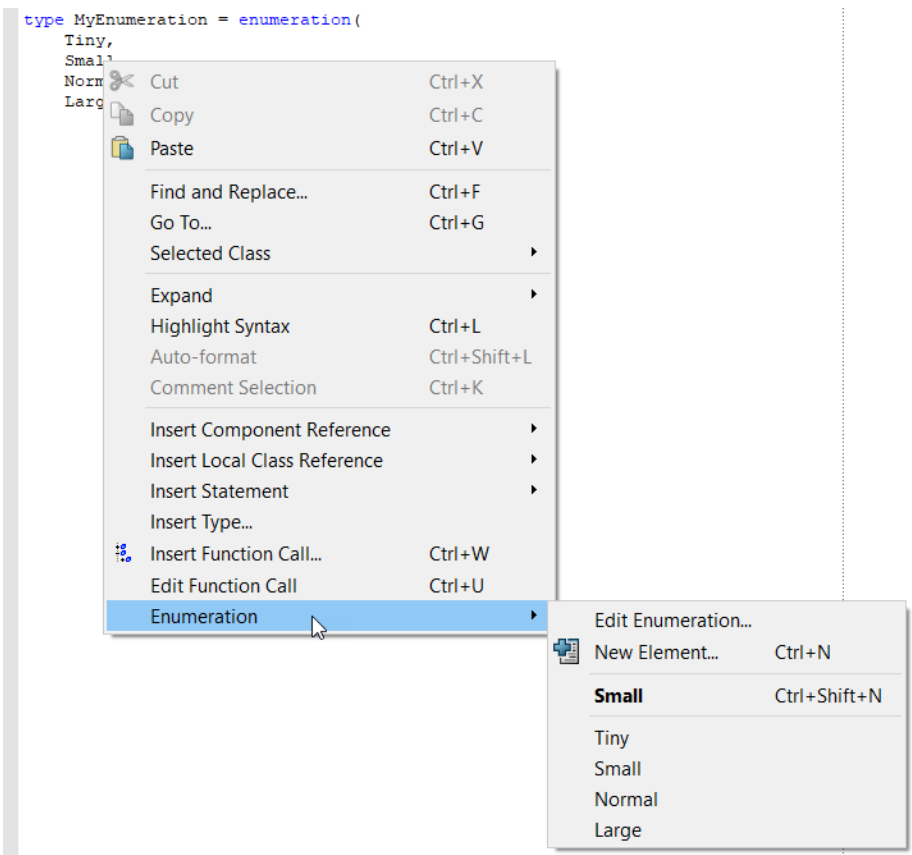


The context command **Enumeration > New Element…** above can be used to define new elements in the enumeration type, an example:



The new element will be the last in the element list.

Defining a number of elements this way, and right-clicking on one of them, e.g. “Small”, gives:



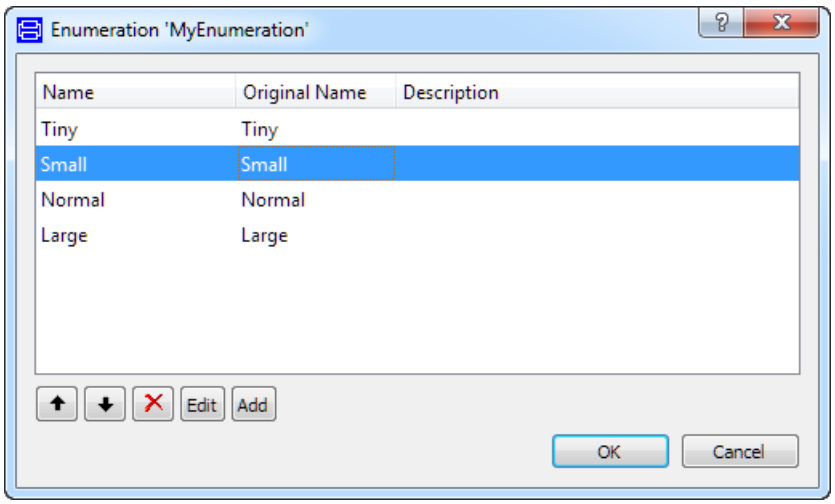
The available Enumeration context commands now also contains a shortcut to the element selected (in bold), and the full element list.

To edit this or another element, click or right-click it in the element list.

Note that you can use **Ctrl+Shift+N** to go directly to the dialog for the enumeration element you pause over in the Modelica Text editor. The dialog is the same as for “Tiny” above.

Renaming an enumeration element will also update use of the element in other models.

The context command **Enumeration > Edit Enumeration…** gives the following for the above example, given we still use the context menu on the “Small” element:



Note that the dialog also presents the original names of the elements.

From this menu you can (using the buttons) sort the list, delete a selected element, edit an element, and add a new element.

Keys can be used for the actions as well, for the selected element:

- Move up: **Ctrl+Up**
- Move down: **Ctrl+Down**
- Edit: **Space**
- Add: **Plus**

For editing an element, you can also double-click it in the list.

4.3.16 Matching and variable selections

General

To simplify postprocessing by reducing the amount of variables saved and displayed in the variable browser when simulating a model, variable selections can be used. You can select which variables to store in the simulation result file and group them logically even if the variables are in fact buried deep in the model.

Note that a complement to variable selection is that if you only need the result file at stop time, you can use the setting **Write variables to result file only at stop time** in the simulation

setup, reached by the command **Simulation > Setup**, the **Output** tab. Please see the corresponding section in next chapter for more information.

Selections are based on pattern matching of variable names, attributes and tags. Tags can be introduced as text annotations for variables and components.

Variable selections are implemented using annotations. The annotations can be put in the actual model, but they can also be put in separate models, enabling great reusability. Selection annotations can be included in several models, not only on the top level. The `override` attribute in the `Selection` constructor (see below) determines if selections with the same name found on lower hierarchical levels of the model should be overridden or merged.

It is important to note that as soon as variable selections are used, there is a default selection **Parameters and States** present. This selection contains interface variables, that is:

- Parameters
- State variables
- Top-level input and output variables (if any)

(For parameters and states the items in the selection correspond to the interactive fields for setting parameters and initial conditions of states.) This selection is automatically created without creation of any corresponding annotations.



What variables can be stored in the result file using variable selections depend on the general settings what variables to store in the simulation result. To access these settings, on the **Simulation** tab, in the **Simulation** group, click **Setup**, and, in the resulting simulation setup dialog, go to the **Output** tab; the settings are in the **Store** group. Please see the corresponding section in next chapter for more information.

By default, if variable selections are present in a model, variables not present in any selection are not stored in the result file; resulting in a decrease of file size, valuable for large models. Whether non-selected variables should be saved in the result file is controlled by the setting **All variables; ignore selections in model** in the **Store additional variables group** in the simulation setup (just after the **Store** group, see above how to reach this group). Please see the corresponding section in next chapter for more information.

Below the rules for variable selections are presented. A user interface is available for creating and editing variable selections. See section “User interface editing variable selections” starting on page 413.

When simulating, the user can select what selections to display in the variable browser, by commands in a panel in the variable browser. Please see section “User interface for displaying variable selections in the variable browser” on page 419 for example and further reference.

Annotations

Two annotations are available, one for defining selections and one for defining tags.

Selections

The annotation for selection is:

```
annotation(__Dymola_selections={Selection(...),...});
```

where the selection constructor is:

```
record Selection
  parameter String name;
  parameter Boolean override=false;
  parameter MatchVariable match[:]=fill(MatchVariable(),0);
end Selection;
```

and the matching constructor is:

```
record MatchVariable
  parameter String className="" "Regular expression for classes
    to look in";
  parameter String name="*" "Regular expression for variable
    name";
  parameter String description="" "If non-empty, regular
    expressions that should exactly match description";
  parameter String tag[:]=fill("",0) "If non-empty, regular
    expressions that should exactly match any tag";
  parameter String newName="" "For storing under different
    name";
end MatchVariable;
```

Tags

The annotation for tags is:

```
annotation(__Dymola_tag={"Tag2", "Tag3"});
```

If a tag is attached to a component, it is also attached to sub-components and variables.

Matching can then be made on tags as well.

Rules for matching and substitution

There are some rules for the matching and substitution. For illustration, refer to the example section below.

All matches must be exact, and only scalar variables are considered for the matching, however, tags (see below) are inherited by sub-components.

Regular expressions can be used in all fields in the `MatchVariable` constructor. Special symbols in the regular expression are:

*	Match everything.
?	Match any single character.
{ab}	Match characters a or b.
{a-z}	Match characters a through z.
{^ab}	Match any characters except a and b.
E+	Match one or more occurrence of E.
(ab|cd)	Match ab or cd.
\d	Match any digit.

\w	Match any digit or letter.
^	Match from start.
$	Match at end.

The `name`, `description` and `tag` parameters in the `MatchVariable` constructor support leading "!" for selecting non-matches.

The parameter `newName` in the `MatchVariable` constructor can use `%componentPath%`, `%name%`, `%path%`, `%componentName%`, and `%i%`.

The `%name%` corresponds to the part matching the regular expression and `%componentPath%%name%` is the full path of the variable – and `%componentName%` the last part of `%componentPath%`. If you use `%componentPath%` as the final part of the pattern, the ending "." will automatically be removed.

An example of the use of `%i%` is that `%1%` is the part matching the first parenthesis-enclosed part of the regular expression.

Local pattern matching in certain classes is supported, refer to the examples below.

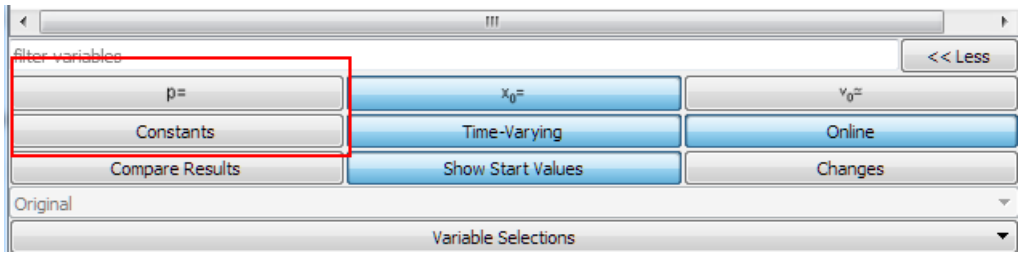
Modelica text examples

The following Modelica text examples illustrate some features.

Note that the Modelica text examples of selections below correspond to the graphical user interface examples in section "GUI examples" starting on page 416.

To simplify the code and the presentation in the variable browser, the following adoptions have been made:

- In Modelica text, basic graphical annotations have been deleted.
- Comments have not been used.
- The `override` attribute has always been set, since this corresponds to most real user cases, and is set by default when using the graphical user interface to create selections.
- In order to easily change the selections, they are put in separate models, e.g. Selection1.
- To hide parameters and constants in the variable browser for the examples where the selections are not also presented in additional content of the variable browser, the following changes in the default setting of the advanced mode of the variable browser have been made when creating the images:



Note that by default the variable selection Parameters and States is present; see section “User interface for displaying variable selections in the variable browser” on page 419.

(If you want to see only the selected variables, use the possibility to present selections also as additional content in the variable browser. Such additional content always only displays the selected variables. See “Selections – introducing new names creating additional content in the variable browser” on page 412)

The selections are applied, in these examples, on the model example fullRobot, by extending the demo **File > Demos > Robot** by right-clicking fullRobot in the package browser and use the command **New > Extend From…** to create a new model, MyFullRobot, and in that model also add a line extending the selection model. It might look the following (in this case applying Selection1, described further below):

```
model MyFullRobot
  extends Modelica.Mechanics.MultiBody.Examples.Systems.RobotR3.fullRobot;
  extends Selection1;
end MyFullRobot;
```

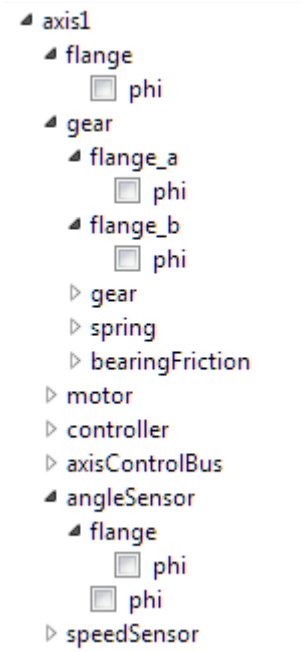
Selections - basic functionality

(The following models are to be extended like above.)

A selection of all variables called “phi” in a model can be made by using the following annotation:

```
model Selection1
  annotation (__Dymola_selections={Selection(
        name="MySelection1",
        override=true,
        match={MatchVariable(name="*.phi")})});
end Selection1;
```

Activating this selection by extension results in the following content of the variable browser when simulating the model MyFullRobot:

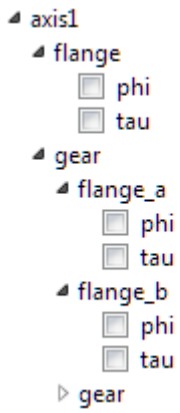


In addition to the selection, parameters and states are preselected. See “User interface for displaying variable selections in the variable browser” on page 419.

Including several variables in the selection can be made by using “|” in the regular expression for name:

```
model Selection2a
  annotation (__Dymola_selections={Selection(
        name="MySelection2a",
        override=true,
        match={MatchVariable(name="*.(phi|tau)")})});
end Selection2a;
```

resulting in:

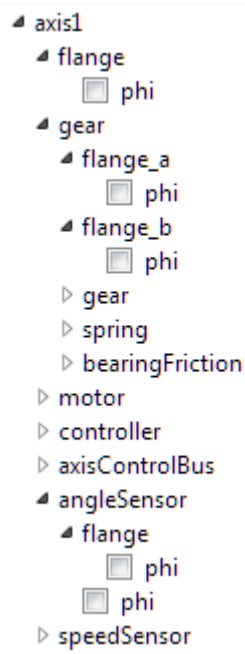


Alternatively of using "|", several matches can be used. They are or ´ed.

```
model Selection2b
  annotation (__Dymola_selections={Selection(
        name="MySelection2b",
        override=true,
        match={MatchVariable(name="*.phi"),MatchVariable(name="*.tau")})});
end Selection2b;
```

Selections - matching within a scope

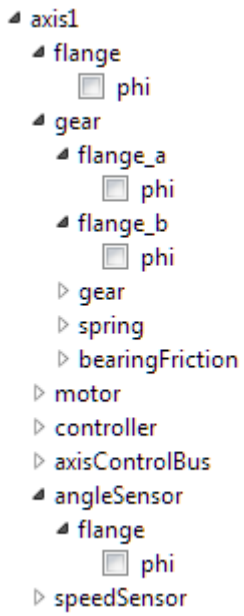
It can be noted that the annotation in the model Selection1 gave the result:



i.e. the local variable `phi` of angleSensor is present. The reason was that the search pattern given was "`*.phi`", i.e. any prefix to phi. The pattern matching is performed from the top level model so the path "`axis1.angleSensor.phi`" was found. It is possible to make the pattern matching locally in certain classes only, by providing a (pattern for the) class name. In this case the pattern is just "`phi`".

```
model Selection3
  annotation (__Dymola_selections={Selection(
        name="MySelection3",
        override=true,
        match={MatchVariable(name="phi",
            className="Modelica.Mechanics.Rotational.Interfaces.Flange*")})});
end Selection3;
```

The result is then:



Notes:

- If you change the className string in the match above to `className="Modelica.Mechanics.*.Interfaces.Flange*"` you match both rotational and translational flanges.
- You can filter on specific components by specifying them in the name string in the match, for example, if you change the name string in the match above to `name="axis1.*.phi"` you match only variables `phi` in subcomponents of the component `axis1` (note that you will not match `axis1.phi`).

- You can filter on variable descriptions by adding a line in the match, for example, `description="*ratio*"` will filter out all variables not containing `ratio` in the description.

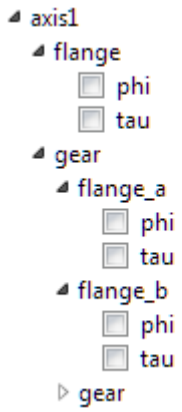
Be careful when selecting both class and component, note that the class selection and the component selection must work with the same component if in the same selection.

Selections – introducing new names creating additional content in the variable browser

The `MatchVariable` has an attribute `newName`. When `newName` is used, the variable is included in a new subtree with the name of the selection. To use the same variable name as before you can use `newName="%componentPath%%name%"` or alternatively `newName="%path%"`.

```
model Selection4
  annotation (__Dymola_selections={Selection(
        name="MySelection4",
        override=true,
        match={MatchVariable(name="*.phi", newName="%componentPath%%name%")})});
end Selection4;
```

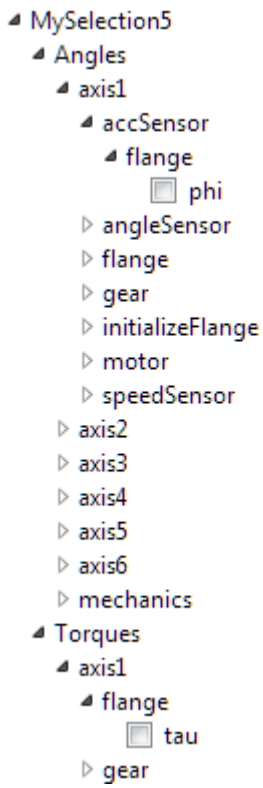
It will give the same result above, but *also* the following *additional* content of the variable browser:



It is possible to build subtrees in the additional content in the variable browser, by including a subtree name in the `newName` path:

```
model Selection5
  annotation (__Dymola_selections={Selection(
        name="MySelection5",
        override=true,
        match={MatchVariable(name="*.phi", newName="Angles"),
               MatchVariable(name="*.tau", newName="Torques")})});
end Selection5;
```

resulting in:



Tags

Variables can be tagged using the tag annotation:

```
Real w annotation(__Dymola_tag={"Mechanical"});
```

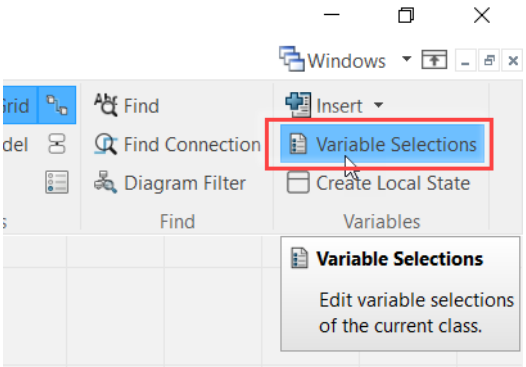
Several tags can be associated with a variable:

```
Real w annotation(__Dymola_tag={"Mechanical","Rotational"});
```

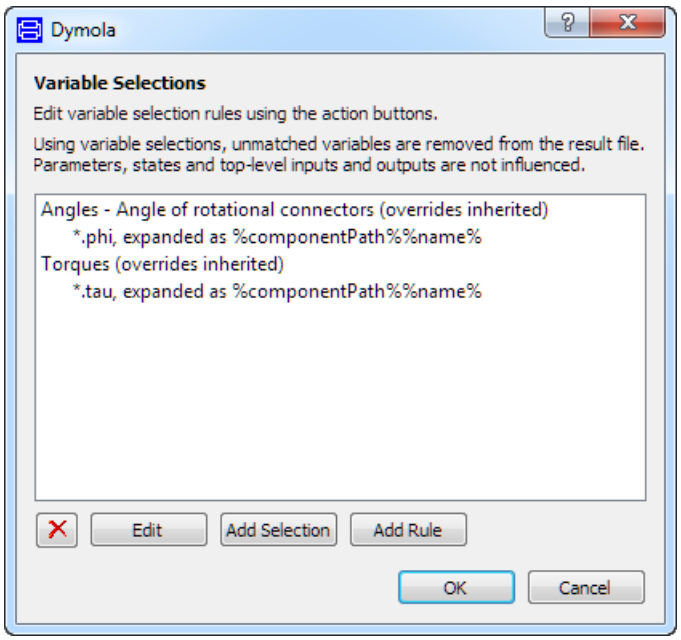
User interface editing variable selections

General

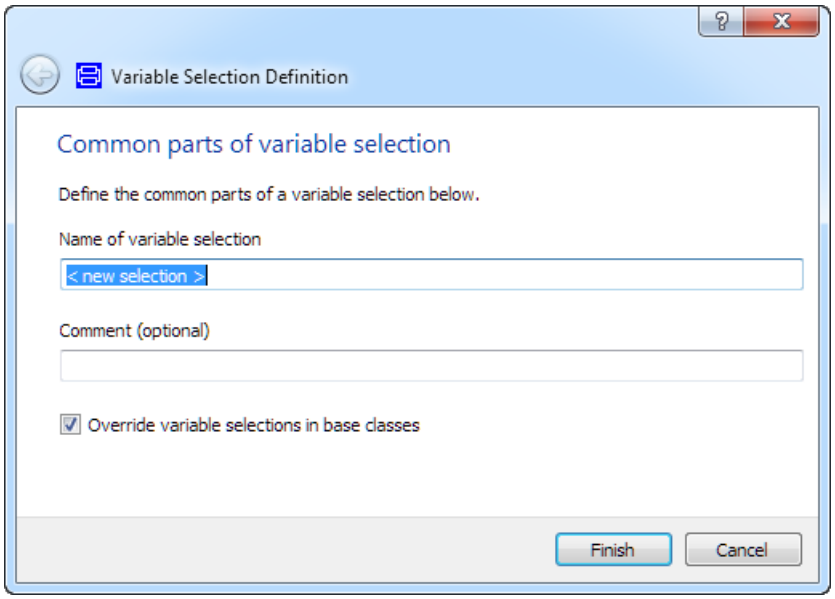
To display a dialog summarizing all available variable selections in the model, on the **Graphics** tab, in the **Variables** group, click **Variable Selections**:



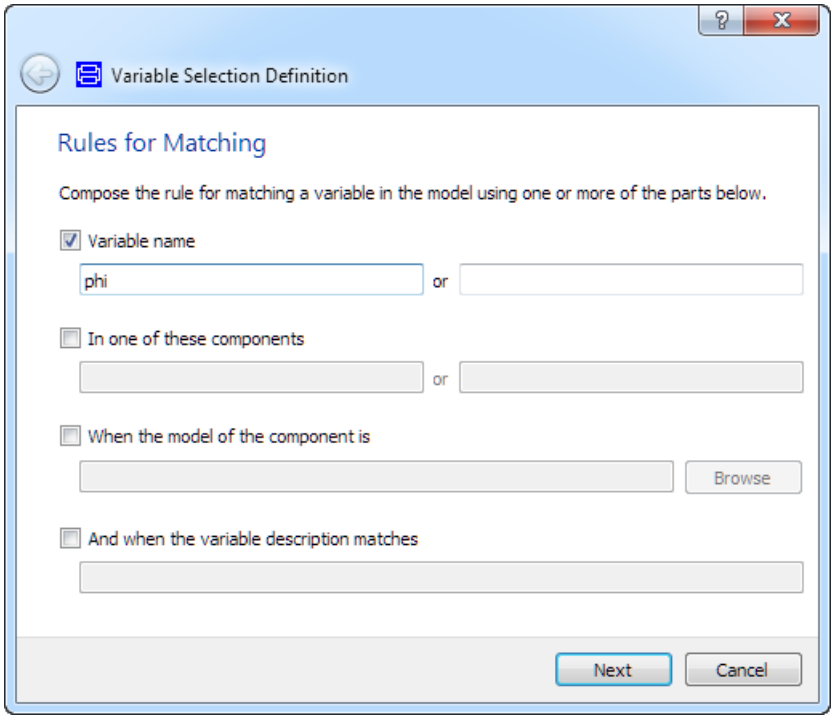
There can be multiple selections, each containing one or more matching rules:



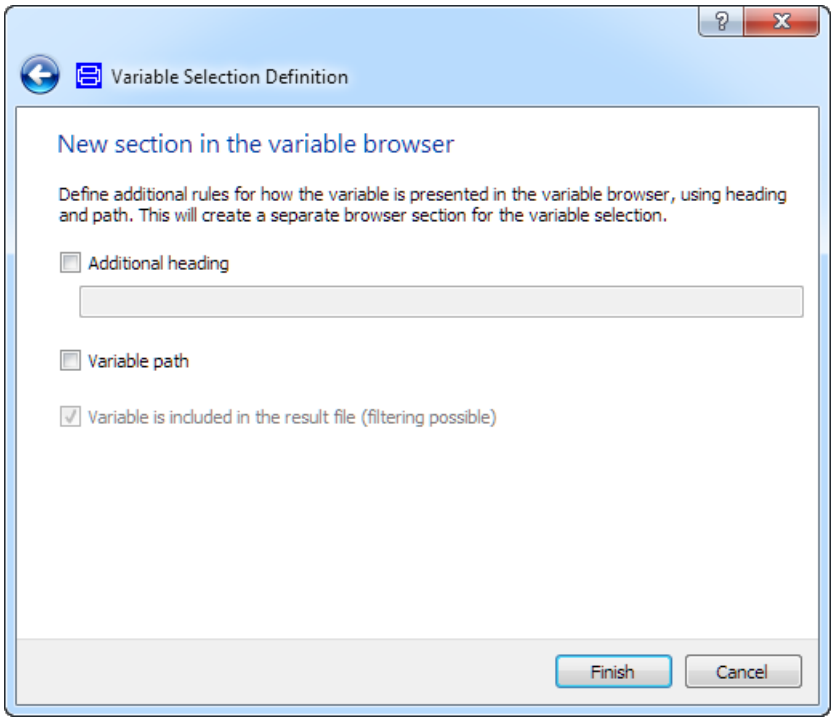
Pressing **Add Selection**, or selecting an existing selection and pressing **Edit** opens a dialog to enter the name of a selection and an optional comment. It is also possible to define that this variable selection shall override any variable selection with the same name inherited from base classes (this is the default).



Pressing **Add Rule**, or selecting an existing rule and pressing **Edit**, starts a wizard to define a rule in a variable selection. The first page defines the rules for matching variables in a model, based on variable name, the component or model that contains the variable, or the variable description.



The next page of the wizard is used to define how the variable will be presented in the variable browser after reading the result file in Dymola. It is possible to define an additional heading and to include the path of the variable. If neither heading nor path is selected, the variable selection can be used for filtering, but no separate section is created. Note that by default parameters and constants are displayed as well in the variable browser, the only way to be sure to be able to always only see the variables of the selections is to create additional content in the variable browser and look at that content For details, see the last item in the item list in the beginning of “Modelica text examples” on page 407.

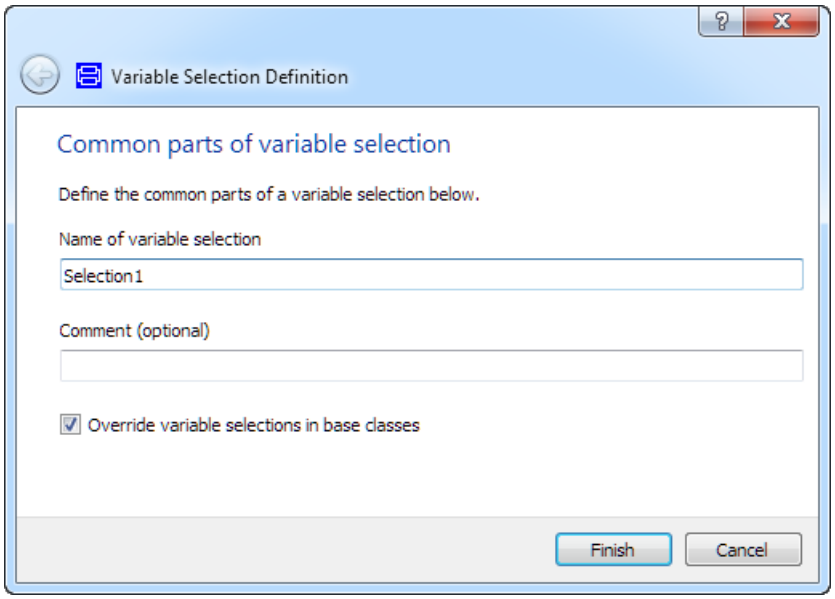


GUI examples

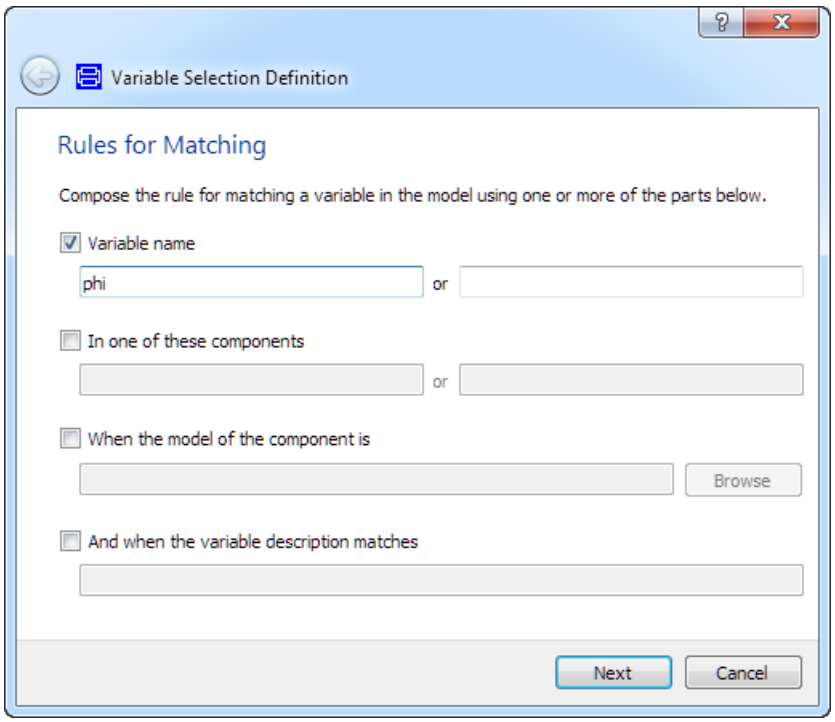
The Modelica text examples of variable selections in section “Modelica text examples” starting on page 407 can be created using the graphical user interface. To create the Modelica text examples above, you can use the following GUI examples:

Selection1

To create the model Selection1 (selecting all variables `phi`), use **File > New > Model** to create a model, name it Selection1. Then, on the **Graphics** tab, in the **Variables** group, click **Variable Selections** to display the menu for creating variable selections. Click **Add Selection** and define the following:



Click **Finish**, and in the dialog that is the result, select MySelection2a and click **Add Rule**. Define the following in that dialog:



Click **Next** to display the final dialog, but here just click **Finish**. Click **OK** to finalize the creation.

Selection2a

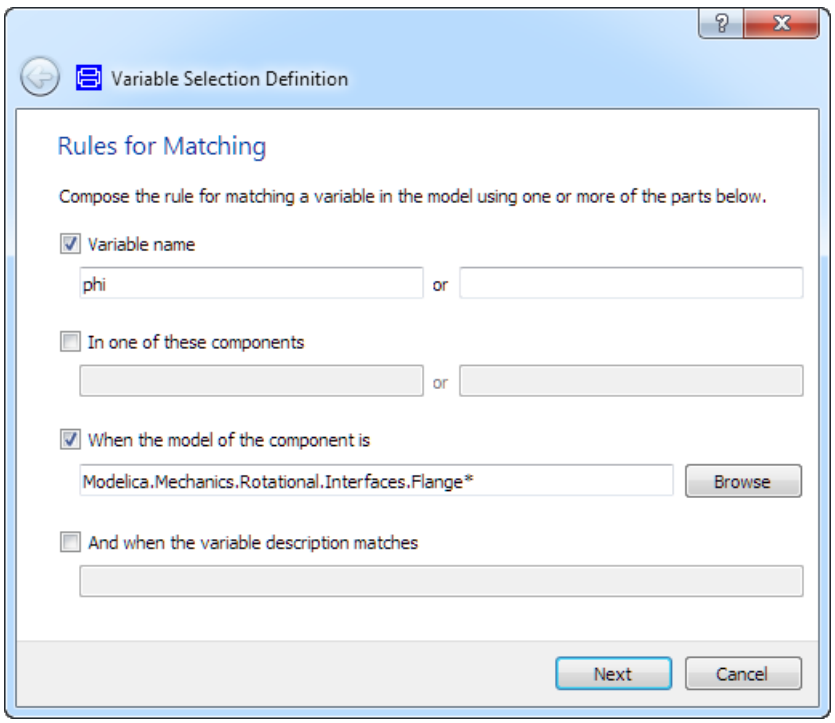
To create the model Selection2a (selecting both `phi` and `tau`), use the above description but name the selection Selection2a, and enter `tau` in the second box for the variable name.

Selection2b

To create the model Selection2b (an alternative way to select `phi` and `tau`), use the description how to create Selection1 above, but also add an additional rule specifying `tau` the same way as `phi` was specified.

Selection3

To create the model Selection3 (selecting phi in any Modelica `Flange` class), the rules dialog should be like this:



The easiest way is to browse for, for example, `Modelica.Mechanics.Rotational.Interfaces.Flange_a` and then change the last part to `Flange*`.

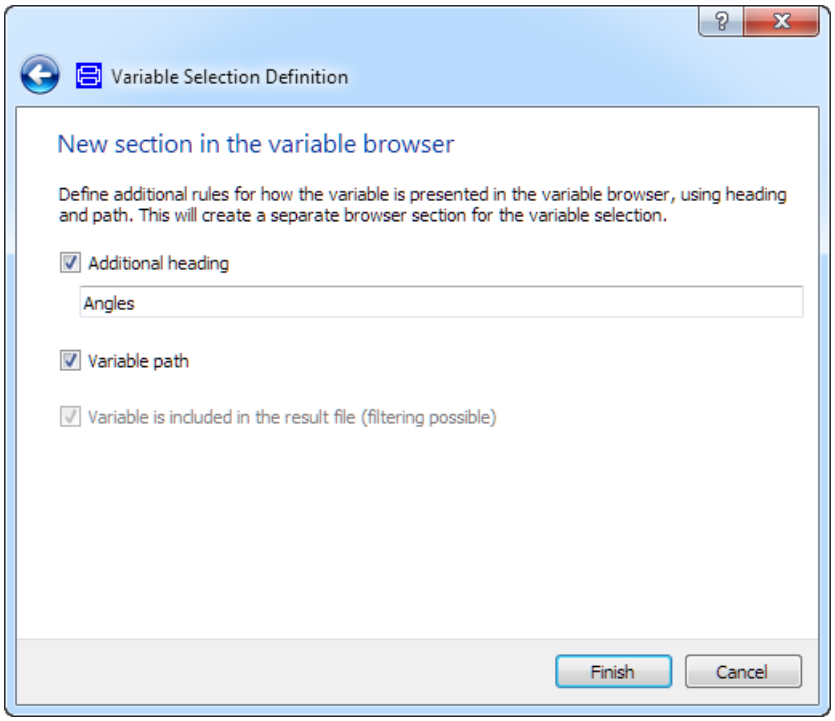
Note that are some further examples in the "Notes" after this example; the first one corresponds to altering the class path similar to the above, the second corresponds to working with **In one of these components**, and the last one corresponds to **And when the variable description matches** in the dialog above.

Selection4

To create the model Selection4 (creating, in addition to the selection, a new section in the variable browser with the selection), you can create a model like Selection1 above, but in addition also tick **Variable path** in the last dialog of the rule creation.

Selection5

To create the model Selection5 (creating, in addition to the selection, a new section with subtrees in the variable browser), you can create a model like selection 2a above, but in addition change the last dialog of the rule creation for both rules, to specify the corresponding subtrees **Angles** and **Torques**. As example, the dialog for the rule handling phi selection, defining the **Angles** subtree:



The corresponding changes must be done for the tau selection, to create the **Torques** subtree.

User interface for displaying variable selections in the variable browser

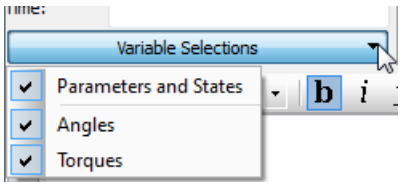
The below is an example of user interface in when simulating a model for displaying selections in the variable browser, for more information about this please see next chapter, section “Model simulation”, sub-section “Variable browser interaction”, part “Advanced mode”.

When simulating a model, it is possible to get a list of all selections and to select which of these selections that should be included in the variable browser.

As an example, consider the below model where two selections have been defined:

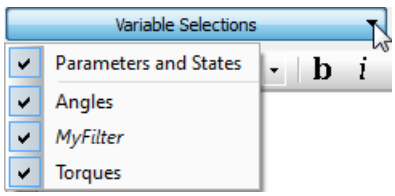
```
model Selection6
  annotation (__Dymola_selections={Selection(
        name="Angles",
        override=true,
        match={MatchVariable(name="*.phi", newName="%componentPath%%name%")}),
        Selection(
        name="Torques",
        override=true,
        match={MatchVariable(name="*.tau", newName="%componentPath%%name%")})});
end Selection6;
```

This corresponds to the following items in the advanced filter menu in the variable browser:



The variable selection **Parameters and States** is predefined, containing parameters, state variables, and (if any) top-level input and output variables. See “General” on page 404 for more information.

In the above example, the variable selections are part of the currently active simulation results in the variable browser. Selections *not* part of the currently active simulation results in the variable browser are indicated by italics, like **MyFilter** below:



4.3.17 Using data from files

Data files (tables etc.) can be read and written from Dymola in several ways. Common file formats are:

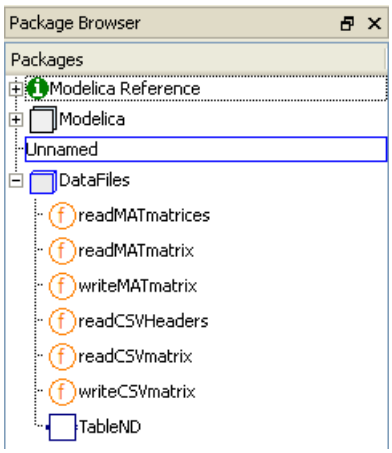
- Matlab 4 files (.mat).
- Comma separated values files. Comma separated values files (used by e.g. Microsoft Excel). We recommend the file extension .csv but .txt format is also supported. In the latter case there might also be a choice of separator characters.
- Scientific Data Format (SDF) files. For more information on this format, see section “SDF editor” below.

Matrix editor

The **Edit** menu of variables/parameters can import/export data from files. Please see section “Edit menu of a matrix of values (matrix editor)” on page 561. Matlab 4, CSV and text files can be handled.

DataFiles package

The content of DataFiles package.



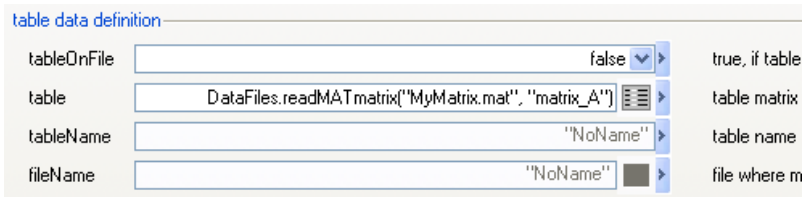
The package `DataFiles.mo`, located in `Program Files\Dymola 2023\Modelica\Library` contains functions for reading and writing data, and a block TableND.

The DataFiles package is available using the command **File > Libraries > DataFiles**.

Functions in the package

Using the functions, Matlab 4, CSV and text files can be handled. The data in CSV or txt files can be separated by tab, space, semicolon or comma. Please see the documentation of the functions in the package for more details.

The following figure (the parameter “table”) is an example where a function from the DataFiles package is used for reading a .mat file (instead of going into the matrix editor and then selecting **Import...**). The matrix editor is not accessed at all.



The block TableND

The block `TableND` is used for linear interpolation in N dimensions. The block can read and write files in Matlab 4 and text format. Please see the documentation in the block for more details. Please also see the package Modelica.Blocks.Tables (where e.g. the specific text format is described more in detail).

SDF Editor

The SDF Editor is a standalone application to create, view, and edit files in the Scientific Data Format.

It is available from Windows Start menu, or by opening the file

```
Program Files\Dymola 2023\bin64\SDF_Editor.exe
```

For more information, see the documentation of the application.

Note! Due to Microsoft already having reserved the .sdf extension for SQL Server Compact Edition Database File, the automatic association for .sdf files with the SDF Editor is not done anymore, from Dymola 2018. You have to do this association manually if you want, for example, that the file should be opened in the SDF Editor when you double-click the file in the browser. To do the association, right-click the file in a file browser, and select **Properties**. In the General tab, click **Change** and select `SDF_Editor.exe` if available. If not available, browse to …`Program Files\Dymola 2023\bin64\SDF_Editor.exe` and select that one.

If you have used Dymola versions older than Dymola 2018, .sdf files are automatically associated with the SDF Editor. If you get problems because you are to use SQL Server Compact Edition Database files, you must change the file association manually (like above) to associate them with the SQL software.

Setting inputs by data from files

Top level inputs of a model or block can be set by the file `dsu.txt`. If this file exists, the trajectories of input signals are read from this file. For more information about this file, see next chapter, section "Dynamic Model Simulator", sub-section "Running Dymosim"; "Dymosim as a stand-alone program".

Note that top level inputs of a model or block only can use literal start-values given in the model. If an input has a non-literal value, the value zero is used instead, and diagnostics is given.

Using alist.exe for file, variable, and matrix conversion

In the folder `Program Files\Dymola 2023\bin` there is a program `alist.exe` that can be used to

- Read ASCII or binary files (e.g. Matlab files) and store them in ASCII or Matlab format.
- Export variables in CSV format.
- Convert single precision matrices to double precision.
- Convert double precision matrices to single precision.

For more information, use the help command `alist -h` in the folder above. Note that you can store signals in CSV format with double precision; that is, you can use the `-e` option together with the `-d` option.

4.3.18 Display units

The use of display units

It is strongly recommended that the developer uses unscaled SI units when specifying a unit attribute of a real variable. This is a condition for the unit checking (please see section "Unit checking and unit deduction in Dymola" on page 437). However, the user often wants to input and plot those variables in other units. To meet that demand, an attribute displayUnit is available.

displayUnit is used when entering a value in e.g. the parameter dialog (please see section "General tab" on page 239) or selecting the display unit of a curve in a plot window (please see next chapter, section "Model simulation", sub-section "Plot window interaction" – "Changing the displayed unit of signals").

Some display units have several selectable alternatives. How this can be seen and how to select between the alternatives are described in the references above.

Adding new unit alternatives and setting default display units

Three files, located in the folder `Program Files\Dymola 2023\insert` deals with unit conversion. The set of common unit conversions which are used by default can be found in `displayunit.mos`. Additional US unit conversions are defined in `displayunit_us.mos`; which are used by default can be changed in `dymola.mos`. (Changing any of these files might require administrator rights.)

To make it easier to use display units suitable for a given application, you can now control which setup scripts are executed, by setting a variable in Dymola. For example, to run the two display unit scripts mentioned above:

```
Advanced.DisplayunitSetupPath= "$DYMOLA/insert/displayunit.mos;
$DYMOLA/insert/displayunit_us.mos"
```

The user can add new alternatives for a displayUnit by modifying the file `displayunit.mos` (or `displayunit_us.mos`, depending on which one should be used). The following is an example of how the first part and the last part of the file can look like. The first part defines possible unit conversions; the last part defines default display units.

```
// Unit conversions in Dymola
//
// ----------------------------------------------------------------------
// Possible unit conversions are defined below. They are used for
// selecting display unit in Plot/Setup.
//
// Syntax:
//
// defineUnitConversion(<unit>, <derived unit>, <scale>, <opt. offset>);
```

```
// Time
defineUnitConversion("s", "ms", 1000);
defineUnitConversion("s", "min", 1/60);
defineUnitConversion("s", "h", 1/3600);
defineUnitConversion("s", "d", 1/86400);

// Angle
defineUnitConversion("rad", "deg", 180/3.14159265358979323846);

....

// Density
defineUnitConversion("kg/m3", "kg/dm3", 1e-3);
defineUnitConversion("kg/m3", "kg/l", 1e-3);
// defineUnitConversion("kg/m3", "g/cm3", 1e-3);
// defineUnitConversion("kg/s", "g/s", 1e3);

....

// For Rectification and Chemical engineering
defineUnitConversion("mol/s", "mol/h", 3600);
defineUnitConversion("J/(mol.K)", "J/(kmol.K)", 1000);
defineUnitConversion("s/m", "h/m", 1/3600);
defineUnitConversion("W/(m2.K)", "kW/(m2.K)", 1e-3);
defineUnitConversion("m2/s", "m2/h", 3600);

// Surface-area-to-volume ratio (for reactivity in chemical reactions)
defineUnitConversion("m2/m3", "1/m", 1);

// various
defineUnitConversion("1", "%", 100);

// ------------------------------------------------------------------
// The default display unit used by Dymola (until another display unit
// has been specified in Plot/Setup) can be defined.
//
// Syntax:
//
// defineDefaultDisplayUnit(<unit>, <derived unit>);

// defineDefaultDisplayUnit("Pa", "bar");
// defineDefaultDisplayUnit("m3/s", "l/min");
// defineDefaultDisplayUnit("K", "degC");
// defineDefaultDisplayUnit("m/s", "km/h");
```

The first part of the file, possible unit conversions, defines what alternatives are available for a certain unit, e.g. time. When presenting “time” in a parameter dialog or plot window the user can select between **ms**, **s**, **min**, **h**, and **d**. If the user wants to also be able to also have **y** (year) selectable, the following has to be added in the “Time” group:

```
defineUnitConversion("s", "y", 1/31536000);
```

Note that some possible unit conversions are prepared but not activated, as example see the “Density” part above – you can remove the // in front of those lines to have these unit

conversions active as well. You can also comment away unit conversions by adding // in front of them.

The last part of the file, default display units, defines what the default unit of presentation should be. (When the user has selected anything else, Dymola will remember that.) If **y** (year) should be the default presentation format for time, the following line should be added as the last line:

```
defineDefaultDisplayUnit("s", "y");
```

The file has to be saved and Dymola has to be restarted to implement these changes. The file displayunit.mos is read from the file dymola.mos that in turn is executed each time Dymola starts.

Absolute and relative units

For the special case of temperatures (in general – any type with an offset in the conversion) a new annotation on the type is needed to differentiate between absolute temperature and temperature differences (default is absolute):

```
annotation(__Dymola_absoluteValue=true)
```

Defining reciprocal display units

Reciprocal unit conversions can be handled through display units, such as the relationship between l/100km and mpg (miles per gallon). This feature can be used to handle other reciprocal conversions like, for example, between frequency and period. The feature is in particular useful for library developers.

Looking at the Fuel consumption items in the displayunit.mos extract above, we find the following part:

```
defineUnitConversion("cl/km", "l/100km", 1, 0);
defineUnitConversion("cl/km", "mpg", 0, 454.609/1.609344);
```

Setting the input argument `scale` to `0` in the `defineUnitConversion` function call, defines the following argument `offset` to be used not as an offset, but instead as scale for an reciprocal conversion.

(Concerning the mpg conversion, the related variable should be given in the unit "`cl/km`" (centiliter/kilometer). The reason is that we do not want existing quantities with the SI unit m^2 to acquire surprising display units.)

Support for prefixed display units

The support for prefixed display units include:

- The scale-factor is checked for correctness.
- Prefixed units are automatically handled in the parameter dialog (e.g. when typing **2 mV** in a parameter input field), or if found in models.
- The base-unit is checked (to prevent using e.g. **mV** in an input field with base unit **A**).

Note 1. It is possible to define prefixed display unit without entering the definition of them in any file, by entering value with prefixed display units in the parameter dialog. See “Editing parameters and variables” starting on page 237, item “E” in the description of the parameter dialog. They are also available for corresponding curves when simulating the model.

Note 2. The diagnostics for bad units is not yet fully complete.

Setting imperial units as default units for plots

It is possible to set imperial units as the default for plot generation, by adding a third argument in the `defineDefaultDisplayUnit` definitions in the file `displayUnit_us`, like the following:

```
defineDefaultDisplayUnit("Pa", "psi", true);
defineDefaultDisplayUnit("m3/s", "gal/min", true);
defineDefaultDisplayUnit("K", "degF", true);
defineDefaultDisplayUnit("m/s", "mph", true);
```

This third argument `true` makes the setting override the units set in the model, and thus e.g. graphs will as defaults use these units.

4.3.19 Working with annotations using the Annotations dialog

You can use the **Annotations** dialog to display and edit top-level annotations of a model, as well as any annotation that is related to graphics, for example, annotations of graphical objects, components and connectors.

When it comes to annotations for items not related to graphics, for example, parameters and variables not represented by graphical objects, you cannot use the annotation dialog, but you can edit the annotations in the Modelica Text layer, or, when it comes to parameters and variables, you can use the variable declaration dialog.

All annotations can be edited in the Modelica Text layer.

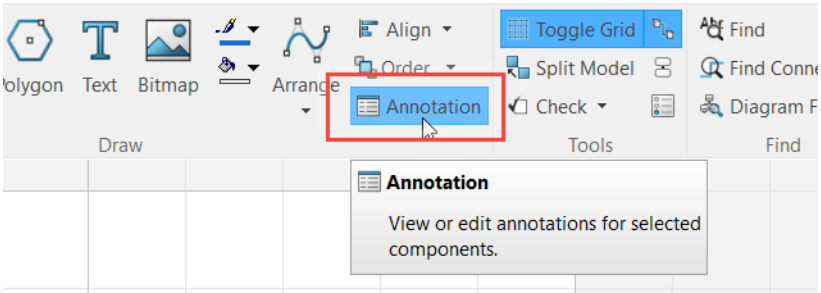
Working with model annotations

Model annotations are top-level annotations of a model, including annotations for graphical items at top-level.

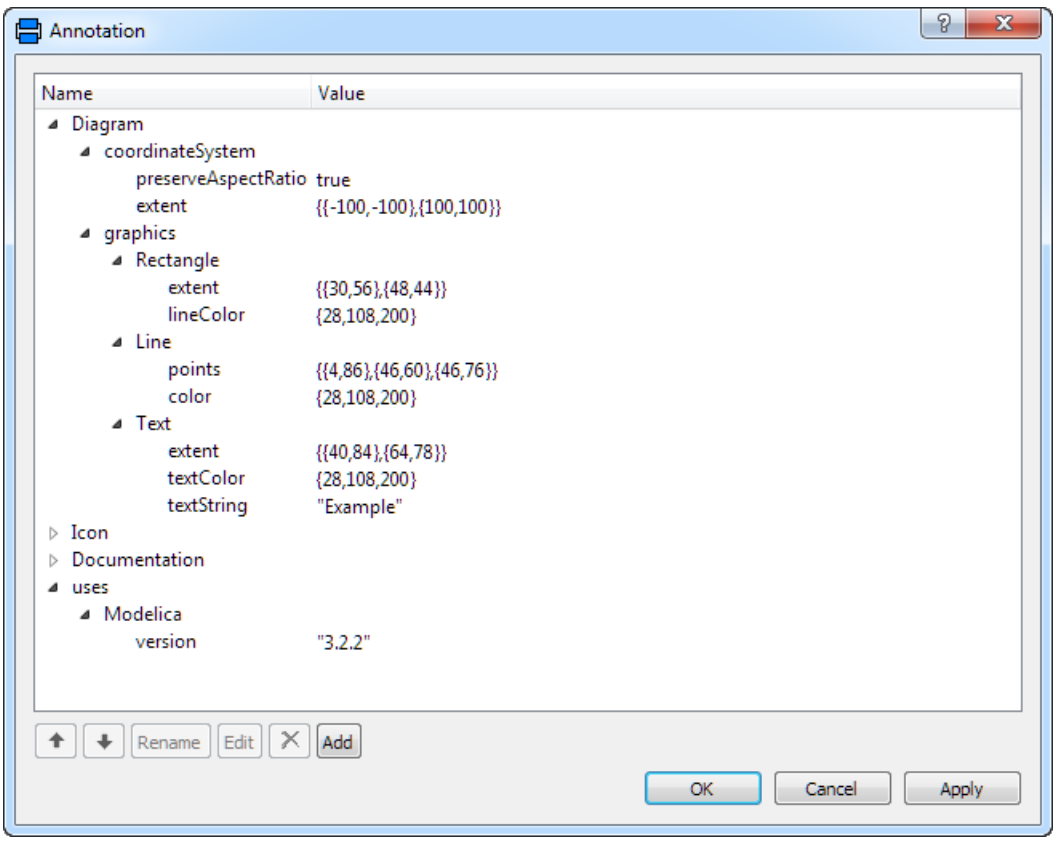
To open the annotations dialog for model annotations, do the following:

- In the diagram layer, (in the **Graphics** tab) make sure you have nothing selected – you can click outside any object to make sure it is so.
- In the **Draw** group, click **Annotation**.

- [Diagram]



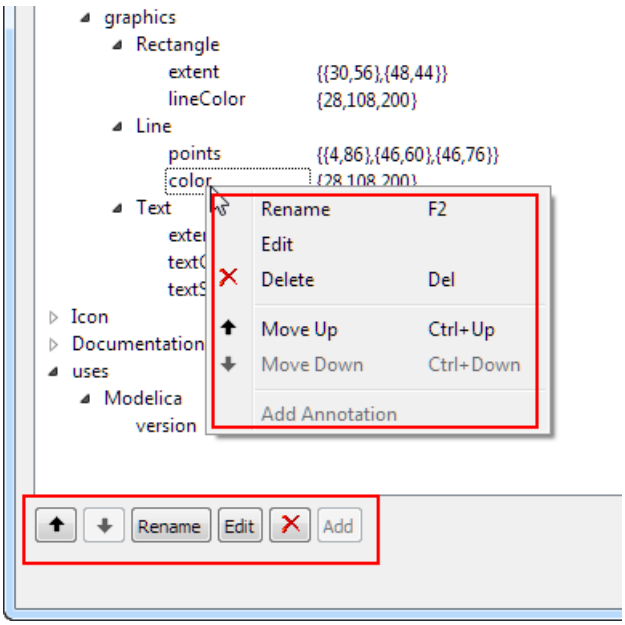
An example of a resulting dialog:



The following apply:

- All model annotations are shown.

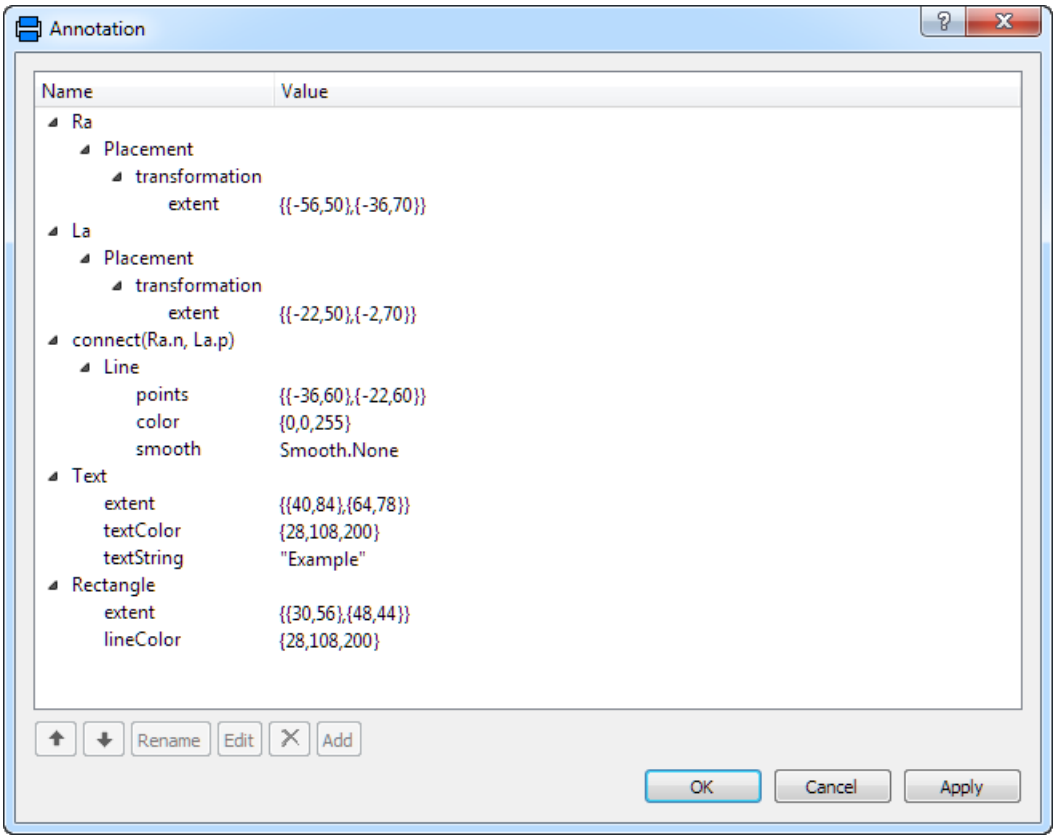
- The part corresponding to the active layer when opening the dialog is shown expanded (in this example Diagram), annotations for the other graphical layers (in this example Icon and Documentation) are collapsed.
- You can create, edit, rename, change order (up/down), and delete annotations, including top-level ones, using either a context menu or the toolbar buttons in the bottom of the dialog.



For a detailed description on how to edit annotations, see section “Details in working with annotations using the Annotation dialog” on page 430 below.

Working with component annotations

To work with component annotations, select or multi-select components in the diagram, and then, in the **Draw** group, click **Annotation**. An example of such a dialog from a multi-selection of connected components and some graphical objects:



The following apply:

- All annotations for selected components, connections, and graphical primitives are shown.
- The root for connection annotations is the connection string; the root for component annotation is the component name. These root names are not editable.
- You can create, edit, rename, change order (up/down), and delete annotations, like for model annotations (see above). For a detailed description on how to edit annotations, see next section. There are some exceptions when editing, in this dialog you cannot:
    - Change the root names of connection annotations and component annotations.
    - Create top-level annotations.
    - Change the order of the top-level annotations. However, you can change the order of children of such annotations.

Details in working with annotations using the Annotation dialog

Adding top-level annotations

You can add top-level annotations when you work with model annotations. In that dialog, click below the present annotations (you might have to make the window longer) and then click the **Add** toolbar button (or right-click and select **Add Annotation**).

You can now enter a name for the new node. An error message will appear when you click **Apply** or **OK** if you try to apply a name that is not legal Modelica.

Nodes that are added on the top level only have a name. Nodes added further down also have values.

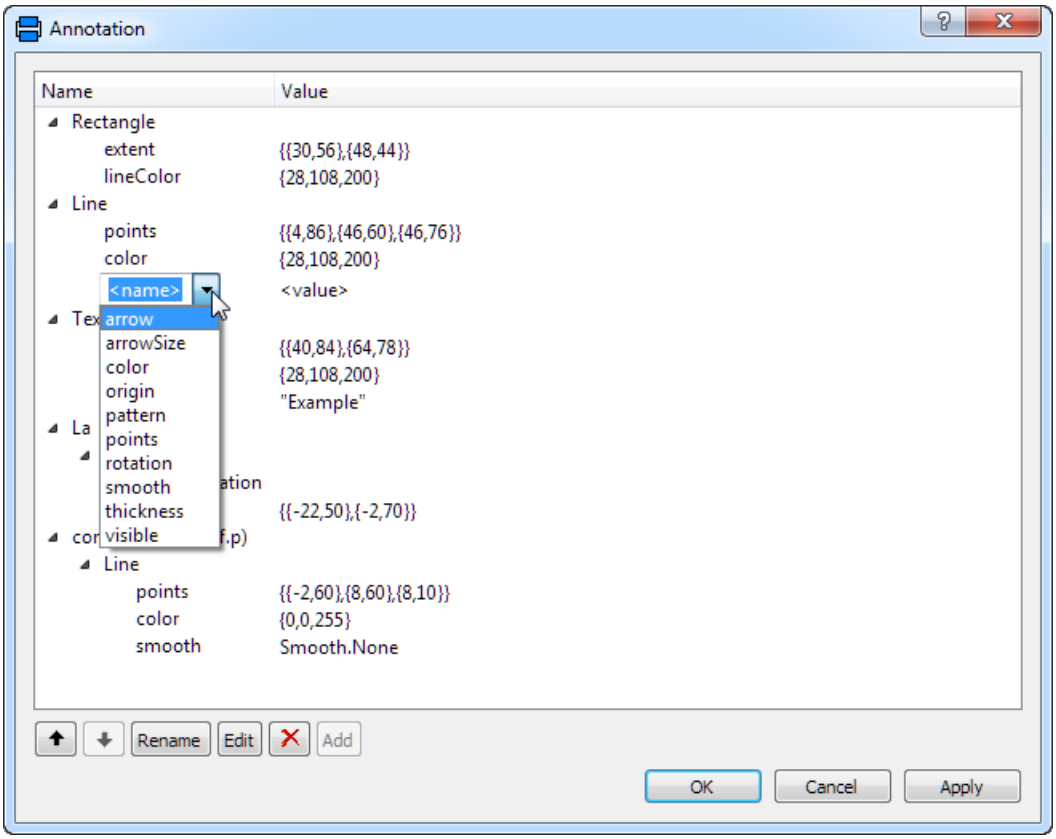
Adding children to an annotation

To add children to an annotation without value, select the annotation and click **Add** (or right-click and select **Add Annotation**).

To be able to add children to an annotation with a value, first remove the value.

The first thing to do after adding an annotation (child) is to rename the new annotation.

If you add an annotation to a node that is a graphical primitive, component, or connection, you will have a dropdown list to select from when you are to specify the name of the annotation:



Note. The connection `Text` has different attributes than the Text primitive.

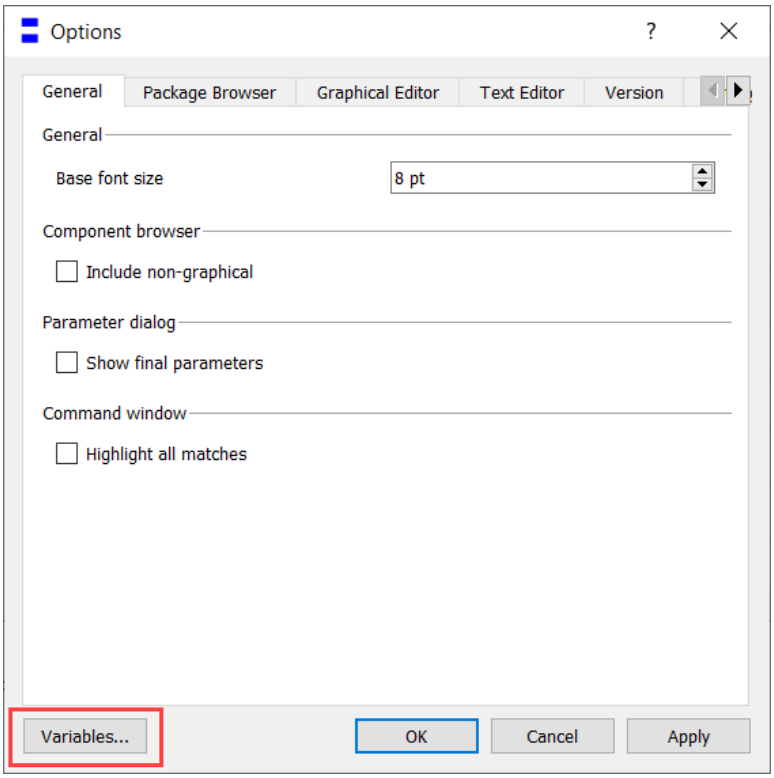
Annotations not being top-level ones must have values. You get `<value>` when creating the annotation; you must yourself change this to a proper value. Presently there are no dropdown lists for the values, only text boxes.

4.3.20 General GUI for displaying and setting flags

General

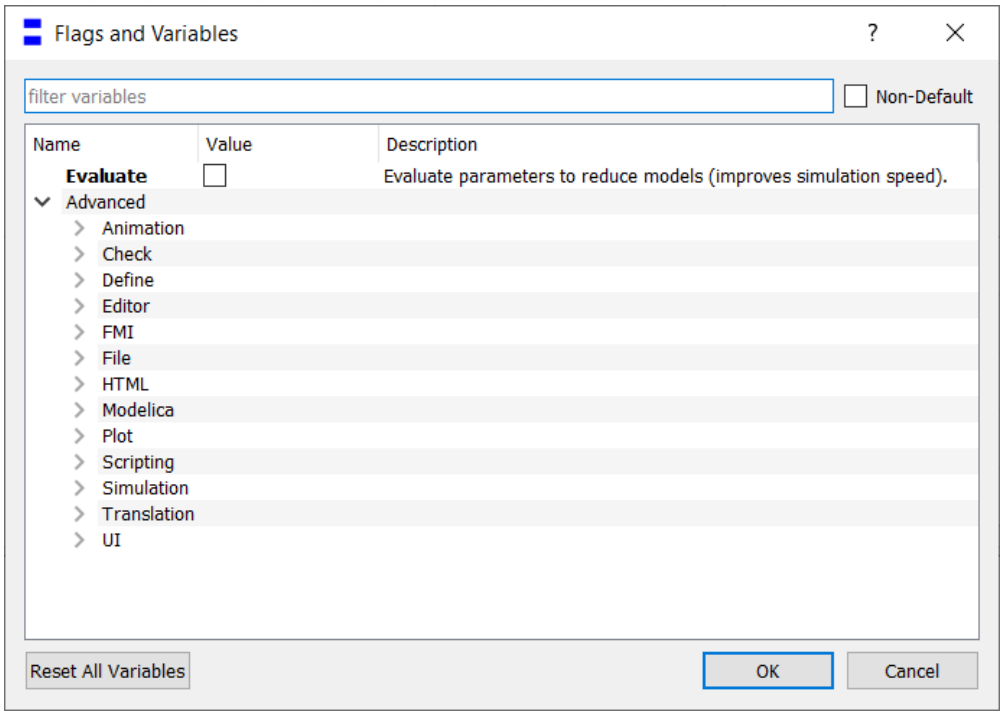
In most cases, the setting of flags corresponds to options available in, for example, the different tabs of the **Tools > Options** command. However, in some cases there is a need to display and/or change the value of flags not covered by other GUI options.

To display the dialog, perform the command **Tools > Options** and click the button **Variables**. (The button is available in all tabs.)



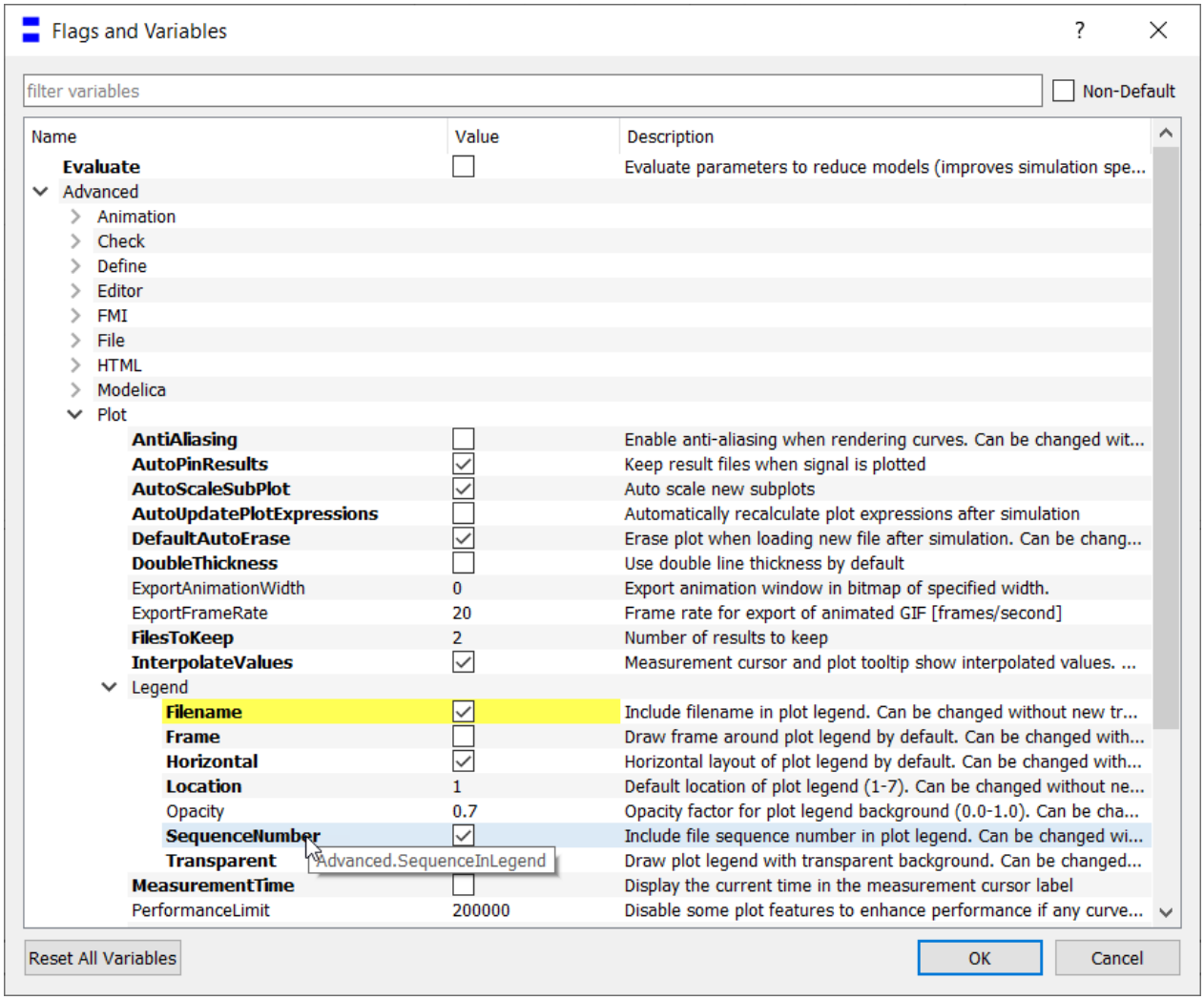
Note. When clicking the button, you also apply any change you have done in any tab in the **Tools > Options** dialog before clicking the button.

Clicking the button displays:



The flags have been grouped and renamed in this window to make it easier to explore related flags. Groups can be collapsed or expanded by clicking the symbol in front of it.

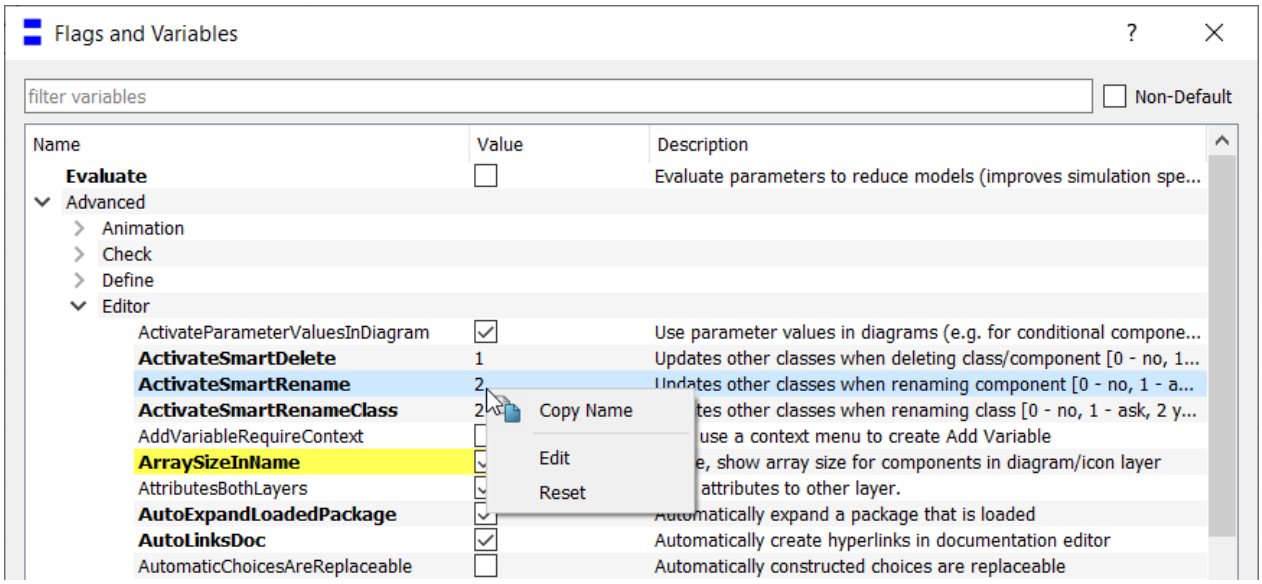
A tooltip is available for displaying the full present name of the flag or the full description, depending on what part you pause the cursor on. Below pausing over the name:



Important! The restructuring of the flags is presently only implemented in this window – the tooltip displays the present real name of the flag, which is the same as in previous versions.

Bold flag name indicates that the flag is stored between sessions, yellow background that the value of the flag is not the default value.

A toolip is available by right-clicking any flag:



You can change the value of a flag the following way, depending on what type of flag:

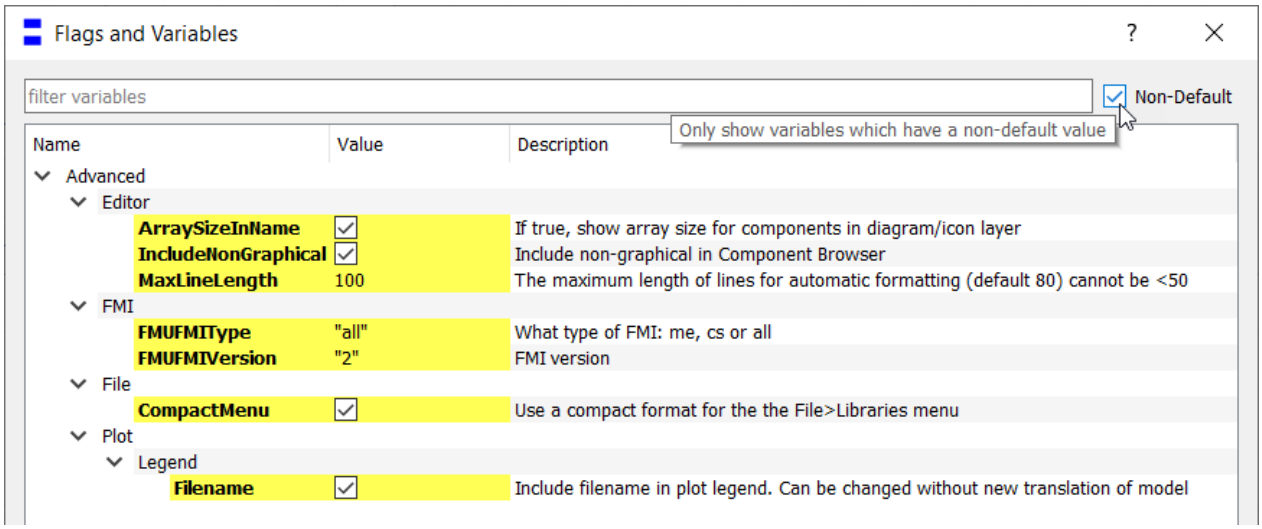
- For Boolean flags, click the checkbox in the Value column to change value, or right-click it and select **Edit**. Any of these actions toggle the flag.
- For a non-Boolean flag, click the value twice to open an editor to change the value (note that double-clicking has no effect), or right-click the flag and select **Edit** to open the editor.

To reset the value of the flag to its default value, right-click it and select **Reset**.

To copy the name of a flag to the clipboard, right-click the name of the flag and select **Copy Name**. Important! The real name of the flag is copied (the name displayed by the tooltip).

Displaying flags with other values than the default ones

As mentioned, flags having other values than the default values are marked yellow in the dialog. You can display only such flags by ticking **Non-Default** after the filter box in the upper part of the dialog. An example of ticking this box:

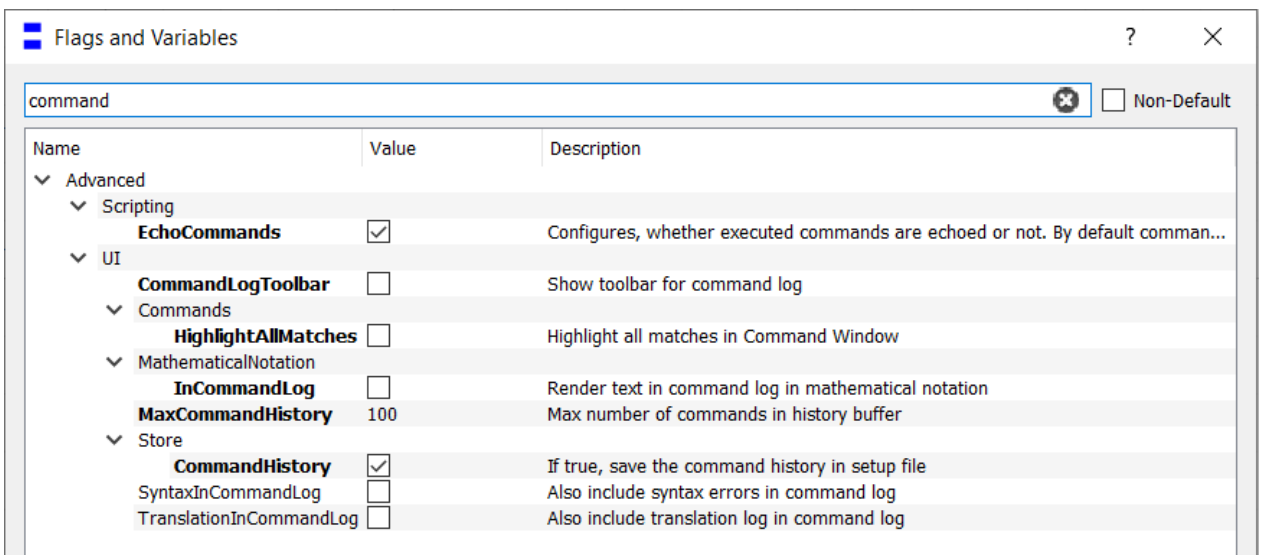


Resetting all flags that can be displayed in this dialog to the default values

You can reset all flags that can be displayed in this dialog to the default values by clicking **Reset All Variables** in the lower left of the dialog.

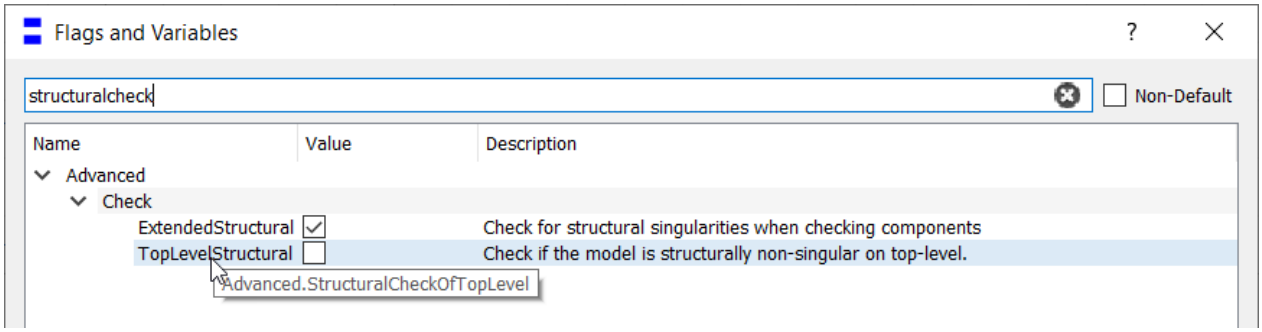
Filtering flags

You can filter flags by the filter box in the top of the dialog. An example:



The filtering is dynamic and is updated when typing. The filter searches for the character typed inside any flag name in the dialog (the description strings are not searched).

Important! The filter searches for both the name of the flag used in this dialog and the real present name of the flag. An example where the filter matches only the present real names:



4.4 Checking the model

4.4.1 Commands

Please see section "Text > Check" starting on page 491 for commands and also for some general comments.

4.4.2 Unit checking and unit deduction in Dymola

Introduction

When modeling a physical system, a variable typically corresponds to a physical quantity. The value of a quantity is generally expressed as the product of a number and unit. When declaring a real variable, it is possible to specify its unit of measure. The package Modelica.Units.SI provides a large set of predefined quantities and it is recommended to use them whenever possible. See the Users Guide of package Modelica.Units.SI.

Dymola has a feature for checking units. It is active when checking a package, function or model as well as when translating a model for simulation. It includes checking of unit strings and unit compatibility of equations. There is also a feature for deducing units automatically.

Supported units

The Modelica specification states "A basic support of units in Modelica should know the basic and derived units of the SI system." Dymola fulfils this requirement.

The SI units were invented to allow equations to be written in a clean way without conversion factors. Thus, it is recommended that unscaled SI units are used when specifying the unit attribute of a real variable. To be clear, this also means that prefixes shall not be used. For example, "m", "kg", "V", "N.m" or "W" is good, but not "cm", "g", "kV", "MW" or "bar". Use the quantities defined by Modelica.Units.SI whenever possible.

Dymola recognizes the users' needs to enter parameters and plot variables in different units. Modelica defines displayUnit for that purpose. Dymola supports displayUnit when plotting variables and when entering values in parameter dialogs. The parameter value is stored in SI units. Thus portability is preserved and it is a tool issue to support the displayUnit in the dialogs.

A good reference on SI units is what commonly is called the SI brochure published by Bureau International des Poids et Mesures [BIPM, 2006]. The NIST Reference on Constants, Units, and Uncertainty [NIST, 2000] gives a good overview; see also [Taylor, 1995].

ISO does not specify a formal syntax for unit expressions but there are recommendations, which are strict. The Modelica language specification includes a formal specification based on the recommendations.

Dymola supports all the 20 SI prefixes to form decimal multiples and submultiples of SI units.

SI prefixes.

Factor	Name	Symbol	Factor	Name	Symbol
10^{1}	deca	da	10^{-1}	deci	d
10^{2}	hecto	h	10^{-2}	centi	c
10^{3}	kilo	k	10^{-3}	milli	m
10^{6}	mega	M	10^{-6}	micro	μ
10^{9}	giga	G	10^{-9}	nano	n
10^{12}	tera	T	10^{-12}	pico	p
10^{15}	peta	P	10^{-15}	femto	f
10^{18}	exa	E	10^{-18}	atto	a
10^{21}	zetta	Z	10^{-21}	zepto	z
10^{24}	yotta	Y	10^{-24}	yocto	y

Dymola knows all the seven SI base units

SI base units.

Name	Symbol
metre	m
kilogram	kg
second	s
ampere	A
kelvin	K
mole	mol
candela	cd

as well as the 22 SI derived units that have been given special names and symbols

SI derived units.

Name	Symbol (in Modelica)	Definition
radian	rad	1
steradian	sr	1
hertz	Hz	1/s
newton	N	$kg.m/s^2$
pascal	Pa	N/m^2
joule	J	N.m
watt	W	J/s
coloumb	C	A.s
volt	V	W/A
farad	F	C/V
ohm	Ohm	V/A
siemens	S	A/V
weber	Wb	V.s
tesla	T	Wb/m^2
henry	H	Wb/A
degree Celcius	degC	K
lumen	lm	cd.sr
lux	lx	lm/m^2
becquerel	Bq	1/s
gray	Gy	J/kg
sievert	Sv	J/kg
katal	kat	mol/s

There are also units that are not part of the International System of Units, that is, they are outside the SI, but they are accepted for use with the SI. Dymola knows the following of these accepted non-SI units:

Non-SI units.

Name	Symbol	Expressed in SI units
minute	min	60 s
hour	h	60 min
day	d	24 h
degree	deg	$(\pi/180)$ rad

litre	l	dm^3
decibel	dB	1
electronvolt	eV	0.160218 aJ
bar	bar	0.1 MPa
phon	phon	1
sone	sone	1

In power systems the unit for apparent power is “V.A”. Dymola knows

```
var = V.A
```

which has been adopted by the International Electrotechnical Commission, IEC, as the coherent SI unit volt ampere for reactive power, see IEC [2007].

The rotational frequency *n* of a rotating body is defined to be the number of revolutions it makes in a time interval divided by that time interval. The SI unit of this quantity is thus the reciprocal second (s^{-1}). However, the designations "revolutions per second" (r/s) and "revolutions per minute" (r/min) are widely used as units for rotational frequency in specifications on rotating machinery. Although use of rpm as an abbreviation is common, its use as a symbol is discouraged. Dymola knows

```
r = 2π rad
```

It can be used for example as “r/s” or “r/min”.

Dymola also knows the temperature units degF (degree Fahrenheit) and degRk (degree Rankin).

Unit checking

Dymola performs checking of units. It is active when checking a package, function or model as well as when translating a model for simulation. It includes checking of unit strings and unit compatibility of equations. It can be seen as a part of the type checking. It includes the checking of actual function input arguments and output arguments against their formal declarations.

Currently Dymola makes a relaxed checking. It means that an empty unit string, "", is interpreted as unknown unit. Also number literals are interpreted to have unknown unit. The unknown unit is propagated according to simple rules

```
unknown unit * "unit1" -> unknown unit
unknown unit + "unit1" -> "unit1"
```

There is one important exception. Let e be a scalar real expression. Consider the inverse of e given as 1/e. The number 1 (one) in the numerator does not relax the checking. If e has a well-defined unit then also 1/e has a well-defined unit.

The unit checking is applied to the original equations. This has implications for vector, matrix and array equations. For an array where all elements have the same unit, the check works as if it was a scalar. Arrays and array expressions where the elements have different units are

allowed. However, the check is then relaxed and the array is viewed to have an unknown unit that is compatible with all units. Checking the unit consistency between two records is done recursively for each component.

Currently, the unit checking does not issue error messages but it generates only warnings. The unit checking can be disabled setting the flag

```
Advanced.CheckUnits = false
```

As an example of unit checking consider the following incorrect model. Here velocity has been used instead of acceleration in Newton's second law.

```
model Newton
  parameter Modelica.Units.SI.Mass m = 10 "Mass";
  Modelica.Units.SI.Force f = 10 "Force";
  Modelica.Units.SI.Velocity v "Velocity";
equation
  m*v = f;
end Newton;
```

Running variable unit checking on this model generates the following log:

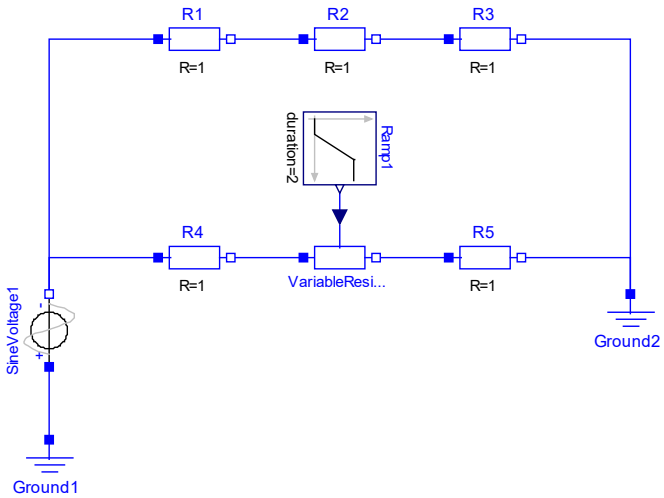
```
Warning: Incompatible units in
  equation
  m*v = f;
The part
  m*v
has unit m.kg.s-1
The part
  f
has unit m.kg.s-2
in equation
  m*v = f;
```

A warning was issued, indicating that the left- and right hand side of the equation has different units. Correcting the equation (using acceleration instead of velocity) removes the warning.

Unit deduction

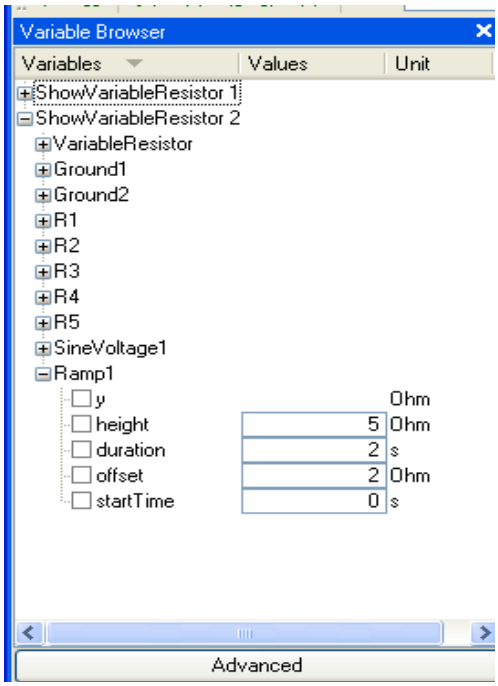
The Modelica.Blocks library includes general blocks to define sources and mathematical operations. Their inputs and output have of course no units specified. For user convenience, Dymola has introduced automatic deduction of units. Consider the example Modelica.Electrical.Analog.Examples.ShowVariableResistor.

Example.



The block Ramp1 is used to specify a resistance. Neither its parameters height and offset nor its output y has units specified. However, these units are deduced at translation and displayed in the variable browser.

Displaying units in the variable browser.



The unit deduction is activated by default but can be de-activated by setting

```
Advanced.DeduceUnits = false
```

The unit deduction requires that the unit check is active, i.e., `Advanced.CheckUnits = true` (default setting).

The result can be logged by activating the option **Log deduced units**. To access this settings, on the **Simulation** tab, in the **Simulation** group, click **Setup**, and, in the resulting simulation setup dialog, go to the **Translation** tab. (Please see the corresponding section in next chapter for more information.) This option corresponds to the flag

```
Advanced.LogDeducedUnits = true.
```

The deduction of units may reveal unit inconsistencies. In such case it may be useful to enable the logging and inspect the log. It is also useful to check the log when developing a model component, because if a real variable gets its unit deduced that may indicate that the variables shall be declared using any of quantities defined by Modelica.Units.SI.

Here is a short description of how it works. Consider the expression

```
e1 + e2
```

where Dymola has found that the expression e1 has a well-defined unit u1, but the unit of the expression e2 is unknown. We can then deduce as described in the introduction that the unit of the sum e1 + e2 is u1. Moreover, for unit consistency reasons the unit of e2 must also be u1. If now e2 is a simple variable reference, v, we can deduce that v must have the unit u1. We have thus deduced the unit of v. For more complex expressions Dymola makes a downwards recursion to see if it is possible to deduce units for some variable with unknown units.

References

[BIPM, 2006] The International System of Units (SI), Bureau International des Poids et Mesures, 8th edition, 2006. Available in electronic form at www.bipm.org/en/si/si_brochure

[IEC, 2007] SI Zone, International System of Units, SI units – Electricity and magnetism, http://www.iec.ch/zone/si/si_elecmag.htm

[NIST, 2000] The NIST Reference on Constants, Units, and Uncertainty, - International System of Units (SI), http://physics.nist.gov/cuu/Units/

[Taylor 1995] B.N. Taylor: "Guide for the Use of the International System of Units (SI)", NIST Special Publication 811, United States Department of Commerce, National Institute of Standards and Technology, USA, 1995. Available in electronic form at http://physics.nist.gov/cuu/pdf/sp811.pdf

4.5 Editing model reference

This section presents a number of subjects aimed at library developers.

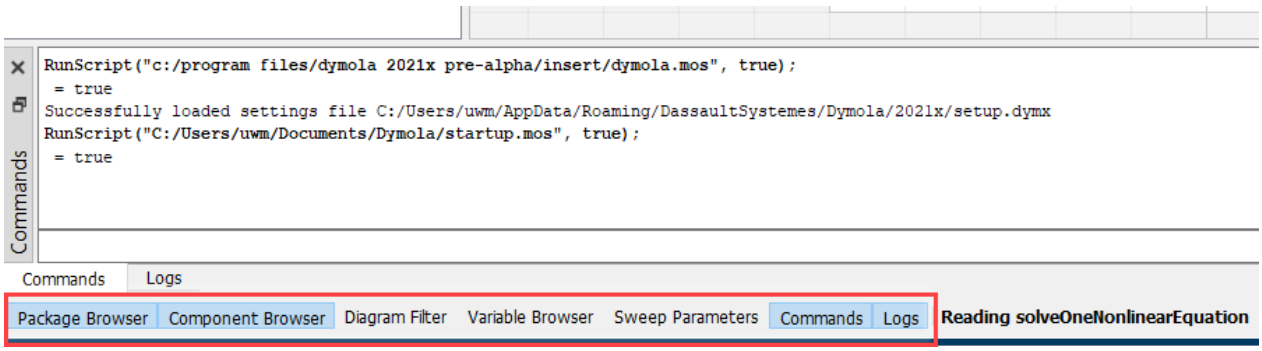
The first sub-section in this section presents some window settings that can be done, both when it comes to what windows should be shown and the settings in the windows.

4.5.1 Window settings

General Dymola settings

Window content – Dymola Main window

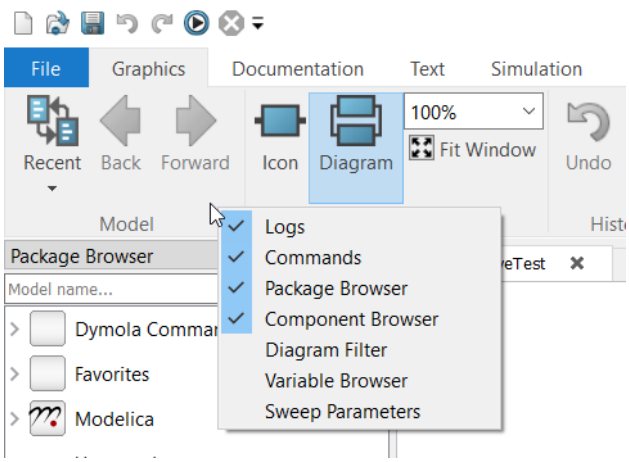
The content of the Dymola main window depends on which tab you are in, but in any tab you can also decide what browsers and other windows to display. The selection is done by clicking the buttons in the bottom of the Dymola window:



You can click on the buttons to display/hide the browsers/windows. The buttons work in toggle mode.

The settings are individual for each tab. The settings are saved between sessions.

These settings can also be displayed by clicking the gray area in the ribbon tab (or the header of for example the package browser). An example for the **Graphics** tab:



Using the command **Tools > Options…**, the options in the **General** tab also influences what is shown in Dymola Main window. Please see section “General options tab” on page 511.

Window content - other windows

Depending what is shown, sometimes the context menu of a window can be used to change the content of the window.

Font size etc.

Apart from the settings that can be made in Microsoft Windows, it is possible to e.g. change the font size in Dymola using the command **Tools > Options…**, the **General** tab. Please see section “General options tab” on page 511.

Model editor initialization

Dymola can automatically recognize different libraries in order to e.g. build the **File > Libraries** and **File > Demos** menus. It is very easy to add new libraries and to add new versions of existing libraries. Please see the chapter “Appendix – Installation”, section “Installation on Windows”, sub-section “Additional setup” for more information.

4.5.2 Default view of classes

Selecting what layer should be presented in edit window when a class is selected

Dymola supports an annotation to control what layer to show when a class is selected. As an example, the following annotation will always show the Modelica Text layer of the class when the class is selected:

```
annotation(preferredView="text")
```

“diagram”, “text”, “icon” and “info” are the possible selections; “info” will show the documentation layer.

Creation of information class



This symbol symbolizes a package containing only information. Selecting such a package will always show the documentation layer and dragging of such package is inhibited. It is possible to create such a class by using the annotation:

```
annotation(__Dymola_DocumentationClass=true)
```

This annotation will create the icon and the description string.

4.5.3 Package browser details

Controlling the clicking behavior of the package browser

There is a switch to control browser behavior: Advanced.SingleClickOpensTree 0=no open on single click, 1=toggle open, 2=only open; default=2.

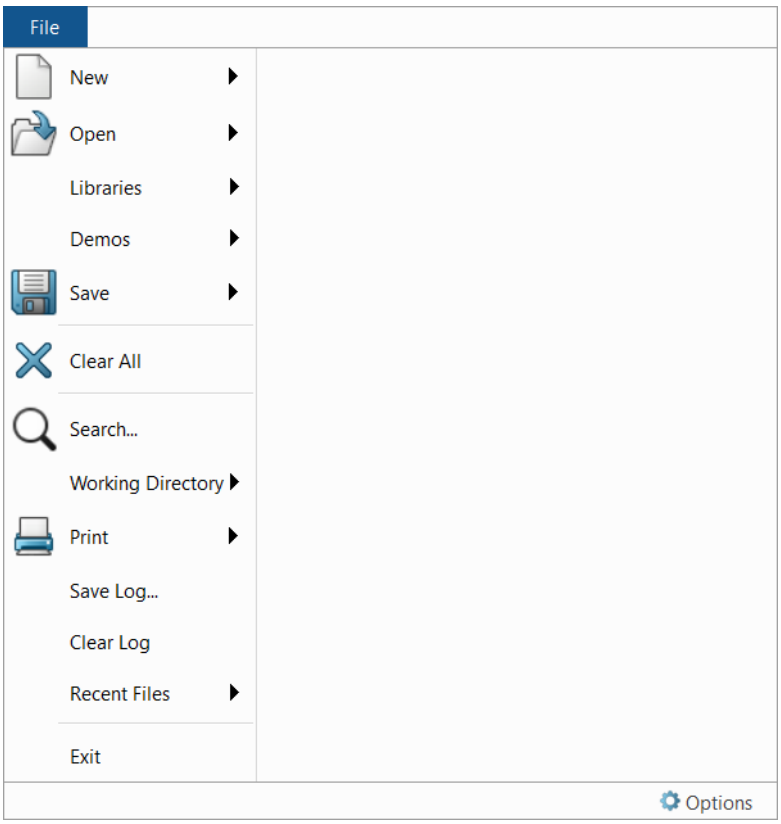
4.6 Editor command reference when developing a model

The first sub-sections in this section cover the menu entries in Dymola Main window related to developing a model (the **File** menu, and the **Graphics**, **Text**, and **Tools** tabs). The **Simulation** tab and the related **Plot**, **Animation**, and **Script Editor** tabs are to be found in the next chapter since they deal with simulating a model.

One sub-section lists some keyboard commands.

The context menus are presented in a number of sub-sections. Graphical object menus (line and fill style) are presented in connection to the context menu for graphical objects.

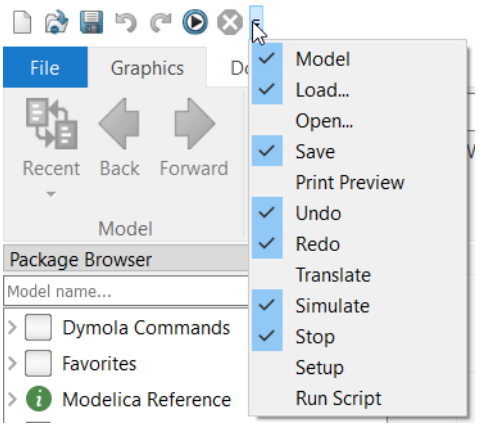
4.6.1 Main window: File menu



Shortcuts to File commands in the top left toolbar



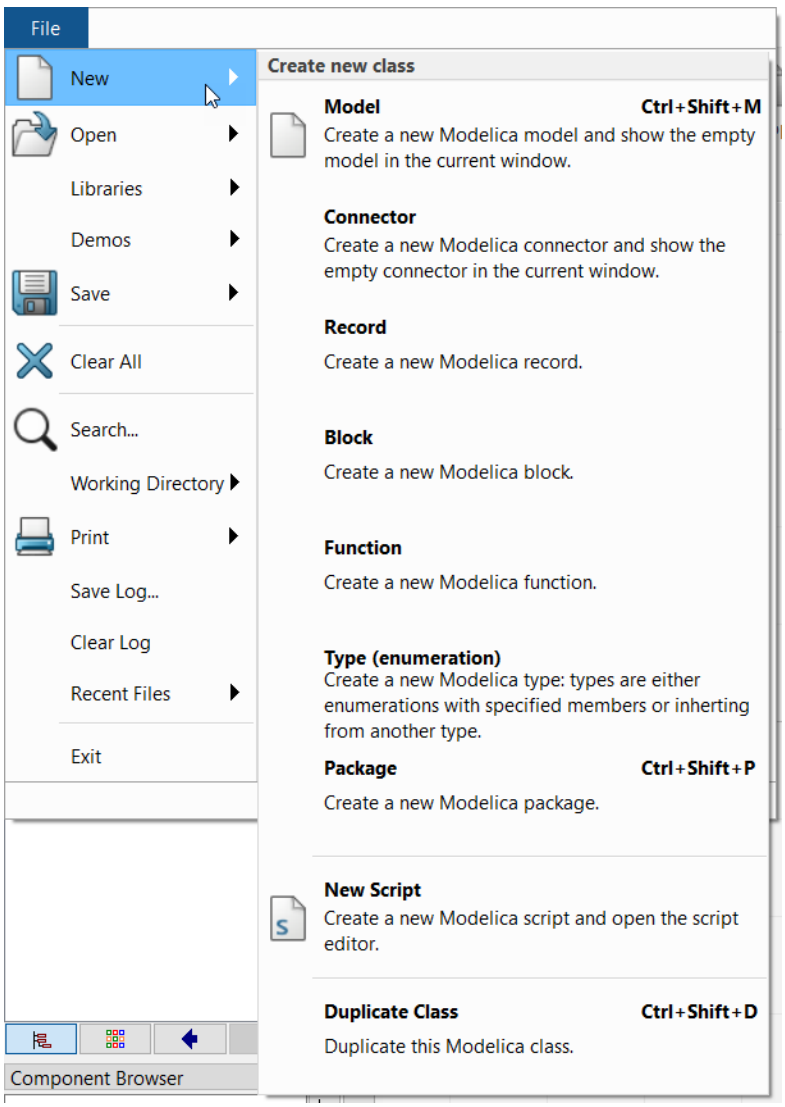
The top left toolbar of the Dymola window by default contains buttons for creating a new model, loading a model, and saving a model. Using the arrow to the right you can configure what commands to display in the top left toolbar, for example, also add the print preview command (as done in the image above):



File > New

The **File > New** command can be used to create a number of entities.

The File > New command.



Below the alternatives are treated in a number of sections, with the same header as the command name. The first commands, used to create any of the items model, connector, record, block or function are treated in the first section below, since they are quite similar.

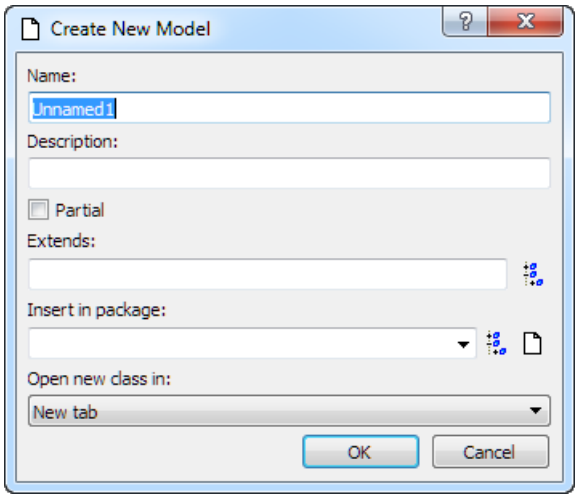
File > New > Model etc.

Creates a new model, connector, record, block or function.

(For all alternatives of the **File > New** command, see the section "The File > New… command alternatives" above.)

Creating a new model.

File > New > Model gives:



(Note that by default the button  in the top left toolbar of the Dymola window is a shortcut for this command.)

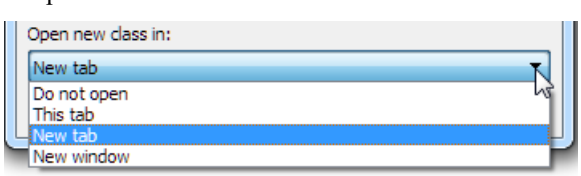
Dymola presents a dialog window where the user can enter the name of the new model, a short description, and if this is a partial model. The name must be a legal Modelica identifier and not clash with existing models. The default name is `Unnamed`. If a model already exists with this name, the default name will be `Unnamed1` etc.

Drag from the package browser to extend or insert in a package

The **Extends:** field is the name of the base class (if the class should extend from an existing model). It is possible to drag a class name from the package browser into this field. You can use **Select Base Class** to browse for base classes to extend. For more about extending, see "Creating a model by extending from an existing model" on page 232.

The **Insert in package**: field is the optional name of the package the new model should be inserted into. The package must already exist. Leave this field empty if the model class should not be inserted into a package. Existing modifiable packages in the package browser are available in a drop-down list. You can click **Select Package** to browse for other packages. Note that you can click the **New Package** button after the **Select Package** button to create a new package. Clicking that button will open the dialog for creating a new package.

The **Open new class in:** dropdown list presents alternatives where the created model should be opened:



The default is different for different items created; compare with creating a package below.

For more about creating functions, see next chapter “Simulating a model”, section “Scripting”.

File > New > Type (enumeration)

Creates a new enumeration type. For more information about this, see section “Defining and editing enumeration types” starting on page 401.

(For all alternatives of the **File > New** command, see the section “The **File > New** command alternatives” above.)

File > New > Package

Creates a new package; as for creating new models, the user is asked for package name, a short description, optional base class to extend from, and the name of an optional parent package.

Dymola supports two ways of storing packages:

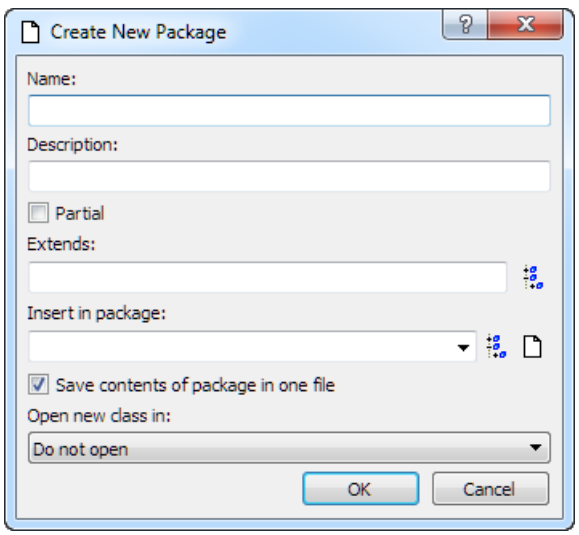
- Saving the package as a single file. All models etc. of the package are stored in this file. This alternative is recommended for small packages.
- Saving all models etc. of a package in separate files. The package itself is stored as a directory in the file system, plus a file called package.mo which stores package documentation and other annotations. This alternative is recommended for large packages.

Use separate files for concurrent development.

Saving the contents of a package in separate files supports multiple developers concurrently working on the package (editing different models, of course). When a version management system is used, storing models as separate files allows version control on a per-model basis.

Note: the storage format can be changed the package has been created see “Storing packages hierarchically” on page 329, or, for a top-level package using **File > Save As...** and not changing the name, see “File > Save > Save As” on page 457.

Creating a new package.



For other selections, see creation of a model above.

(For all alternatives of the **File > New** command, see the section "The **File > New** command alternatives" above.)

File > New > Script

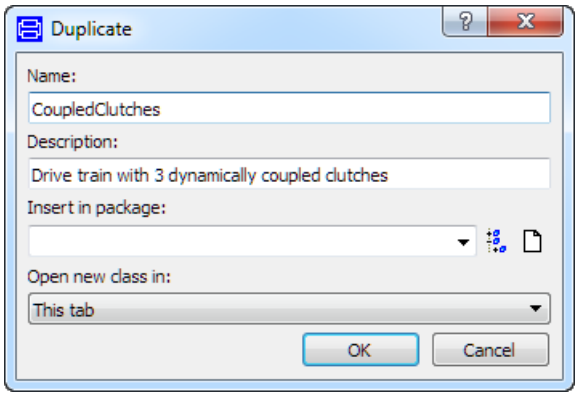
Creates a new script. For script handling, see next chapter, section "Scripting", subsection "The Dymola script editor".

(For all alternatives of the **File > New** command, see the section "The **File > New** command alternatives" above.)

File > New > Duplicate Class

Creates a new class with the same definition as the current class. The user must give the new class a unique name in the scope where it is inserted. If the package name is empty, the duplicate class is inserted in the global scope, i.e., outside of any packages. Note that you can also change if the package should be saved as one file or as a file hierarchy.

Duplicating a class.



Directories pointed to by LibraryDirectory and IncludeDirectory are copied, and the references are updated. Note that presently other resources (e.g. images) are not copied.

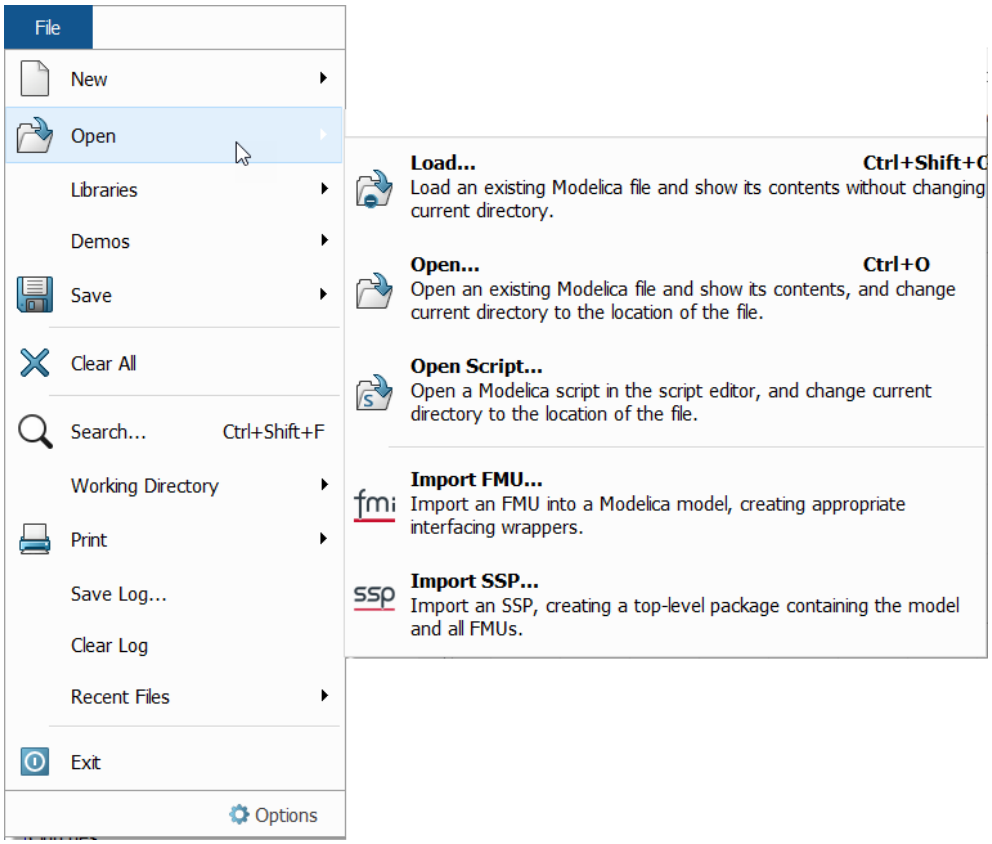
For other alternatives in this dialog, see creating a new model above.

(For all alternatives of the **File > New** command, see the section "The **File > New** command alternatives" above.)

File > Open

The **File > Open** command contain a number of subcommands.

The File > Open command.



Below the alternatives are described.

File > Open > Load...

Loads the contents of a Modelica file. The name of all loaded/read classes is shown in the status bar; errors found when parsing the Modelica model are shown in the log window. The number of loaded/read classes and the number of errors and warnings are shown after the file has been loaded.

The load operation does not change the current working directory.

(Note that by default the button  in the top left toolbar of the Dymola window is a shortcut for this command.)

If any referenced classes (used as base class or the class of a component) cannot be found, Dymola will search for additional packages defined in the directories of DYMOLAPATH or MODELICAPATH.

If the string $DYMOLA/Modelica/Library is not found in MODELICAPATH it is added first (and not last). The environment variable MODELICAPATH specifies a semi-colon separated list of directories where packages will be searched for. For more information on MODELICAPATH, see the chapter "Appendix – Installation". You can also use the index entry "*MODELICAPATH*" in the index of the manual.

Note that there is an option to use also the paths of enclosing packages in addition to the paths in MODELICAPATH. This makes it easier to, for example, automatically find multiple libraries that are stored in the same directory. See the reference above for details.

When dragging a .mo or .moe file into the Dymola main window; it will automatically be opened; the action corresponds to giving this load command.

File > Open > Open...

This command works as the above command, the only difference is that the current working directory *is* changed; Dymola will automatically change the current working directory to the directory of the opened file. Compare the above command.

(Note that you can add this command, as the button  , in the top left toolbar of the Dymola window – see section "Shortcuts to File commands in the top left toolbar" on page 446.)

File > Open > Open Script...

This command opens a Modelica script in the script editor, and change the current working directory to the location of the opened script file. For more information about script handling, see the next chapter, section "Scripting".

File > Open > Import FMU...

This command imports an FMU into a Modelica model, creating appropriate interface wrapping. This command corresponds to the built-in function `importFMU` of the selected FMU, with import settings defined in the **FMI Import** tab of the simulation setup. For more information about these settings, see index entry "*simulation setup : FMI import tab*" in this manual. Note that dragging an .fmu file into the Dymola main window will perform this command on it automatically.

File > Open > Import SSP...

This command imports a System Structure and Parameterization (SSP) package into Dymola. For more information about SSP, see the chapter "FMI, eFMI, and SSP Support in Dymola", section "SSP introduction".

File > Libraries

The libraries menu is extended when optional libraries are installed.

Displays a menu with shortcuts to known libraries. Selecting one of the libraries is equivalent to opening the filename shown in the status bar with **File > Open**.

It is possible to modify the list of know libraries. Please see the chapter "Appendix – Installation", section "Installation on Windows", sub-section "Additional setup".

In case you have many libraries, the **File > Libraries** menu may be quite large by default. To have a smaller menu in such a case, you can set the flag

```
Advanced.File.CompactMenu=true
```

(The flag is by default `false`.) Note that this flag only works on the **File > Libraries** menu, not on the general **File** menu.

File > Demos

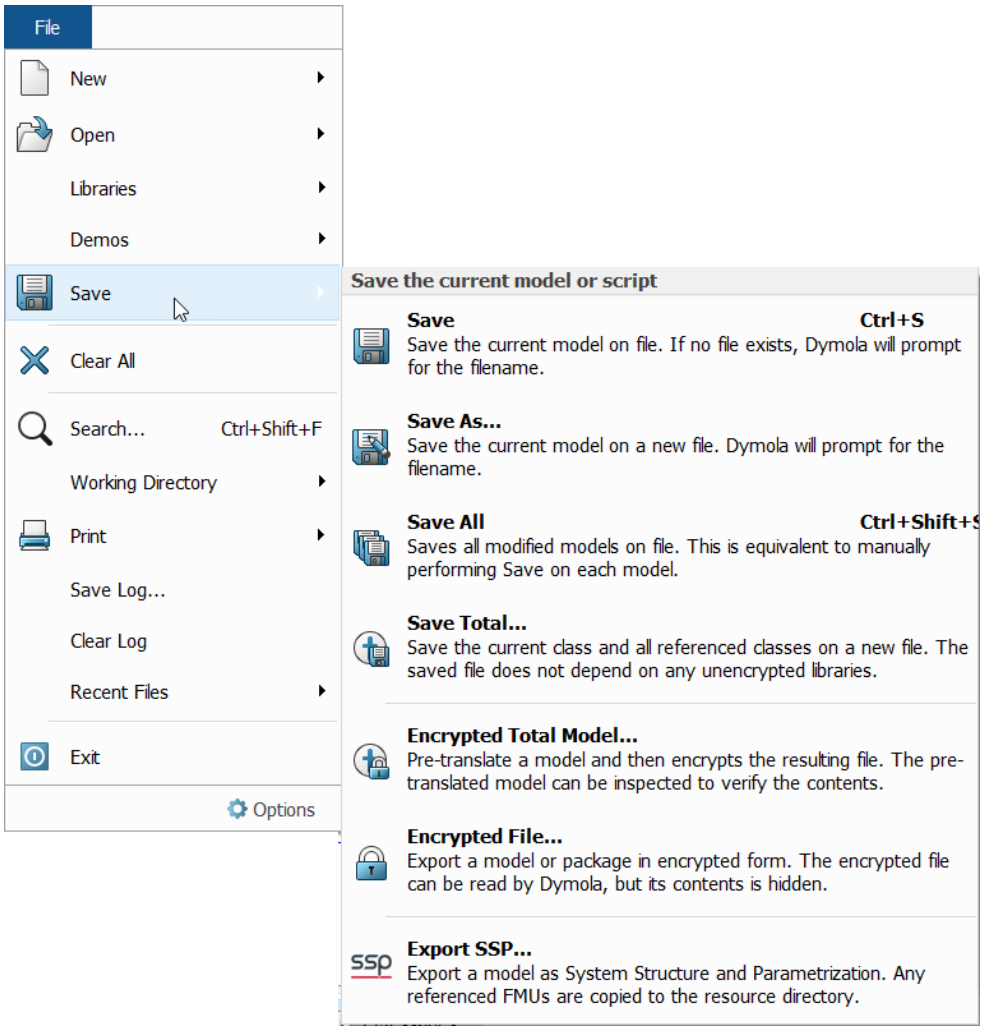
Displays a menu with shortcuts to predefined demo models. Note that in many cases there is a corresponding script file which should be opened after the model. See also the next chapter, section "Editor command reference", sub-section "Main window: Simulation menu", command "Simulation > Run Script…".

It is possible to modify the list of predefined demos. Please see the chapter "Appendix – Installation", section "Installation on Windows", sub-section "Additional setup".

File > Save

The **File > Save** command contain a number of subcommands.

The File > Save command.



Below the alternatives are described

File > Save > Save

Saves the class in the current window.

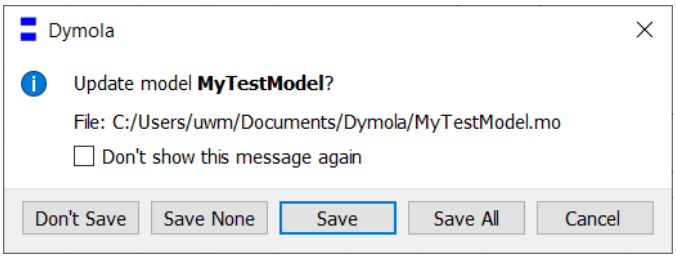
Notes:

- By default, the button  in the top left toolbar of the Dymola window is a shortcut for this command. You can also use the keyboard shortcut **Ctrl+S** to save.
- Dymola will always ask the user to save a modified class when terminating.

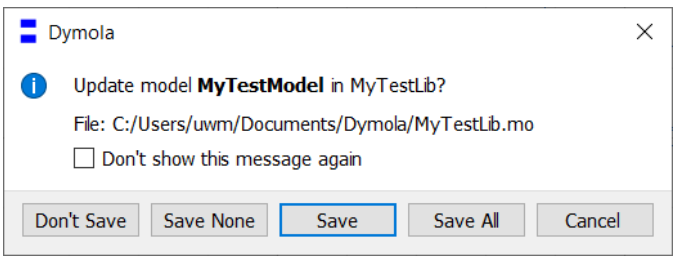
- You will have a status message that the model is saved in any case, in the status bar at the bottom of the Dymola window.

Saving by updating an existing model

If the original model definition was read from a file, you will have a dialog like below when applying the command for a stand-alone model:



If the model is included in a package, you may have the following dialog:



Clicking **Save** or **Save All**, the file is updated.

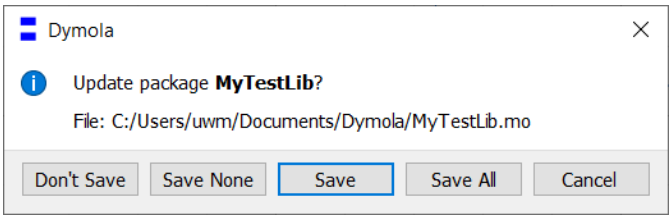
You can select to perform **Save** without displaying this dialog any of the following ways:

- Activating **Don´t show this message again** in the dialog above.
- Clearing the option **Prompt user when saving the model**, reached by the command **Tools > Options**, in the **Package Browser** tab. See “Package Browser options tab” starting on page 512.
- Setting the flag `Advanced.UI.PromptBeforeSave = false`. (The flag is by default `true`.)

Saving a new model

If you save a new standalone model, you a propted for a file name. The name of the class with file extension `.mo` is default.

If save a new model in an existing package, you get a dialog like the following, where you are asked to update the package, saving the model in the package:



Saving Unnamed

If the name of the class is "Unnamed" (for a new class), Dymola will ask for another class name.

File > Save > Save As...

Duplicates the current model and saves it to a new file. See also "File > New > Duplicate Class" on page 451. For a top-level class or package it is also possible to save it with the same name, but to a different file (or directory). Note that you can also change if such a package should be saved as one file or as a file hierarchy.

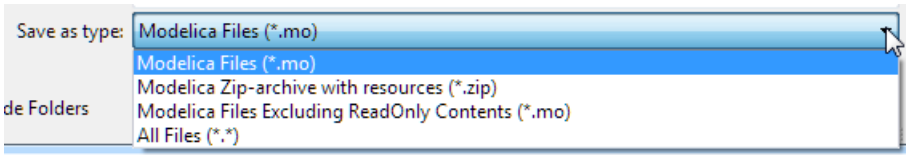
File > Save > Save All

Saves all modified model classes with a File > Save operation. (You can also use the keyboard shortcut **Ctrl+Shift+S** to perform this operation.)

File > Save > Save Total

Saves the class in the current window and all classes used by it. Resources can also be saved.

Whether resources should be saved is decided by selecting file type when saving:



- **Modelica Files (*.mo)** Creates a single file with the complete definition of a class, which is independent of other files and libraries.
- **Modelica Zip-archive with resources (*.zip)** Creates one zip-file with the class and all classes used by it – and all resources. There are two flags that can be used:
    - `Advanced.SaveTotal.CopyDocumentation` – this flag can be set to true if documentation should be included in the zip directory. "documentation" is here defined as directories named Documentation or files with suffix pdf, ppt, doc, docx, html, ott, and css. The flag is by default false. Note that other file formats than the ones listed are always included, e.g. files with suffix pptx.

- `Advanced.SaveTotal.CopyImages` – this flag is by default true, meaning that images are by default included in the zip directory. "images" is here defined as directories named Images, or files with suffix png or gif. If images should not be included, the flag can be set to false. Note that other file formats than the ones listed are always included, e.g. files with suffix stl and jpg.

- **Modelica Files Excluding ReadOnly Contents (*.mo)** Works as the first alternative above, but read-only content is excluded.

Note. Save Total with resources on Linux requires the Linux utility "zip". If not already installed, please install using your packaging manager (e. g. apt-get) or see e. g. http://www.info-zip.org.

File > Save > Encrypted Total Model...

For information about this command, please see chapter "Model Management", section "Encryption in Dymola".

File > Save > Encrypted File...

For information about this command, please see the chapter "Model Management", section "Encryption in Dymola".

File > Save > Export SSP...

Exports a Modelica model to the System Structure and Parameterization (SSP) format.

For more information about SSP, see the chapter "FMI, eFMI, and SSP Support in Dymola", section "SSP introduction".

File > Clear All

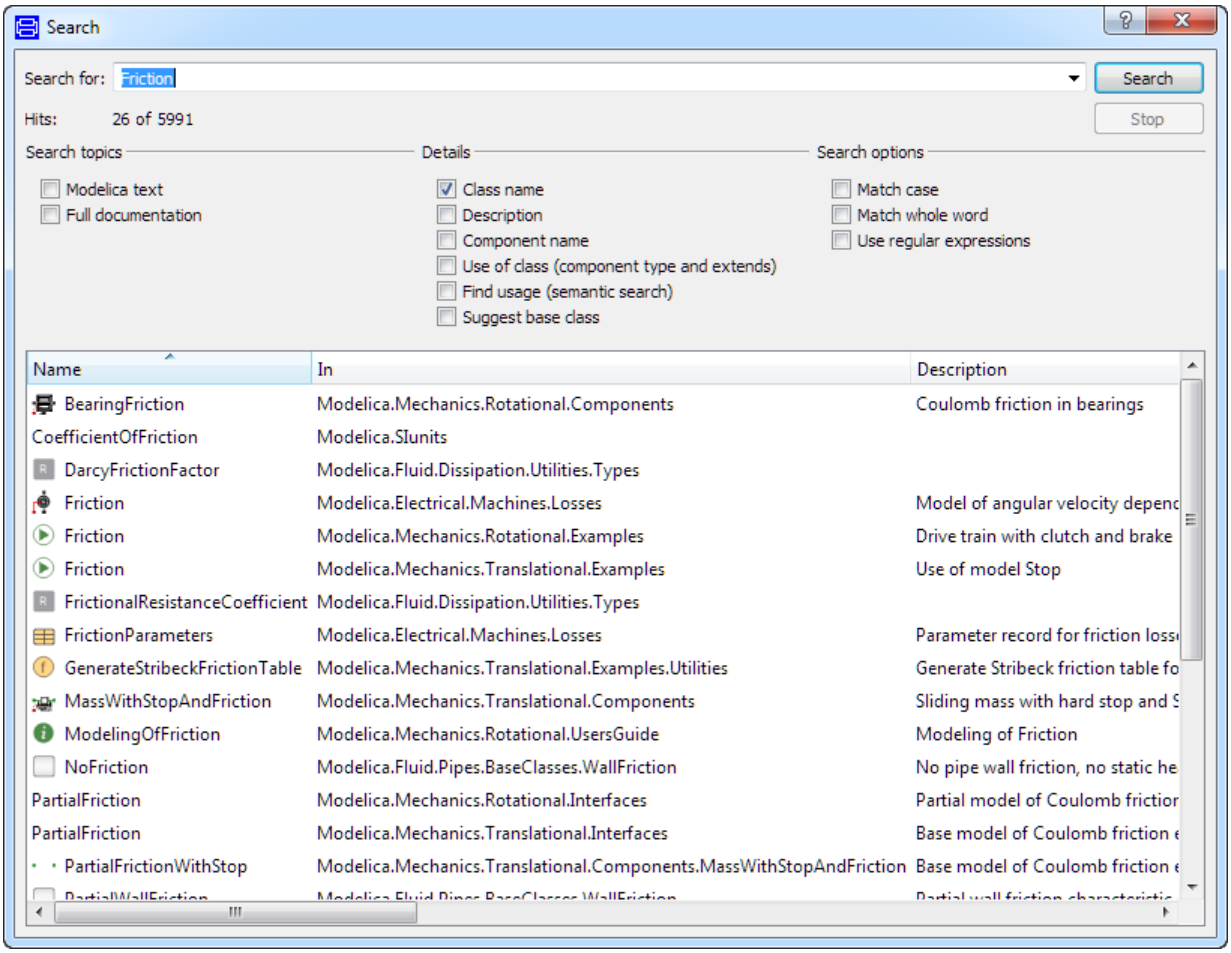
Performs a Save All operation and then removes all model definitions in Dymola, except the Modelica standard libraries.

File > Search



Shows all classes or components matching a search pattern. **File > Search...** will search all classes read by Dymola. Selecting **Search...** in the package browser (using the context menu of a class) will only search the current package.

Note that in the context menu of a class in the package browser two commands **Find Usage…** and **Suggest Base Class…** are available. These commands are actually the **Search…** command, but with specific settings. See the setting **Find usage (semantic search)** and **Suggest base class** below.



The search pattern can be matched against two general **Search topics** of the classes:

Modelica text Match search pattern anywhere in Modelica text, including in annotations. The full Modelica text of a package is searched, including classes inside it. If a hit is inside such a class, the location will be presented in the search result. Examples:

```
model M "pattern"
  Real x = pattern;
equation
  pattern = 0;
end M;
```

Full documentation Match search pattern against the full documentation of classes.

It is also possible to match the search pattern against specific **Details** of the model. This group is disabled if the Modelica text is searched.

Class name Match search pattern against names of classes.

Description Match search pattern against the description string of classes or components.

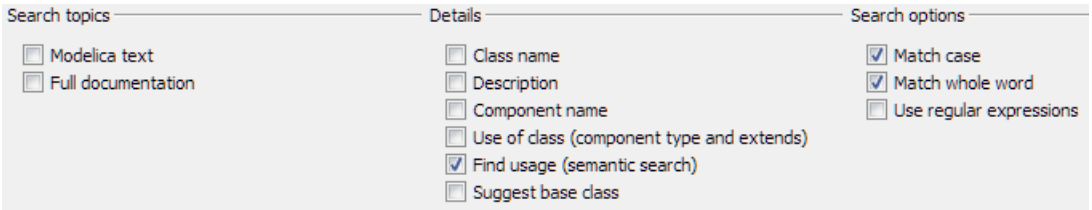
Component name Match search pattern against the names of components. Example:

```
model M
  CompType pattern;
end M;
```

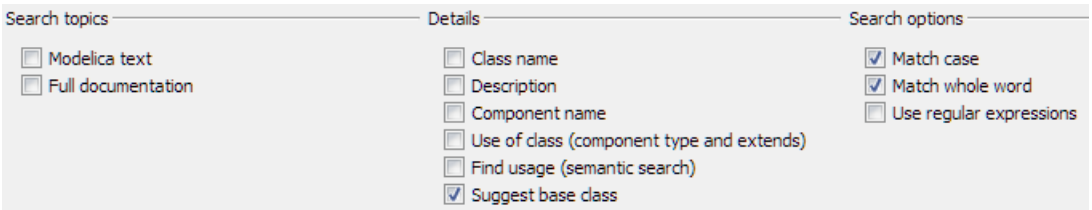
Use of class (component type and extends) Match search pattern against uses of class as the type of components and in extends clauses. Matches in redeclarations and extends are marked with a special icons. Example:

```
model M
  extends pattern;
  pattern comp_name;
  CompType c1(redeclare pattern c2);
end M;
```

Find usage (semantic search) Find usage of class using a semantic search. This setting enables finding e.g. items where the full name is not used. Such a search is also facilitated from the context menu of a class in the package browser, as the specific command **Find Usage...**. This command is actually a specific case of the context command **Search...**, with the following settings set by default:



Suggest base class Find base classes matching the selected class. This command is actually a specific case of the context command Search..., with the following settings set by default:



By double-clicking any of the results you can select to adapt the selected class to the found base class. Please see section "Building test-driven template models" starting on page 372 for an example how to do this.

The search options control how the matching is done for each searched item:

Match case Match upper and lower case exactly. If checked, the search pattern `a` will not match `A`.

Match whole word Match the complete contents of topics. If not checked, the search pattern will match parts of the topics.

Use regular expressions Regular expressions can be used for the search. Special symbols in the regular expression are:

*	Match everything.
?	Match any single character.
{ab}	Match characters a or b.
{a-z}	Match characters a through z.
{^ab}	Match any characters except a and b.
E+	Match one or more occurrence of E.
(ab|cd)	Match ab or cd.
\d	Match any digit.
\w	Match any digit or letter.
^	Match from start.
$	Match at end.

The search result at the bottom of the search dialog displays the name of the class or component matching, where the matching item is located, and the short description of the matching item. The results can be sorted by clicking on the corresponding heading.

Two operations are available on the matching items. Double-click on a matching class opens that class in the associated edit window. Double-click on a component opens the enclosing class and selects the matched item. Classes can be dragged into the graphical editor to insert a component.

A context mnu is available for the mathing items, with the following entries:

Open Class opens the class in the current tab.

Open Class in New Tab opens the class in a new tab.

Open Class in New Window opens the class in a new window

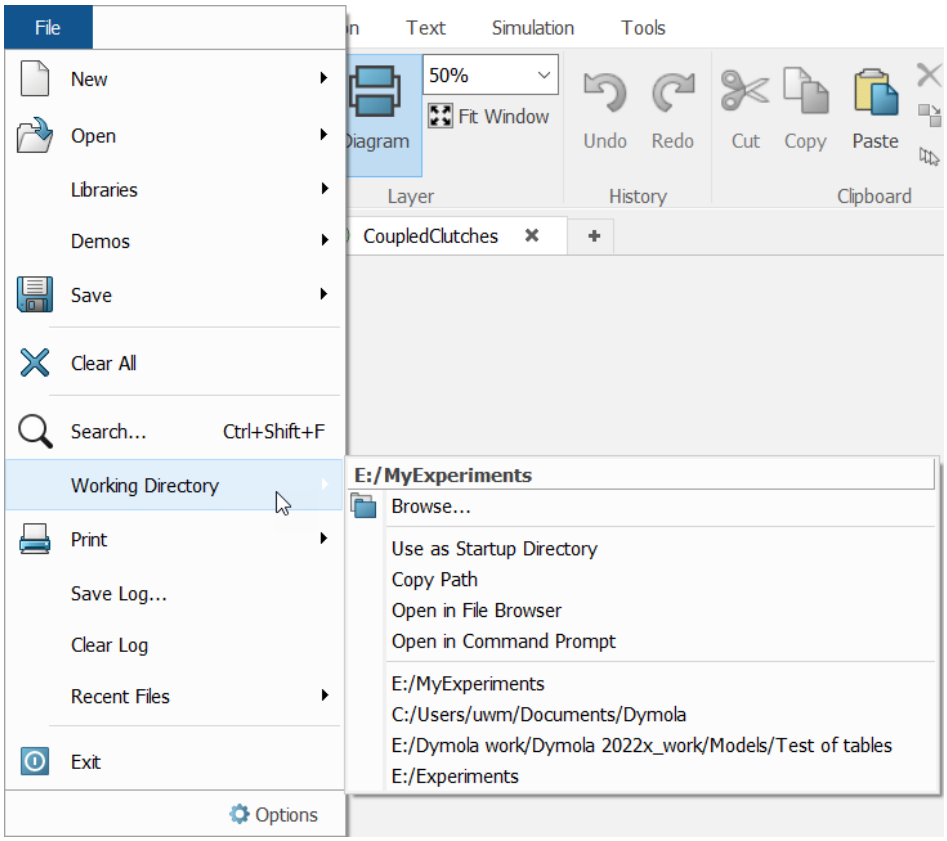
Copy Path copies the Modelica path to the clipboard

Info displays documentation for the class.

For further search inside the class a “find” command (e.g. the command **Graphics > Find**) can be used. Please see section “Graphics > Find” on page 482 for more information about this command.

File > Working Directory

Display a dialog which allows the user to change or display the current working directory by different commands, and also to save it as startup directory. The current working directory is used as the default location for opening files, and for saving simulation results.



You can (for details see further below):

- See the current working directory as a header in the menu (above **Browse**).
- Browse for a directory or select from a history of selected current working directories (last section of the menu).
- Set the current working directory as the startup directory to use as working directory when restarting Dymola.
- Copy the current working directory path to the clipboard.
- Open the current working directory in the file browser.
- Open the current working directory in a command prompt.

Note that the commands are available when simulating a model as well, and also from the **Working Directory** button in the Commands window.

More about current working directory, startup directory, and related commands

On Windows, the current working directory can be an UNC path. Note: Compilation is handled by copying to a temporary directory, which adds some time to the compilation.

Changing/displaying the current working directory may also be given as a textual command. Entering `cd` (without directory specification) in the Command input line of a Command window and pressing Enter prints the current working directory in the Command window. Adding the path will change the current working directory, for example entering `cd E:/MyExperiments`. The cd command is a built-in function in Dymola; you can find more information about it in the next chapter, section "Scripting", subsection "Built-in functions in Dymola".

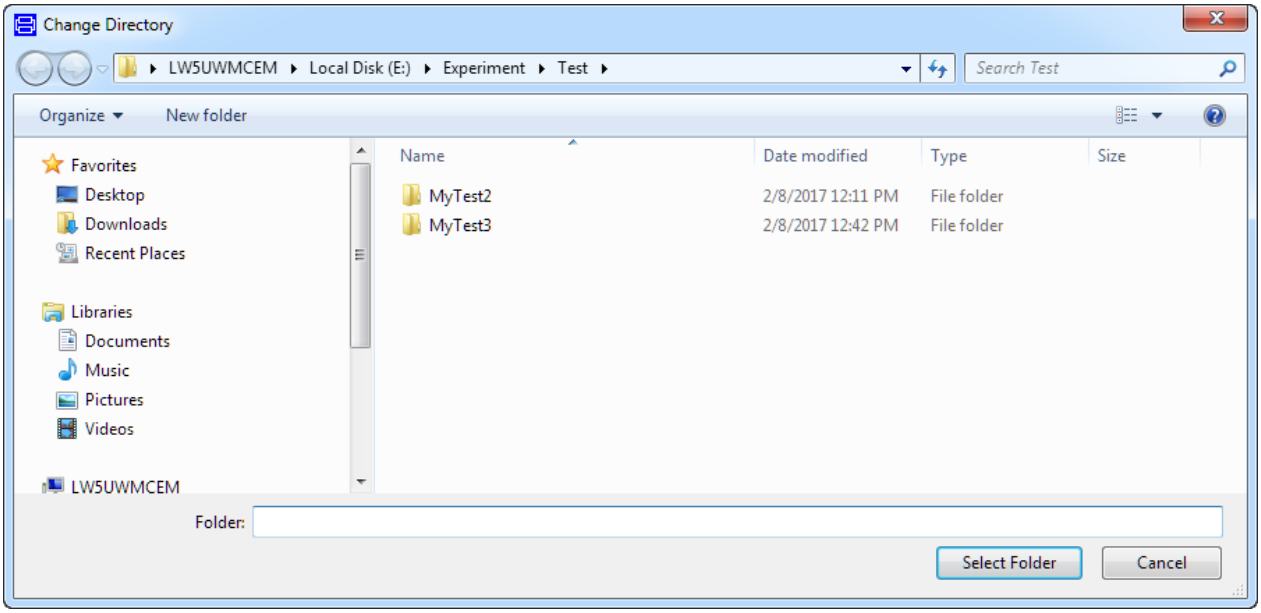
What directory that should be used as current working directory when starting Dymola again (the startup directory) can be set. For details, and alternatives, including the saving of settings, see the section "Settings options tab" starting on page 522. Please also note the command **File > Working Directory > Use as Startup Directory** below.

If no changes have been made by any command etc., the default working directory for Dymola is `…\Documents\Dymola`. This directory is created if it does not exist when installing Dymola.

Dymola defines an environment variable DYMOLAWORK which is set internally to the default startup directory (`…Documents\Dymola` if no default directory is set).

File > Working Directory > Browse...

Browses for a directory that will be the current working directory when clicking **OK** in the dialog. The browsing starts in the current working directory:



Note that you can also select from the history of current working directories. See the general overview of the working directory commands in section "File > Working Directory" above.

File > Working Directory > Use as Startup Directory

This command defines the current working directory as the startup directory, that is, the working directory to be used when starting Dymola the next time. This command is related to a setting in **Tools > Options**, the **Settings** tab. This setting defines the setting of the startup directory. See section “Settings options tab” starting on page 522.

File > Working Directory > Copy Path

Copies the current working directory path to the clipboard.

For a general overview of the working directory commands, see section “File > Working Directory” above

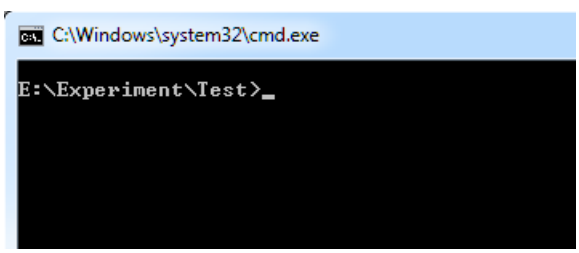
File > Working Directory > Open in File Browser

Opens the current working directory in the default file browser.

For a general overview of the working directory commands, see section “File > Working Directory” above.

File > Working Directory > Open in Command Prompt

Opens the current working directory in a command prompt, an example in Windows:

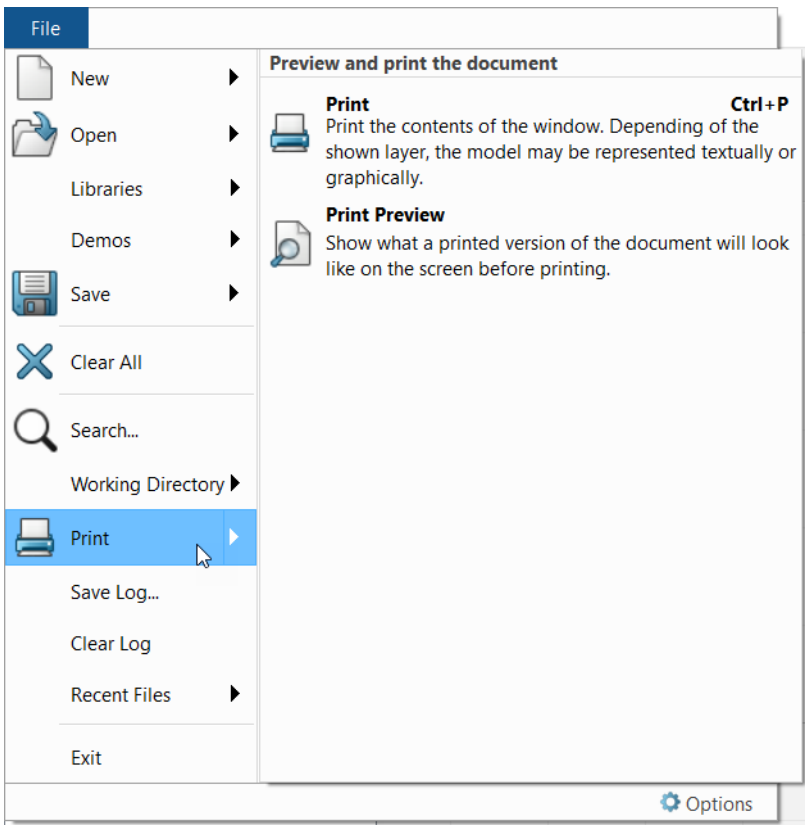


For a general overview of the working directory commands, see section “File > Working Directory” above.

File > Print

The **File > Print** command contains the following subcommands.

The File > Print command.



Below the alternatives are described

File > Print > Print...

Prints the contents of the window on the current printer.

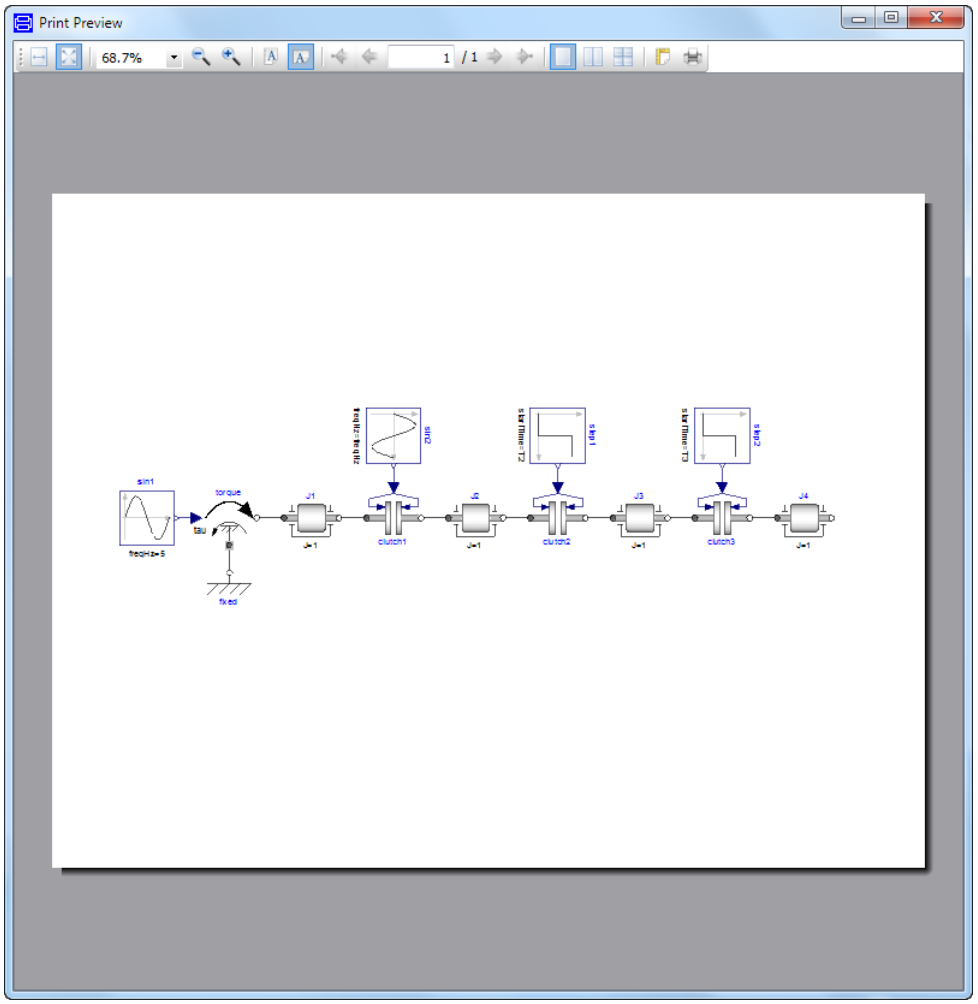
In Windows (not in Linux) you have an additional option: If the active window is a plot window, it is possible to set the resolution for high resolution export to clipboard (or printing). The variable is called `Advanced.ClipboardResolution` with the default 600 dpi.

Please note that this command also is presented in next chapter, since "active window" in that case includes e.g. plot window that implies other considerations.

File > Print > Print Preview...

Displays a print preview in a separate window.

(Note that you can add the button  in the top left toolbar of the Dymola window – that button is a shortcut for this command. See "Shortcuts to File commands in the top left toolbar" on page 446 for details.)



A number of commands are available in the top bar, some particularly useful features are:

- Selection of portrait/landscape 
- Print Setup 
- Print command 

File > Save Log...

Saves the contents of the command window to file. The user is prompted for a filename. The contents of the command log can be saved in three different formats.

- HTML format, including e. g. used features of the documentation editor. This format is the closest to the command log as shown in Dymola command window.

- Textual format, without any images but including command output.
- A script file (format `.mos`), containing all commands given but without the output from the commands.

For a more extensive description of the alternatives, see the index entry "*command : log file*" in the end of this book.

File > Clear Log

Erases the contents of the command log in the command log pane of the command window.

File > Recent Files

Shows a list of the most recent files which were opened with any of the commands **File > Open > Load...**, **File > Open > Open...**, or **File > Open > Import FMU...** (for more information about these commands, see these commands in the section "File > Open" starting on page 452). Selecting one of the files opens it again, typically to open models after **File > Clear All**. The content in the list is saved between Dymola sessions.

Please note that this command does not change the working directory of Dymola.

File > Exit

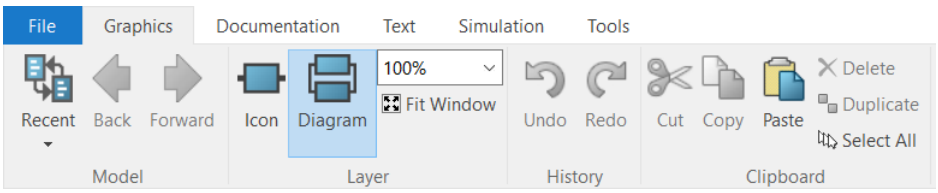
Terminates the program. Before terminating, Dymola will ask if modified model classes should be saved, one by one.

The exit command can also be given as a textual command `exit` for example in a script.

File > Options

This command is also available as **Tools > Options**. See that command in section "Tools > Options" starting on page 511.

4.6.2 Main window: Graphics tab



The **Graphics** tab is divided above into two images since it is long.

The **Graphics** tab contains commands to (the list corresponds to the tab groups):

- Navigating models
- Displaying icon or diagram layer
- Navigating history
- Handling clipboard
- Drawing primitives and handling them
- Tools (splitting models, check, grid, display connector names, handling attributes, and apply connection mode or transition mode)
- Finding items and filtering the diagram
- Handling variables, including creating states

Note that to make some commands clearer when outside context, they are presented with the tab group name included in brackets, either before or after the command, for example, **Graphics > Diagram [Layer]** and **Graphics > [Draw] Line**.

Shortcuts to Graphics tab commands in the top left toolbar

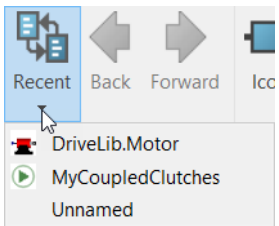


The top left toolbar of the Dymola window by default contains buttons to undo/redo the last performed change. For a description of these commands, see below. For configuring the top left toolbar, see “Shortcuts to File commands in the top left toolbar” on page 446.

Graphics > Recent



Clicking on the **Recent Models** button displays the previous top-level model shown in this window. Repeated clicks on this button toggles between the two most recently shown models. Clicking on the down-arrow displays a menu with recent models to allow arbitrary selection.



Graphics > Back

Display previous model viewed in the editor.

Graphics > Forward

Display the next model selected, if the **Back** command has been used before.

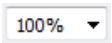
Graphics > Icon [layer]

Displays the icon layer of the class. Note that you can also use the corresponding button in the lower right toolbar of the window. See also “Class layers” on page 189.

Graphics > Diagram [layer]

Displays the diagram layer of the class. Note that you can also use the corresponding button in the lower right toolbar of the window. See also “Class layers” on page 189.

Graphics > Zooming



By selecting a zoom factor, more or less of the model is shown in the icon or diagram layers. Selecting 100% will scale the model so the coordinate system fits in the window. By selecting a larger zoom factor, parts of the model can be viewed in greater detail.

For more alternatives for moving and zooming, please see the section “Moving” and the section “Zooming” on page 217.

Graphics > Fit Window



Clicking on the **Fit Window** button will display the whole model in the window, independently if some parts are outside the coordinate system. In such case the resulting zoom factor differs from 100 %.

For more alternatives for moving and zooming, please see the section “Moving” and the section “Zooming” on page 217.

Graphics > Undo and Graphics > Redo

Undoes or redoes the previous editing operation. The commands are named after the previous editing operation, for example, looking at the menu in the figure above, the undo command available is **Undo Polygon**.

Note that these commands are by default also available in the top left toolbar of the Dymola window:



Graphics > Cut

Copies the current selection to an internal clipboard and deletes them. Text is copied to the operating system clipboard.

Graphics > Copy

Copies the current selection to an internal clipboard without deleting them. Text is copied to the operating system clipboard.

Graphics > Paste

Pastes objects from the internal clipboard to the current window. The current selection is not replaced by the pasted objects.

Graphics > Delete

Deletes the current selection. The objects are not placed in the internal clipboard. See also “Deleting objects” on page 220 regarding deletion of connections.

Graphics > Duplicate

Creates a duplicate set of the selected objects. Duplicated components are given new names by appending a digit to the old name, in order to avoid name conflicts. The new objects are offset one grid point from the originals. The new objects become the current selection.

Connections between duplicated components are also duplicated; connections to unselected objects are not.

Grapics > Select All

Selects all objects in the current window. Note that it also selects objects inherited from base classes.

Graphics > [Draw] Line

The command **Line** creates a line or a polygon. For general information about drawing, see the first part of “Creating graphical objects” on page 261; for specific information about drawing lines and polygons, see “Lines and polygons” on page 262.

Graphics > [Draw] Rectangle

The command **Rectangle** creates a rectangle. For general information about drawing, see the first part of “Creating graphical objects” on page 261; for specific information about drawing rectangles, see “Rectangles” on page 262.

Graphics > [Draw] Ellipse

The command **Ellipse** creates an ellipse. For general information about drawing, see the first part of “Creating graphical objects” on page 261; for specific information about drawing ellipses, see “Ellipses” on page 263.

Graphics > [Draw] Polygon

The command **Polygon** creates a polygon. For general information about drawing, see the first part of “Creating graphical objects” on page 261; for specific information about drawing polygons, see “Lines and polygons” on page 262.

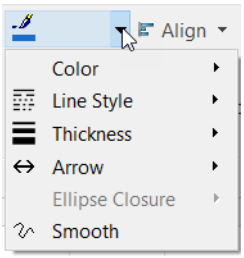
Graphics > [Draw] Text

The command **Text** creates a text. For general information about drawing, see the first part of “Creating graphical objects” on page 261; for specific information about drawing texts, see “Text” starting on page 265.

Graphics > [Draw] Bitmap

The command **Bitmap** inserts a bitmap in the diagram. For general information about drawing, see the first part of “Creating graphical objects” on page 261; for specific information about inserting bitmaps, see “Bitmap” starting on page 268.

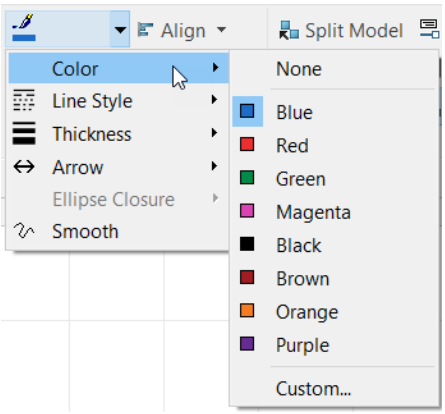
Graphics > Line Style



Clicking directly on the command, not the arrow next to it, opens the color menu for selecting a color for the line, see below.

The line style attributes that appears when clicking the arrow next to the command are

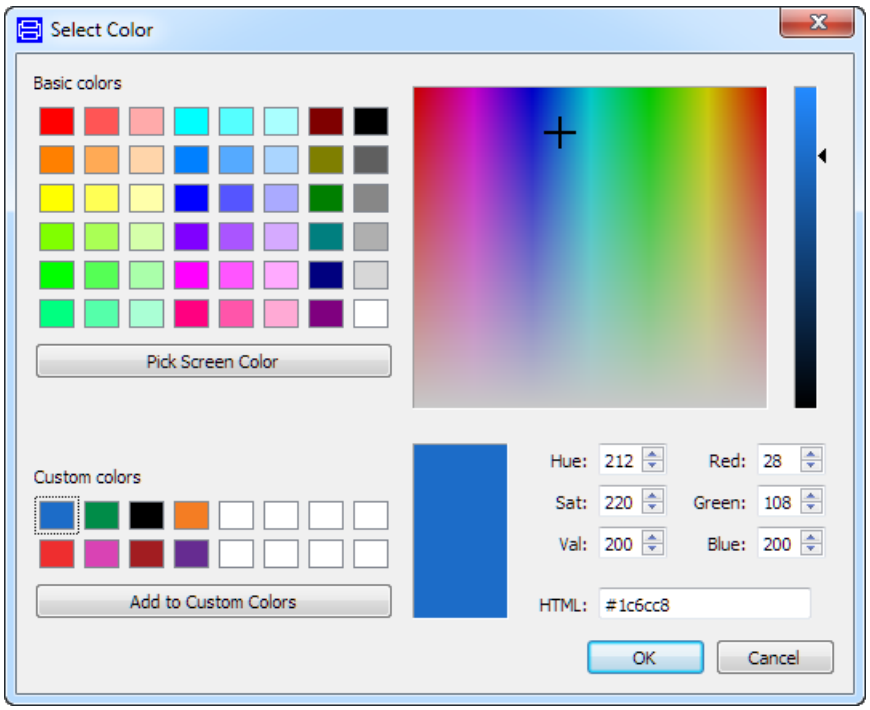
Color Selecting this alternative opens the following menu:



None means that the line will be invisible; can be used e.g. to draw a filled area without a border.

The colors directly selectable from this menu are the default colors of Dymola; these colors are the same as the default color for curves in the plot window. These colors cannot be edited. Blue is the default color.

Custom… Selecting this alternative will give a color palette (this palette is also displayed if you click the command button instead of clicking the arrow after it):



The color can be set in several ways:

- Click on a color in Basic colors or Custom colors.
- Click **Pick Screen Color**. You can now click anywhere in the screen to select that color.
- Define the color by specifying hue/saturation/value, (alternatively red/green/blue, or even the HTML coding). When using the latter alternative, either can values be directly entered, or the cross and the small arrow can be moved to specify the color.
- Click **OK** to conclude the setting.

Custom colors can be defined/changed the following way:

- First, click on the custom color that should be defined/changed. (In the figure above, the first white square under Custom colors is a natural choice, since no custom colors are yet defined.)
- Then set a color by e.g. moving the cross and arrow.
- Finally, click **Add to Custom Colors** to set the selected color as custom color.

Note. The custom colors in this menu are by default the same as the Dymola default colors mentioned above, but the custom colors here can be changed.

The color selection dialog has a choice of standard colors. Dymola supports true RGB color for all graphical objects, offering improved capabilities to design icons.

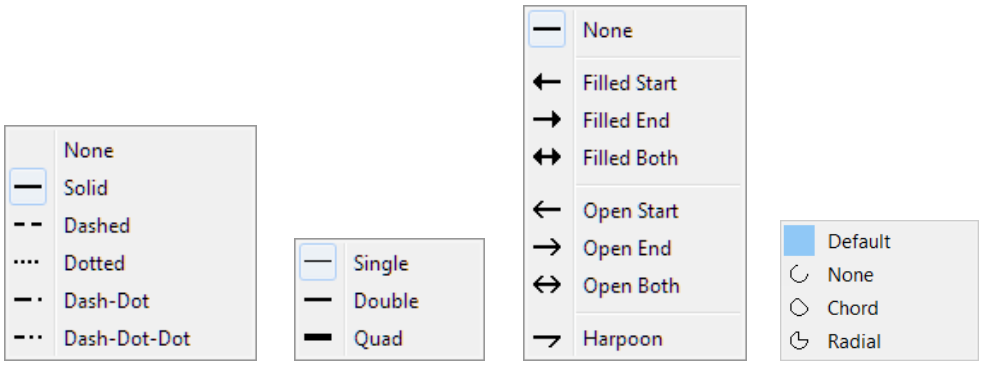
Line Style the alternatives are displayed in the first figure below from the left.

Thickness the alternatives are shown in the second figure below from the left.

Arrow – please see the third figure below from the left.

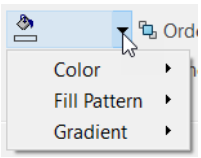
Ellipse Closure please see the last figure. For more information about ellipse closure, see section "Ellipses" on page 263.

Line Style, Thickness, Arrow, and Ellipse Closure.



Smooth converts sharp corners to rounded corners.

Graphics > Fill style



Clicking directly on the command, not the arrow next to it, opens the color menu for selecting a color for the fill, as for the previous command.

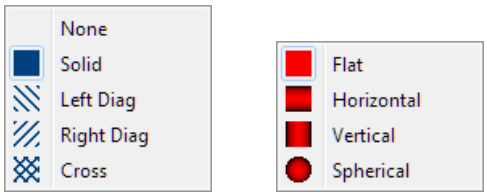
The fill style attributes that appears when clicking the arrow next to the command are:

Color Please see the corresponding section for Line style above.

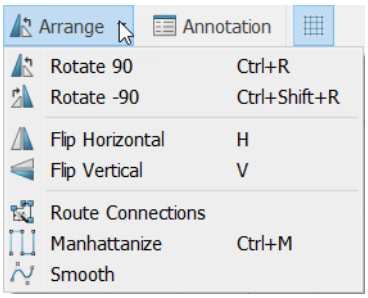
Fill Pattern The alternatives are displayed in the figure below to the left. Setting fill pattern to None, both the fill pattern and fill color are removed.

Gradient The alternatives (flat, horizontal, vertical or spherical) are displayed in the figure below to the right.

Fill pattern and Gradient.



Graphics > Arrange



The arrange operations include rotation, flips, route connections, Manhattanize and Smooth.

Rotate 90 Rotates the selected components 90 degrees clock-wise.

Rotate -90 Rotates the selected components 90 degrees counter-clock-wise.

Flip Horizontal Flips the selected components left-right.

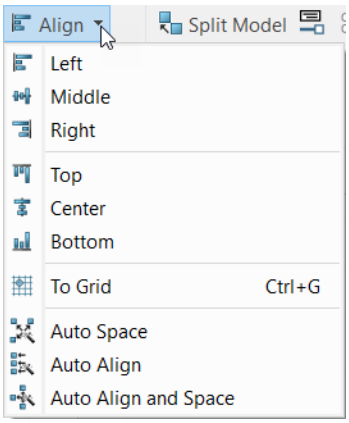
Flip Vertical Flips the selected components up-down.

Route Connections Routes selected connections. If nothing is selected, all selectable connections in the open class are routed.

Manhattanize Applies a "Manhattan" algorithm to make selected lines and connections more pleasing. Non-endpoints are moved to make all line segments horizontal or vertical.

Smooth Converts sharp corners to rounded corners if active. Smooth corresponds to an annotation. Please see the command Graphics > Annotation below.

Graphics > Align



The alignment and space operations are used to organize objects in the graphical editor. It is an alternative to moving the objects with the mouse or by using arrow keys.

The majority of commands align graphical objects with each other. Except for the three last commands, the first selected object is used as reference (and does not move).

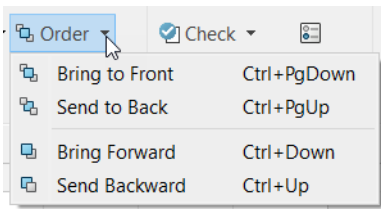
The alignment operations **Middle**, **Center** and **To Grid** use a center point for alignment. For graphical objects, the center point is the center of the bounding box of the object. For components and connectors, the origin of the coordinate system of the component's class is used as center.

First select the reference object. Then select the objects that should be aligned with the reference while holding down the **Shift** key. This creates a multiple selection with a known reference object. Finally, select the appropriate alignment from the menu. Horizontal alignment is specified with **Left**, **Middle** and **Right**, vertical alignment by **Top**, **Center** and **Bottom**.

To Grid aligns selected objects to the gridlines. The center is aligned to the nearest gridline or halfway between gridlines; this allows alignment of components of default size on gridlines or between gridlines. The keyboard shortcut is **Ctrl+G**.

The the last commands Auto Align, Auto Space, and Auto Align and Space align and space components automatically using algorithms. For more information, see section “Spacing and aligninig components automatically” on page 246.

Graphics > Order



The ordering operations are used to move graphical objects, components, or connections forward (drawn later) or backward (drawn earlier). The relative order of the selected objects is maintained. Note that for components, **Bring to Front** or **Bring Forward** means that the component is moved down in the component browser because components are drawn in order.

Bring to Front Brings the selected objects to the front, so they are not obstructed by any other objects.

Send to Back Sends the selected objects to the back, behind all unselected objects.

Bring Forward Brings the selected objects one step forward.

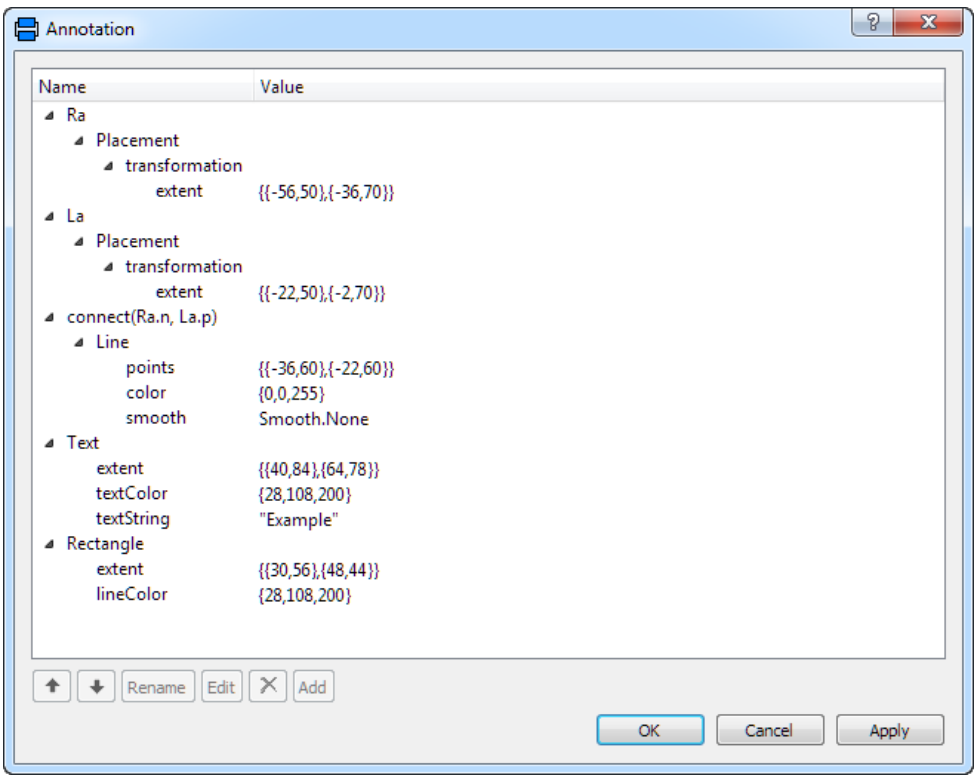
Send Backward Send the selected objects one step backward.

Graphics > Annotation

This command will display a dialog that can be used to change the graphical representation of the object by editing the graphical annotations of the selected object(s) or the model annotations.

The example below shows as one item the transformation extent of La and Ra, that is, the location of the two red handles of the objects. By clicking on the figures and changing them the extent of the object will be changed when pressing **OK**.

Annotation of the object.



For more information about viewing and editing annotations using this dialog, see section "Working with annotations using the Annotations dialog" starting on page 426.

Graphics > Split Model

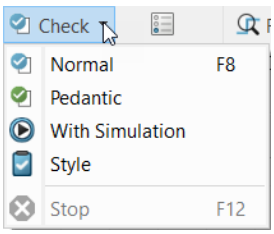
Creates a submodel or base class from selected components. Please see section "Splitting models" starting on page 338 for more information about this.

Graphics > Check



Clicking the **Check** button directly performs a **Normal** check; the first alternative below.

Clicking the arrow next to the command displays the alternatives:



Any of the commands checks the class in the current window for errors. The class is checked for syntactic errors, references to undefined classes and the consistency of equations. Syntax is highlighted. No simulation code is generated.

Important! The check commands can also be given from the **Text** tab. In the documentation of that tab, the check commands are described more in detail, partly since **Pedantic** and **With Simulation** is more related to that tab when it comes to code, partly since it is better to have the more detailed description of these commands, and related features, e.g. how to find unused parameters, in one place. For more information, see "Text > Check" on page 491.

- **Normal** issues warnings for non-critical issues that do not prevent simulating the model.
- **Pedantic** treat all warnings that are due to non-compliant Modelica code as errors, to force compliant Modelica code to be developed.
- **With Simulation** includes also associated commands in the check. The commands will be executed and the model simulated in the check.
- **Style** checks if the model conforms to the style guide when it comes to e.g. documentation. For more information about this, including the possibility to use a custom style check function, see index entry "*style checking : as the Check > Style command*" in the manual.
- **Stop** makes it possible to stop a check that takes too long time.

Graphics > Toggle Grid

Toggle Grid toggles the display gridlines to make alignment of objects easier.

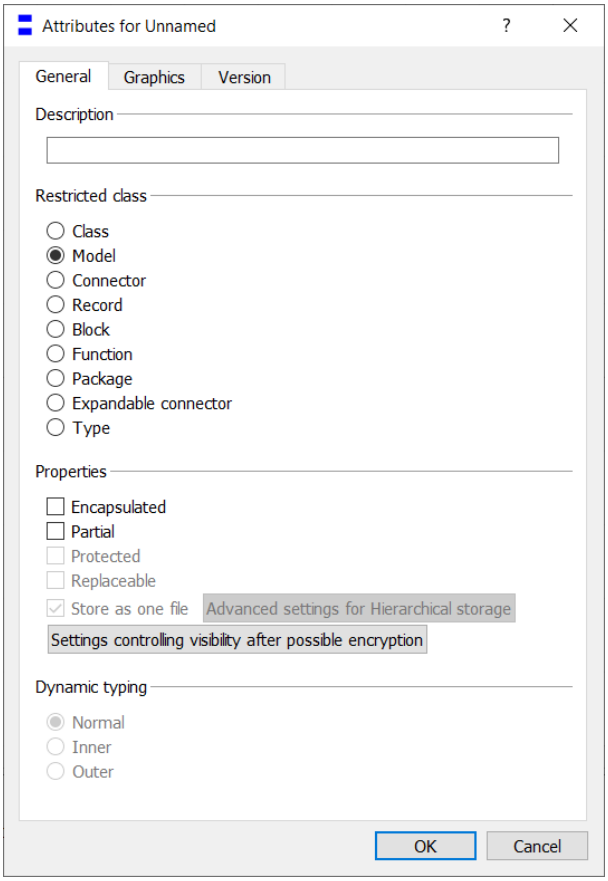
Graphics > Connector Names

Connector Names toggles the display of connector names in the diagram. For more information, see section "Displaying connector names in the diagram" on page 255.

Graphics > Attributes

General tab

General model attributes.

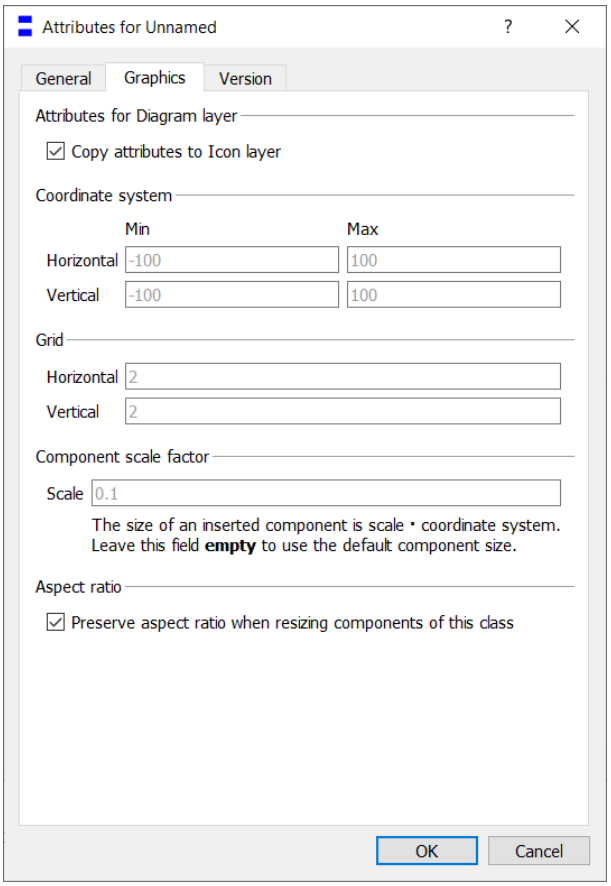


The **General** tab contains the (editable) description of the model and options related to model properties specified in the Modelica language.

- Restriction of the general class (e.g. model or connector).
- Aspects related to protection and redeclaration. It e.g. possible to edit the hierarchical storing of a selected package by clicking **Advanced settings for Hierarchical storage** - for an example of this and the possible alternatives, see section “Storing packages hierarchically” on page 329.
- Controlling visibility after possible encryption. Please see the chapter “Model Management” section “Encryption in Dymola”, sub-section “Special annotations for concealment”.
- Dynamic typing (**inner** and **outer**).

Graphics tab

Graphical model attributes.



The **Graphics** tab contains options that affect the graphics of the class. Please note that this menu also can be reached from the context menu of the selected coordinate system boundary by right-clicking and selecting **Attributes**.

The **Attributes for Diagram layer** group contains the **Copy attributes to Icon layer/Diagram layer** entry, which will copy all settings in this tab to the Icon layer or Diagram layer depending on what layer is open when giving the command. If not activated, the icon layer and the diagram layer can have different graphic settings (e.g. different coordinate system). Manually moving the bounding box of the coordinate system is also included in the setting. The setting corresponds to the flag `Advanced.AttributesBothLayers = true`. The setting/flag is not saved between sessions.

The **Coordinate system** group is used for changing the coordinate system of the current class. See also “Coordinate system” on page 197.

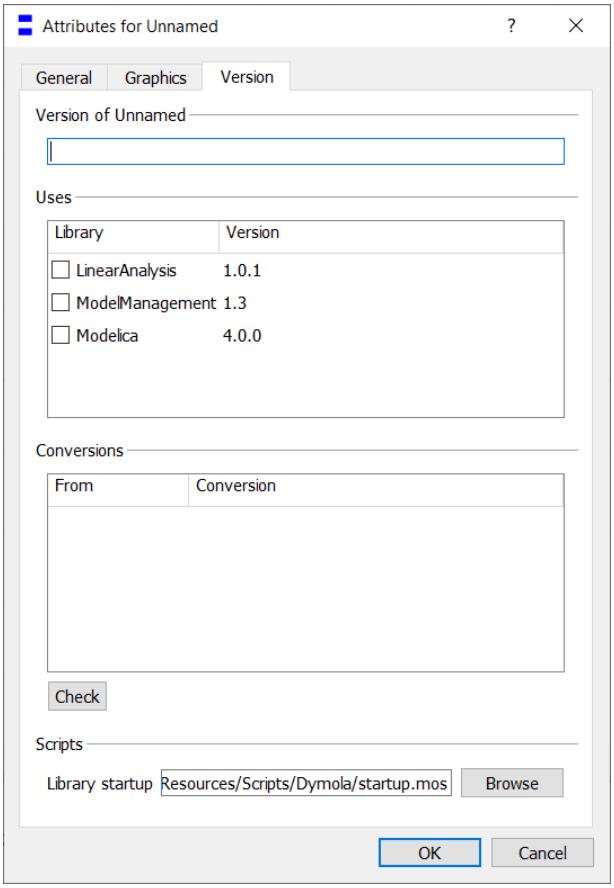
The **Grid** group is used for changing the grid size of the current class. For details about grid and gridlines, see section “Coordinate system” starting on page 197.

The **Component scale factor** group is defining the size of components which are dragged into a model. The component scale factor defines the default size of a component of the class. Assuming a model M has defined the scale factor 0.1, dragging class M into another model A will create a component, which size is "scale" times the size of the coordinate system extent of M. The value 0.1 is used as default if you change the coordinate system. For models with a non-square coordinate system, this change will give a natural component shape.

The Aspect ratio group contains **Preserve aspect ratio when resizing components of this class** that should be ticked if the aspect ratio should be preserved when components are resized.

Version tab

Version attributes of model.



For information about the groups **Version**, **Uses**, and **Conversions** in the **Version** tab , please see the chapter “Appendix - Migration”, section “Migrating to newer libraries”, sub-section “Specifying the version of a a package and checking of conversion scripts”.

The **Scripts** group makes it possible to browse for a library startup script for a top-level package, if you don´t want to use the default script. Given the name of the package is

ThisPack, the default library startup script is `modelica://ThisPack/Resources/Scripts/Dymola/startup.mos`. As long as this default is kept, no annotation is created, and that startup script is used, if existing. However, if you change the path, for example by changing the name of the script to `Mystartup.mos`, and browse for that script file, the following annotation `__Dymola_startup` will be automatically created:

```
package ThisPack
annotation(__Dymola_startup =
"modelica://ThisPack/Resources/Scripts/Dymola/Mystartup.mos");
end ThisPack;
```

Graphics > Connect Mode

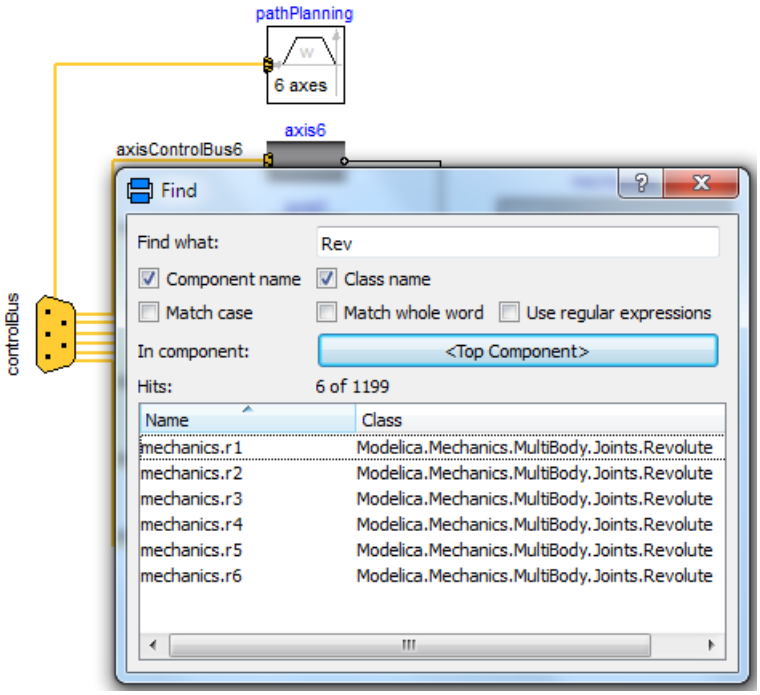
This command enables or disables drawing connections in the diagram. For more information, see section “Connections” on page 247.

Graphics > Transition Mode

This commands toggles drawing of transitions and initial states. For more information, see section “State Machines” starting on page 331.

Graphics > Find

The command **Graphics > Find** will search for components in the diagram layer. In the example below, `Rev` is searched in the Robot model:



Some features of this command:

- The search is recursive and automatic; i.e. you can start typing `Rev` and the result list is automatically built while you type, including components inside components etc.
- The search first looks at local components, before looking at sub-components.
- The search is based on what is shown in the diagram layer when starting the search.
- Clicking on a result will select it; the diagram layer will be zoomed to display the result. This navigation does not influence the search, you can still select another component, or you can modify the search. If you click the **In component** button, the original zooming is displayed.
- You can use wildcards if you select **Use regular expressions**; see below.

Graphics > Find Connection

The command searches the diagram layer for connections and the corresponding connected components. Some features of the command:

- The search is based on what is shown in the diagram layer when starting the search.
- Clicking a result will select the corresponding component; if the component is inside another component than the one displayed, the **Show Component** command is used automatically to display the diagram of that component to select the searched component. This navigation does not influence the search; you can still select another

component. You can go back to the original zooming by clicking the button after **Find connection in component**.

For details, please see section “Searching connections and connected components” starting on page 257.

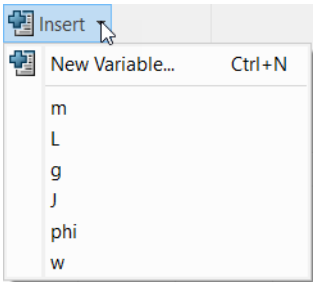
Graphics > Diagram Filter

Filter to dim component from selected packages and/or not matching a name. See section “Filtering objects” on page 222 for more information.

Graphics > Insert [Variables]

This command is used to insert new variables and edit existing variables in the class.

Clicking the arrow next to this command when a class is selected could give the following:



The command **New variable...** can be used to declare new variables. The command will display a variable declaration dialog without any predefined variable. Please see section “Variable declaration dialog” starting on page 378. Note that you can give this command by clicking directly on the **Insert** command.

The other entries are the variables in the selected class. By selecting any of these the variable declaration dialog for that variable is shown.

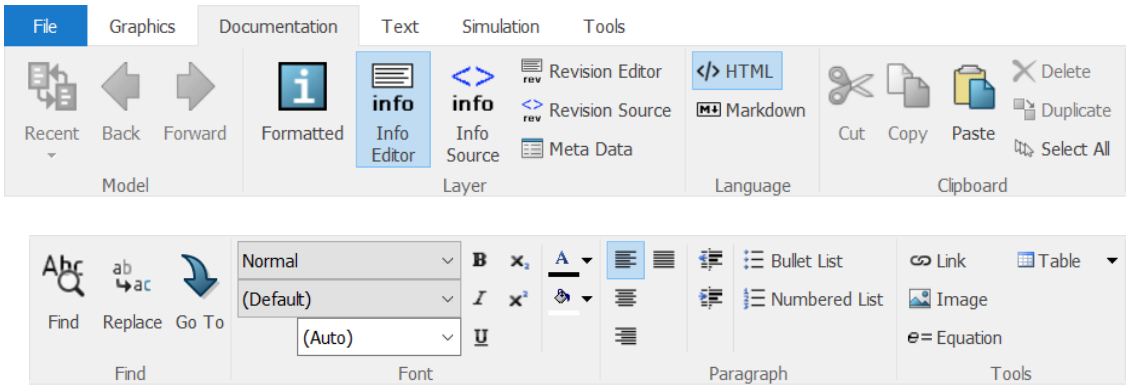
Graphics > Variable Selections

Opens a user interface for editing variable selections. See “User interface for displaying variable selections in the variable browser” starting on page 419.

Graphics > Create Local State

Creates a state in a state machine. For more information about state machines, see section “State Machines” starting on page 331.

4.6.3 Main window: Documentation tab



The **Documentation** tab is divided above into two images since it is long.

The **Documentation** tab contains commands to (the list corresponds to the tab groups):

- Displaying documentation layers and editors
- Handling clipboard
- Finding and replacing items and go to specific code line
- Editing the documentation layer (Font, Paragraph, and Tools group)

Note that to make some commands clearer when outside context, they are presented with the tab group name included in brackets, either before or after the command, for example, **Documentation > Formatted [documentation]**.

The first three commands are generic ones; see these commands in section “Main window: Graphics tab” on page 467.

Note!

For the commands in the Font, Paragraph, and Tools group of the tab, see section “Details of the documentation editor” starting on page 310.

Documentation > Formatted [documentation]

Clicking on the **Formatted** button displays the formatted documentation of the class. This is the default mode, and the result of editing in the below modes. Note that you can also use the corresponding button in the lower right toolbar of the window. See also “Class layers” on page 189.

For more information on how to work with the different documentation modes, see section “The content of the documentation layer” starting on page 301, and the following sections.

Documentation > Info Editor

Opens the documentation editor, to edit the information part of the formatted documentation. If the class is read-only, this button will not be active.

For more information on how to work with the different documentation modes, see section "The content of the documentation layer" starting on page 301, and the following sections.

Documentation > Info Source



Opens the documentation editor to edit the HTML text of the information part of the formatted documentation.

For more information on how to work with the different documentation modes, see section "The content of the documentation layer" starting on page 301, and the following sections.

Documentation > Revision Editor



Opens the documentation editor to edit the revision part of the formatted documentation. If the class is read-only, this button will not be active.

For more information on how to work with the different documentation modes, see section "The content of the documentation layer" starting on page 301, and the following sections.

Documentation > Revision Source



Opens the documentation editor to edit the HTML text of the revision part of the formatted documentation.

For more information on how to work with the different documentation modes, see section "The content of the documentation layer" starting on page 301, and the following sections.

Documentation > Meta Data



Opens a dialog to create and edit metadata.

For more information about metadata, see section "Generating and editing metadata" on page 320.

Documentation > HTML



Select HTML as documentation language. By default, HTML is selected.

For more information about HTML and markdown, see section "Generating and editing metadata" on page 320.

Documentation > Markdown



Select Markdown as documentation language. By default, Markdown is not selected.

For more information about HTML and markdown, see section "Generating and editing metadata" on page 320.

Documentation > Undo and Documentation > Redo

Undoes or redoes the previous editing operation.

Note that these commands are by default also available in the top left toolbar of the Dymola window:



Documentation > Cut

Copies the current selection to an internal clipboard and deletes them. Text is copied to the operating system clipboard.

Documentation > Copy

Copies the current selection to an internal clipboard without deleting them. Text is copied to the operating system clipboard.

Documentation > Paste

Pastes objects from the internal clipboard to the current window. The current selection is not replaced by the pasted objects.

Documentation > Delete

Deletes the current selection. The objects are not placed in the internal clipboard.

Documentation > Duplicate

Creates a duplicate set of the selected items. The new objects are offset one grid point from the originals. The new items become the current selection.

Documenation > Select All

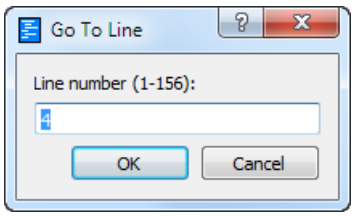
Selects all items in the current window.

Documentation > Find and Documentation >Replace

The commands finds and replaces text in the editor, if the text is not read-only. If the Modelica text is read-only.

Documentation > Go To...

Go to a specified line in a line editor. The command will display:

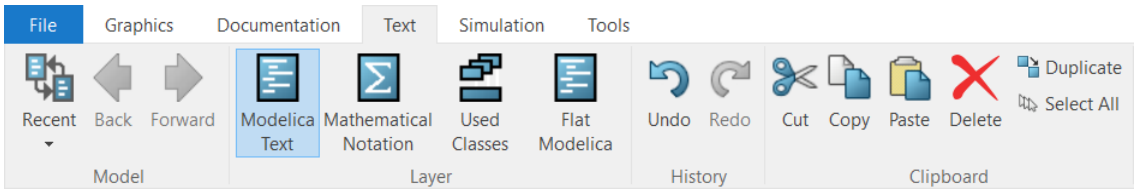


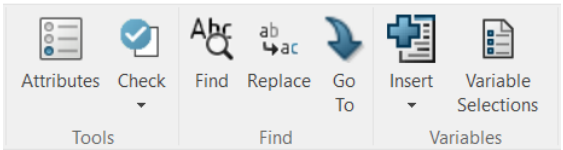
Entering the line number and pressing Enter will place the cursor on the wanted line.

Note!

For the commands in the Font, Paragraph, and Tools group of the tab, see section “Details of the documentation editor” starting on page 310.

4.6.4 Main window: Text tab





The **Text** tab is divided above into two images since it is long.

The **Text** tab contains commands to (the list corresponds to the tab groups):

- Navigating models
- Displaying documentation, Modelica text, or Used Classes layer
- Navigating history
- Handling clipboard
- Drawing primitives and handling them
- Tools (handling attributes and checking)
- Finding and replacing items and go to specific code line
- Handling variables

Note that to make some commands clearer when outside context, they are presented with the tab group name included in brackets, either before or after the command, for example, **Text > Documentation [Layer]**.

Shortcuts to Text tab commands in the top left toolbar

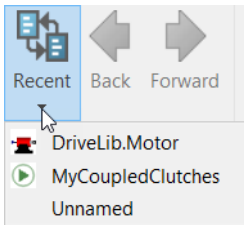


The top left toolbar of the Dymola window by default contains buttons to undo/redo the last performed change. For a description of these commands, see below. For configuring the top left toolbar, see “Shortcuts to File commands in the top left toolbar” on page 446.

Text > Recent



Clicking on the **Recent Models** button displays the previous top-level model shown in this window. Repeated clicks on this button toggles between the two most recently shown models. Clicking on the down-arrow displays a menu with recent models to allow arbitrary selection.



Text > Back

Display previous model viewed in the editor.

Text > Forward

Display the next model selected, if the **Back** command has been used before.

Text > Modelica [layer]



Clicking on the **Modelica** button displays the declarations and equations of the class - the Modelica text of the class. This is the default mode. Note that you can also use the corresponding button in the lower right toolbar of the window. See also “Class layers” on page 189; the icon is the same as the one for the documentation layer.

For more information about how to work with Modelica Text, see section “Looking at the Modelica Text layer” starting on page 272, and the following sections.

Text > Mathematical Notation



This mode displays the equations and declarations of the class in mathematical format. Note that there are some limitations when using this format.

For more information about how to work with Mathematical Notation, see section “Looking at the Modelica Text layer” starting on page 272, and the following sections.

Text > Used Classes [layer]



Clicking on the **Used Classes** button displays the base classes, classes used for components and connectors, and the Modelica text of the class. This facilitates searching for the location of e.g. a specific variable. The icon is the same as the one for the documentation layer. Note that you can also use the corresponding button in the lower right toolbar of the window. See also “Class layers” on page 189.

Text > Flat Modelica



This mode displays the flat Modelica text is showed instead. There are some limitations what can be displayed in this mode. The flat Modelica text is the result when component structure and inheritance has been evaluated; it corresponds to the result when translating the item. The result is a more readable representation; the feature is valuable when investigating more complex models.

Note that the text in the Used Classes layer cannot be edited.

For more information about the Used Classes layer, see section “Inspecting code in the Used Classes layer” starting on page 297.

Text > Undo and Text > Redo

Undoes or redoes the previous editing operation.

Note that these commands are by default also available in the top left toolbar of the Dymola window:



Text > Cut

Copies the current selection to an internal clipboard and deletes them. Text is copied to the operating system clipboard.

Text > Copy

Copies the current selection to an internal clipboard without deleting them. Text is copied to the operating system clipboard.

Text > Paste

Pastes objects from the internal clipboard to the current window. The current selection is not replaced by the pasted objects.

Text > Delete

Deletes the current selection. The objects are not placed in the internal clipboard. See also “Deleting objects” on page 220 regarding deletion of connections.

Text > Duplicate

Creates a duplicate set of the selected objects. Duplicated components are given new names by appending a digit to the old name, in order to avoid name conflicts. The new objects are offset one grid point from the originals. The new objects become the current selection.

Connections between duplicated components are also duplicated; connections to unselected objects are not.

Text > Select All

Selects all objects in the current window. Note that it also selects objects inherited from base classes.

Text > Attributes

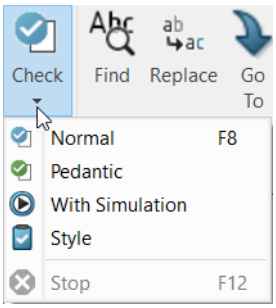
This command is the same command as the command **Graphics > Attributes**. See that command in section "Graphics > Attributes" starting on page 479.

Text > Check



Clicking the **Check** button directly performs a **Normal** check; the first alternative below.

Clicking the arrow next to the command displays the alternatives:



Any of the commands checks the class in the current window for errors. The class is checked for syntactic errors, references to undefined classes and the consistency of equations. Check can also be made to find unused parameters, see below. Syntax is highlighted. No simulation code is generated.

- **Normal** This command issues errors for critical issues, and warnings for non-critical issues that do not prevent simulating the model. This is the default selection.
- **Pedantic** This command treats all warnings that are due to non-compliant Modelica code as errors, to force compliant Modelica code to be developed. For more information about pedantic check and mode, see section "Forcing Modelica compliance" on page 290.
- **With Simulation** This command includes also associated commands in the check. The commands will be executed and the model simulated with stored simulation setup. Please see index entry "*model : associated commands*" in the index in the end of this manual for more information about associated commands.
- **Style** checks if the model conforms to the style guide when it comes to e.g. documentation. For more information about this, including the possibility to use a custom style check function, see the index entry "*style checking : as the Check > Style command*" in this manual.

-  **Stop** This command can be used to stop the check, in particular if using With Simulation, if needed.

Finding unused parameters

It is possible to find unused parameters when checking the model, translating it or simulating it. This is done by setting the following flag before performing any of these actions:

```
Advanced.Check.WarnAboutUnreferenced = true
```

(By default, the flag is `false`.)

Checking of sizes

When checking a package, the following output is the best result:

```
The model has the same number of unknowns and equations.
```

However, any of the following two results are also acceptable:

```
The model has the same number of unknowns and equations for the
given bindings of parameters.

The model has the same number of unknowns and equations for the
given numerical settings of parameters.
```

When checking a model or a connector the message also includes the number or symbolic expression for the size.

By setting the flag

```
Advanced.LogSymbolicSizeCheck=true
```

you also get the numbers when checking a package. Moreover, there might be warnings when symbolic checking is applied to parameters that were already evaluated.

Settings

There are some settings of what should be reported when checking, and also what should be treated as warnings or as errors. Please see index entry "*simulation setup : translation tab*" in the index in the end of the manual for more information.

Handling of built-in functions

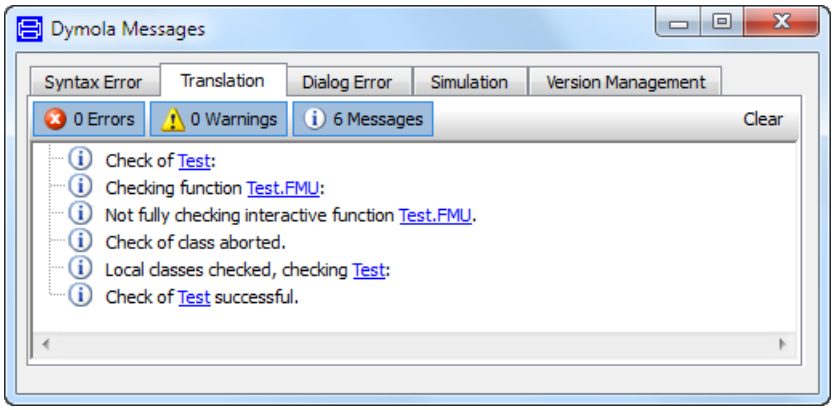
Check will warn for built-in functions not being defined in the Modelica Language Specification. Since a number of built-in functions not being defined in the Modelica Language Specification are present, an annotation `annotation(__Dymola_interactive=true)` can be used when wanting to use such functions in packages without warnings.

Example:

To be able to use the built-in function `translateModelFMU` in a package `Test` without warning when checking the package, create a function `FMU` with the annotation above and the wanted built-in function inside it:

```
package Test
  function FMU
  algorithm
     translateModelFMU("Modelica.Mechanics.Rotational.Examples.
       CoupledClutches", true);
     annotation(__Dymola_interactive=true);
  end FMU;
end Test;
```

The result of checking the package will be:



(Checking only the function separately will still report a warning.)

The check will both with and without the above annotation be less thorough than for normal functions; but performs some checks, e.g. the spelling of function names.

Text > Find and Text >Replace

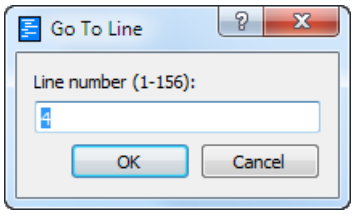
Text > Find and Text > Replace finds and replaces text in the Modelica text editor. If the Modelica text is read-only, the replace command is dimmed.

For more information about these commands, see **Find and Replace** in section “Context menu: Edit window (Modelica Text layer)” starting on page 533.

Note that the **Text > Find** command is used inside a class. To instead search for classes, the command **File > Search...** can be used. Please see section “File > Search” on page 458 for more information.

Text > Go To...

Go to a specified line in a line editor. The command will display:

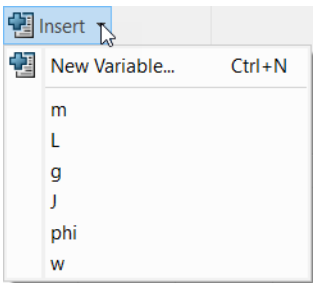


Entering the line number and pressing Enter will place the cursor on the wanted line.

Text > Insert [Variables]

This command is used to insert new variables and edit existing variables in the class.

Clicking the arrow next to this command when a class is selected could give the following:



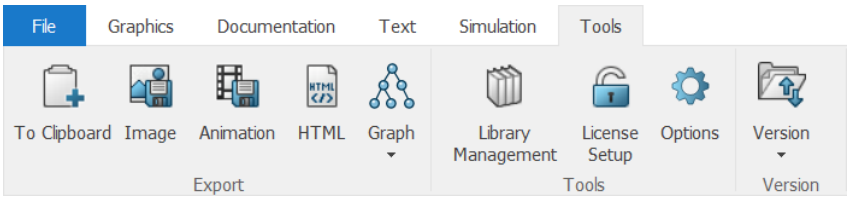
The command **New Variable...** can be used to declare new variables. The command will display a variable declaration dialog without any predefined variable. Please see section "Variable declaration dialog" starting on page 378. Note that you can give this command by clicking directly on the **Insert** command.

The other entries are the variables in the selected class. By selecting any of these the variable declaration dialog for that variable is shown.

Text > Variable Selections

Opens a user interface for editing variable selections. See "User interface for displaying variable selections in the variable browser" starting on page 419.

4.6.5 Main window: Tools tab



The **Tools** tab is divided above into two images since it is long.

The **Tools** tab contains commands for export, various setup, and help/documentation. Custom menus and toolbars can also be found here.

Note that to make the commands for export more clear when looking at them directly, they are presented below with the tab group name included in brackets; for example, **Tools > [Export] To Clipboard**.

Tools > [Export] To Clipboard

Copies the contents of the active window (without window borders) to the clipboard in Enhanced Metafile Format (.emf), which can then be pasted in common word processors.

Such an image can be rescaled and edited in e.g. Microsoft Word.

If the active window is anything else but an animation window or a text window, please note that some old programs (for example Microsoft Word 97) may have problems when high resolution images or pictures containing rotated text are pasted. We suggest using **Edit > Paste Special** and selecting Enhanced Metafile Format (Windows Metafile format don't support rotated text). Newer programs, e.g. Microsoft Word 2003 do not have these problems.

If the active window is a plot window, please note that it is possible to set the resolution for high resolution export to clipboard (or printing). The variable is called `Advanced.ClipboardResolution` with default 600 dpi.

This command is only available in the Windows version of Dymola.

Tools > [Export] Image

Saves a PNG or SVG image of the contents of the active window (without window borders and plot bar, if any). The image is identical to the image shown in the window, so the size and representation can be changed by first resizing the window.

Exported images are included in the command log.

To obtain high resolution images from animation, the flag `Advanced.ExportAnimationWidth` can be set to the wanted width for the exported bitmap image (an example is `Advanced.ExportAnimationWidth=750;`) before animating. The flag is by default 0, meaning that the flag is not used. This flag is not stored between sessions.

It is possible to create an image with transparent background by setting `Advanced.TransparentImageBackground=true.`
This flag will give you objects which are separated from the background; with the same color as on the screen (only the background is transparent).

Setting `Advanced.TransparentImageObjects=true` allows export of image with transparent objects, not only transparent background. Transparent objects will be exported as transparent in the image. The possible drawback is that colors can be changed depending on the background when the image is inserted into a document.

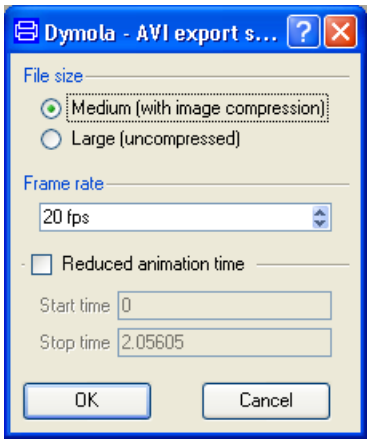
Tools > [Export] Animation

Saves the current animation window in any of the formats:

- AVI (Audio-Video Interleaved)
- animated GIF
- VRML format (.wrl).
- X3D as pure XML (.x3d).
- X3D as XHTML (HTML/JS) (.xhtml). When exporting to this file format, an external library, X3DOM, is used.

To export an animation in AVI format, the user must select medium file size, which may incur some loss of image quality due to compression, or large file size without loss of quality. It is also possible to change the frame rate, i.e., the number of image frames per second. By selecting **Reduced animation time** it is possible to export just a part of the total animation.

AVI export setup.



Tools > [Export] HTML

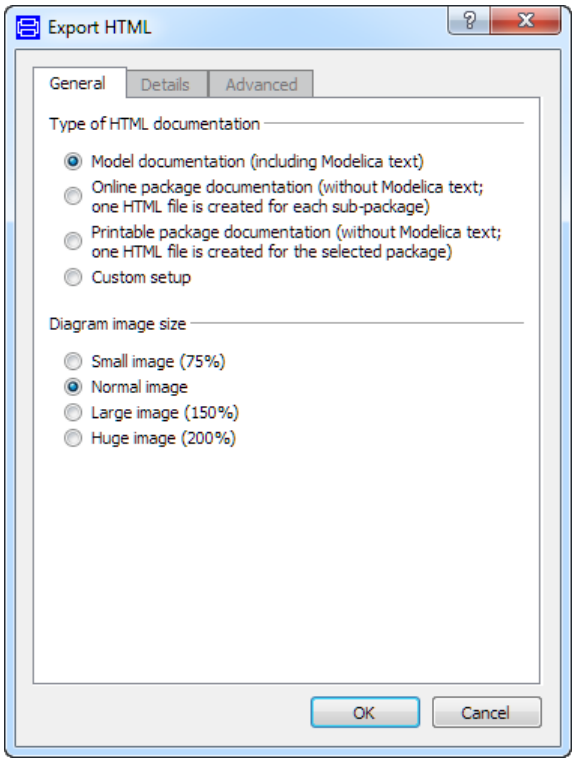
Exports comprehensive model and library documentation in standardized HTML format according to selected options and a CSS (Cascading Style Sheet) file. The generated HTML code is controlled with regards to documentation contents, references to other files and the size of graphics.

The CSS file is located in `Program Files\Dymola 2023\insert` and has the name `style1.css`. For more information, see Working with Cascading Style Sheets (CSS) on page 320.

The command is applied on the selected package/model.

General tab

General HTML options.



The **Type of HTML documentation** group offers three convenient default settings corresponding to typical customer demands, plus the possibility to customize the settings.

Model documentation.

Model documentation (including Modelica text) is used to create detailed online documentation that should be an integrated part of Dymola.

This selection will by default generate HTML documentation and associated bitmaps in the `help` subdirectory of the top-level model or package. A separate HTML file is also created for each sub package. Dymola assumes this structure for references to other classes, creating links to the `help` subdirectories of each package.

Modelica text is included; equations etc. are included in the HTML files.

This setting generates a full set of links to referenced classes.

Important! The location of the generated files is controlled by the flag

```
Advanced.HTMLResourceshelpDymola
```

The flag is by default `false`, meaning that the above `help` subdirectory is the location of the generated files and bitmaps. However, setting this flag to `true` will generate the files and bitmaps in `Resources\helpDymola` subdirectory instead when giving any corresponding documentation command. Note that if both the `Resources\helpDymola` and the `help`

subdirectories are present, the `Resources\helpDymola` subdirectory is given preference to the `help` subdirectory.

Online package documentation.

Online package documentation (without Modelica text; one HTML file is created for each sub-package) will by default generate HTML documentation and associated bitmaps

The setting is intended for creation of online documentation for model libraries. For location of generated files, see the above command.

Printable package documentation.

Printable package documentation (without Modelica text; one HTML file is created for the selected package) should be used when the issue is to create one file for generation of e.g. a pdf file for later distribution/printing.

One HTML file without any links to other files is generated.

The user will be asked for the file name and location for the HTML file. Associated bitmaps will automatically be created in the same location. Present restriction in picture references assumes that the location selected is the `help` folder (or `Resources\helpDymola` folder, see note above) corresponding to the package documented.

A Microsoft Word 2003 template for generation of printed library documentation is available in Program Files\Dymola 2023\Documentation\Documentation template.doc. Index entries for class declarations are generated according to Microsoft Word HTML format.

If needed, the setting can be modified by selecting **Custom setup** (after first having selected **Printable package documentation** to get the corresponding default setup).

Custom setup.

Custom setup allows the user to manipulate the default settings. Any combination is possible to select. However, the idea is primarily to modify a previous selection of HTML document type, that is, to first select the choice that should be modified, e.g. **Printable package documentation**, then selecting **Custom setup** to modify the corresponding default setup. When Custom setup is selected, two additional tabs (described below) are available.

Please note that whenever any of the three other HTML documentation type alternatives is selected, the underlying default setting is reset to the corresponding default, and any custom setting is lost.

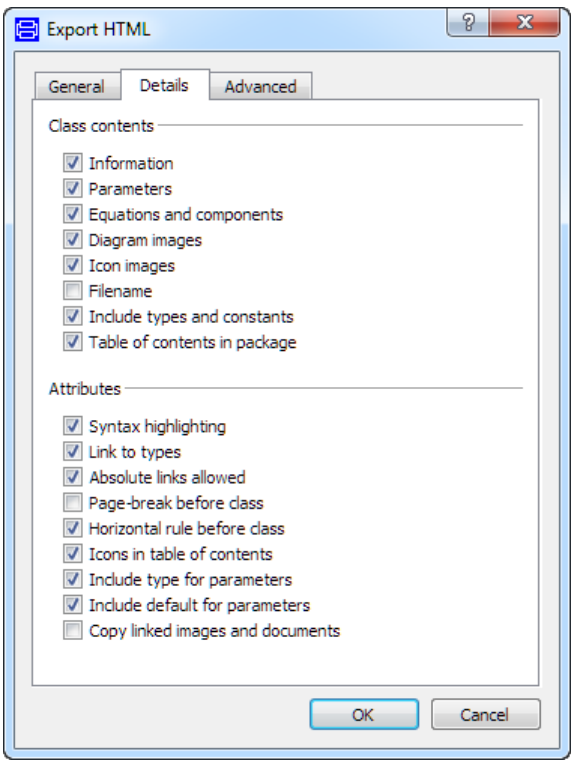
The **Diagram image size** group handles the fact that Bitmap images are not scalable and the best size depends on several factors, e.g., the number of components, complexity of graphics and the intended use of the documentation. Dymola uses a combination of heuristic sizing and user control.



When clicking the **Export** button, the files are generated. A message in the command log and in the status bar of the Dymola main window will show where the files are created.

Details tab

Detailed HTML setup.



This tab is only available when **Custom setup** has been selected in the General tab.

The initial selections in this tab depend on what HTML type of documentation that has been selected in the General tab before **Custom setup** was selected. The example above illustrates the default of **Model documentation**.

The **Class contents** group controls what kind of information is included in the HTML file for each class. This makes it possible to reduce the size of the documentation and hide certain parts, such as the equations. The following selections are possible:

- **Information** The documentation layer is included if this entry is checked.
- **Parameters** Parameters of classes included if this entry checked. The parameter group is a tabular description of all parameters, including parameters inherited from base classes.
- **Equations and components** Modelica Text content (code etc) is included if this entry is checked.
- **Diagram images** Picture of diagram is included if this entry is checked.
- **Icon images** Picture of icon is included if this entry is checked.
- **Filename** The name of the file where the class is stored is included if this entry is checked.
- **Include types and constants** Types and constants is included if this entry is checked.

- **Table of contents in package** Table of contents is generated with the description of the package if this entry is checked.

The Attributes group specifies the attributes of the document. The following selections are possible:

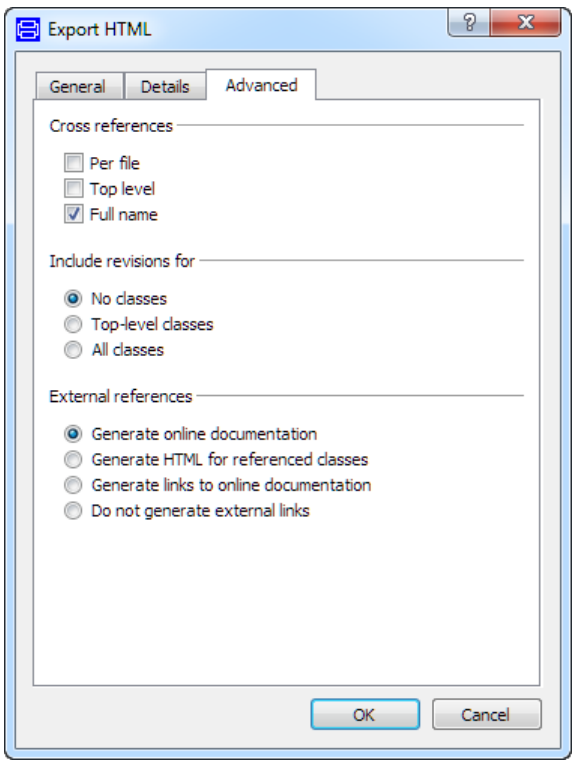
- **Syntax highlighting** Keywords will be highlighted with colors if this entry is checked.
- **Link to types** Links will be generated to types in e.g. Modelica.Units.SI if this entry is checked.
- **Absolute links allowed** If this entry is checked, absolute file-links is used when necessary. This entry should not be checked when generating documentation for customers etc.
- **Page-break before class** A page-break is generated before each class in the HTML documentation if this entry is checked.
- **Horizontal rule before class** A horizontal rule is generated before each class in the HTML documentation is this entry is checked.
- **Icons in table of contents** Icons is generated before the class name in the table of content if this entry is checked.
- **Include type for parameters** A column of types is included in the parameter table if this entry is checked.
- **Include default for parameters** A column of default values is included in the parameter table if this entry is checked.
- **Copy linked images and documents** If this entry is checked, images and documents that are linked to from the documentation layer are copied to the destination folder of the created documentation. Note that only images and documents located inside the package structure (typically in the `Resources` folder) can be copied this way. Examples of typical links in the documentation layer that can be handled by this option are:

  ```
  Modelica://MyLib/Resources/Images/Illustration2.png
  Modelica://MyLib/Resources/Documents/Document1.pdf
  ```

  The linked images and documents are copied to the target folder of the HTML export, typically a folder `help` located at the same level as the `Resources` folder.

Advanced tab

Advanced HTML setup.



This tab is only available when **Custom setup** has been selected in the General tab.

The initial selections in this tab depend on what HTML type of documentation that has been selected in the General tab before **Custom setup** was selected. The example above illustrates the default of **Printable package documentation**.

The **Cross references** group gives possibility to include cross-references in the HTML documentation. The following choices are possible:

- **Per file** Cross-references to classes is generated in the HTML documentation in each HTML file if this entry is checked.
- **Top level** Cross-references to classes is generated in the HTML documentation for the top-level package if this entry is checked.
- **Full name** Cross-references to classes using full names is generated if this entry is checked. This entry should not be checked when checking consistency of referencing to classes.

The **Include revisions for** group makes it possible to select for what classes revision information should be included. More information about revision information is given in the end of section “Using the documentation editor” starting at page 309. Alternatives are:

- No classes

- Top-level classes
- All classes

The **External references** group controls what kind of HTML documentation is generated. Four different types of HTML documentation can be generated. They differ in terms of what classes are documented and how links to other files are created. The following entries are displayed:

Generate online documentation.

Generate online documentation should be checked when ordinary online documentation should be created. (This is the default setting.)

HTML documentation and associated bitmaps are created in the `help` or the `Resources\helpDymola` subdirectory of the top-level model or package. A separate HTML file is also created for each sub package. Dymola assumes this structure for references to other classes, creating links to the `help` or Resources\helpDymola subdirectories of each package. (Whether the `help` or the `Resources\helpDymola` subdirectory is used is controlled by a flag, see the command **Model documentation (including Modelica text)** above.)

This setting generates complete HTML with a full set of links, without the need to create duplicate files. Model changes require only local regeneration of HTML, but all related models should be consistently documented in this way.

Generate HTML for referenced classes.

Generate HTML for referenced classes will also include documentation of other classes locally if checked. This is intended for web publication.

Complete HTML is generated for the model and all referenced classes. The user will be asked for the filename of the top-level HTML file; all other generated files will be saved in the same directory.

This setting creates a directory with a complete set of files and no links to other directories, thus ensuring maximum portability of the produced HTML. Such a directory can easily be moved to a web-server, for example. The main drawbacks are the storage overhead, and the need to regenerate when any referenced model has been changed.

Generate links to online documentation.

Generate links to online documentation is checked if one wants to create internal documentation where references should be made to existing online documentation.

Generates HTML in the file specified by the user and links to other libraries if the HTML files exist. This mode is similar to "Generate online documentation" except that the HTML file is not automatically stored in the `help` or `Resources\helpDymola` directory. It can for example be used to document a model in a library during development, before creating the final library documentation.

HTML without any links to other files is generated. This setting provides pretty-printed documentation for a single model or package only.

Do not generate external links.

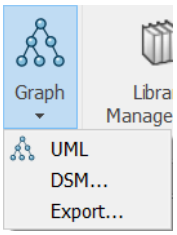
Do not generate external links should be checked if e.g. the aim is to create a free-standing pdf document from the html documentation from Dymola. If this alternative is checked, only one html file is produced.

Tools > [Export] Graph

This command supports the visualization of class hierarchies in three different formats:

- UML diagrams, using an online service called YUML.
- Design Structure Matrix (DSM), see https://en.wikipedia.org/wiki/Design_Structure_Matrix.
- Saving dependency information in SysML v2 format.

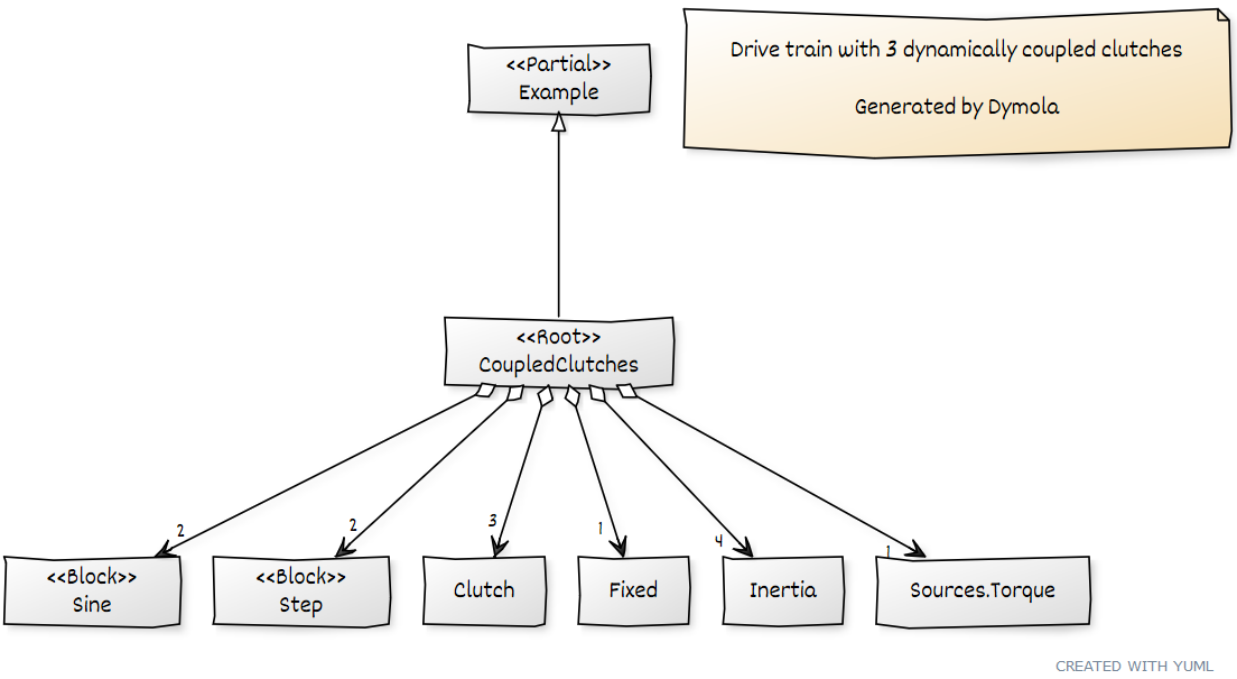
The corresponding commands are:



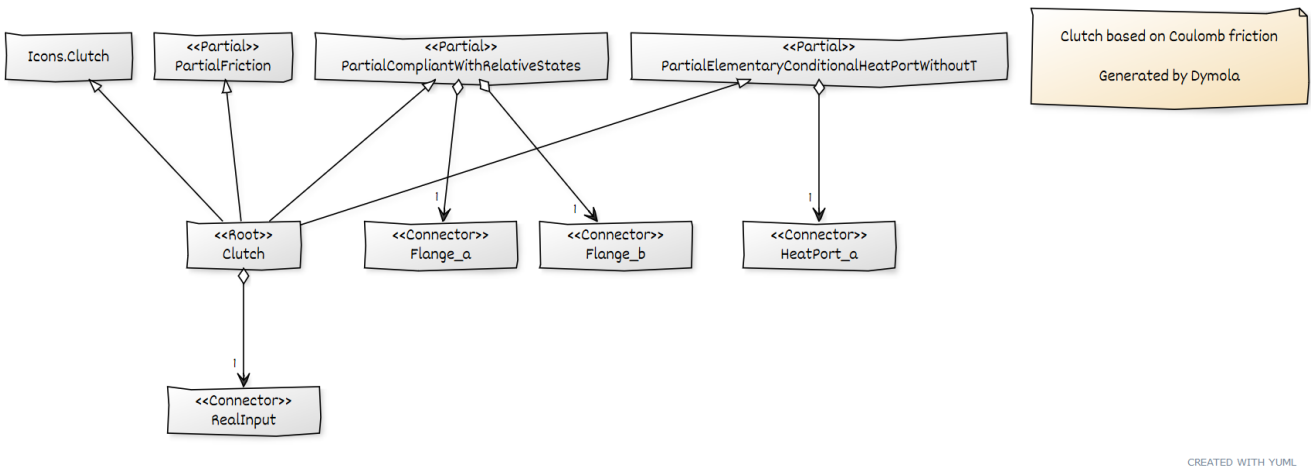
Tools > [Export] Graph > UML

This is the default command, that is, if **Tools > [Export] Graph** is clicked without any selections, that will be the command performed.

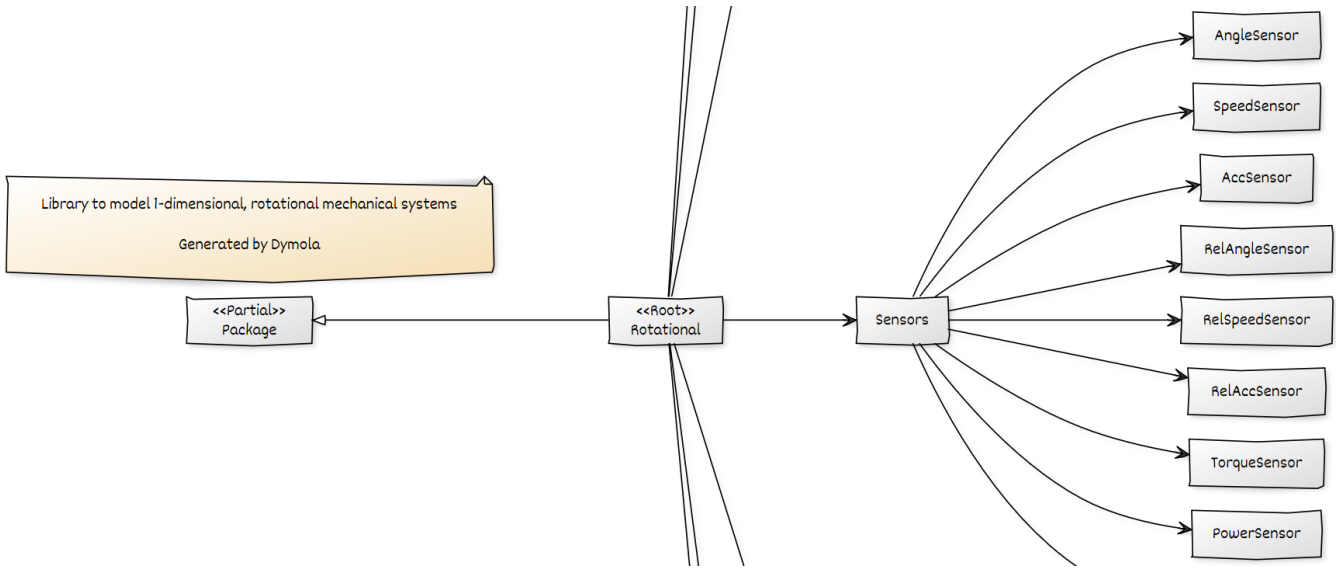
Exporting as UML creates a graphical representation using an online service called YUML, and displays the result in your web browser. For models, the diagram shows inheritance and components (the aggregation of component types and the cardinality of each component type). Dependencies to other Modlica libraries are shown in the description box.



A more complex example with inherited connectors:



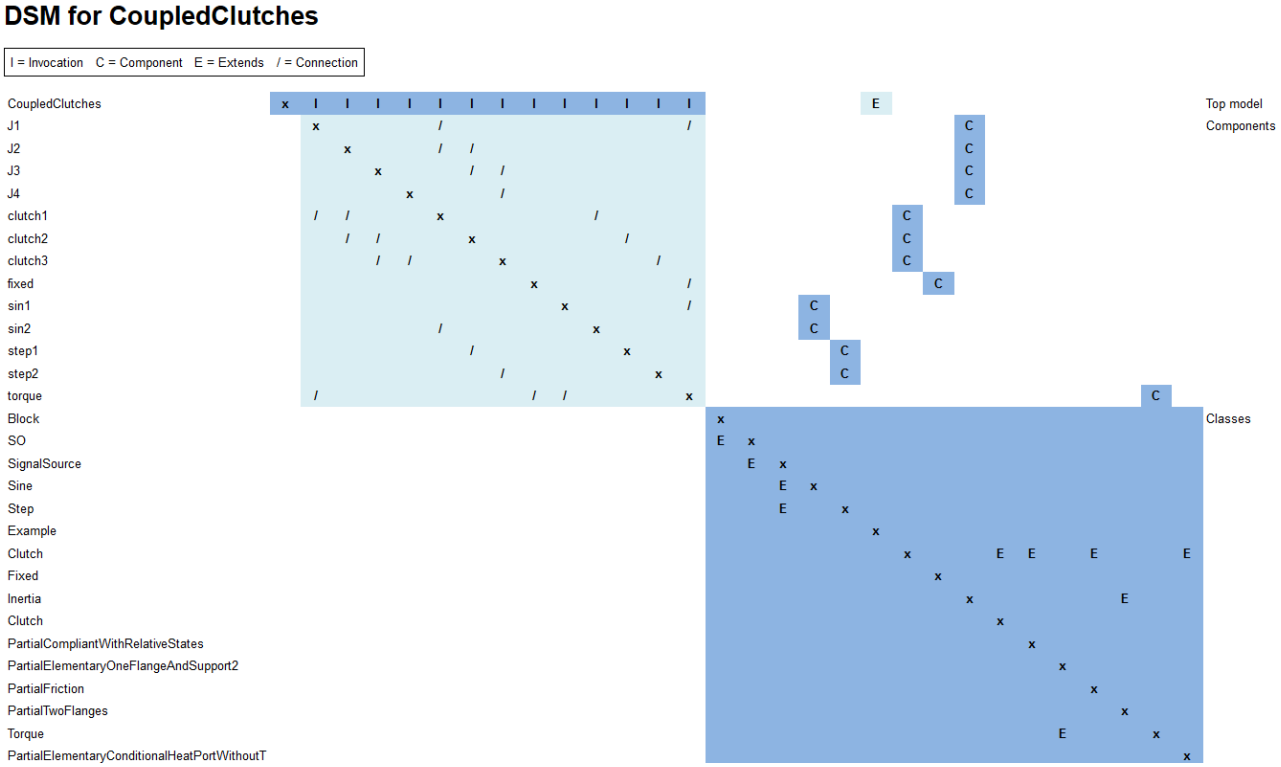
If the current class is a package, a different type of view is displayed, showing only the package hierarchy:



It should be noted that some UML graphs could become too complex to be sent to an online service as an URL, in which case the display will fail. One such example is the entire Modelica Standard Library.

Tools > [Export] Graph > DSM…

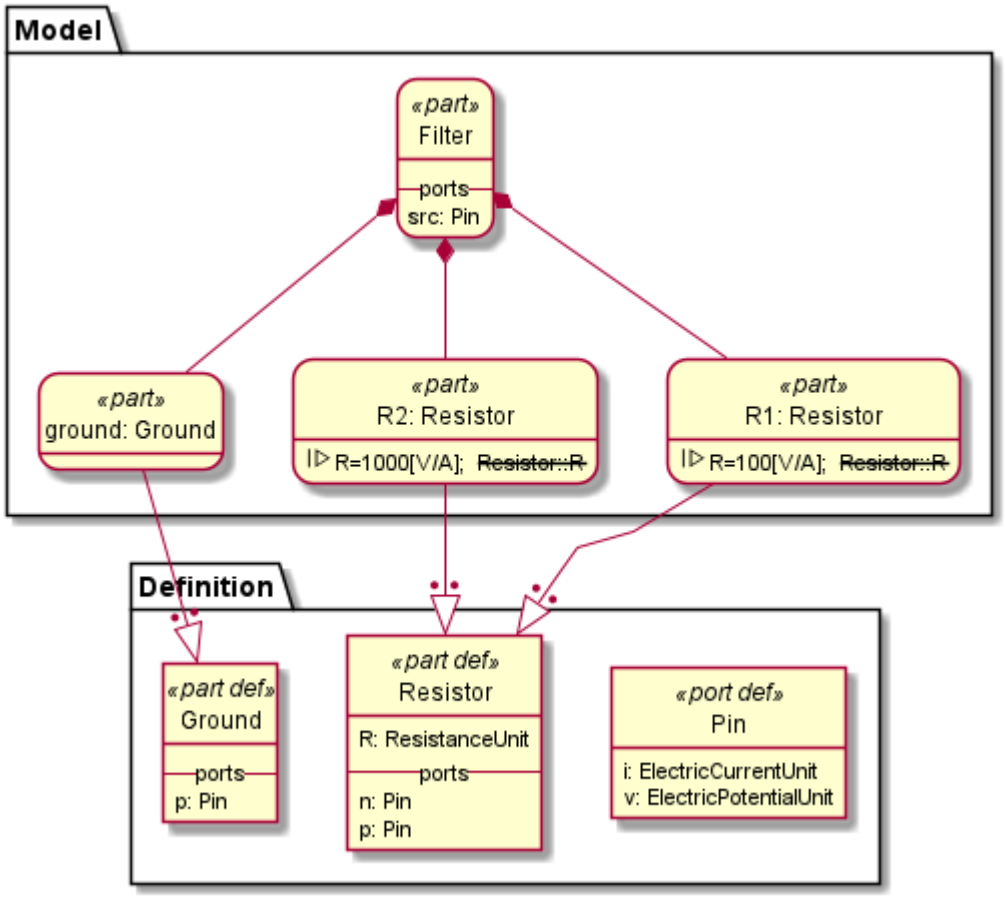
Saves a Design Structure Matrix (DSM) on file and displays it in your web browser. Symbols in the DSM are active and will display extended information as a tooltip. For example, if you hover the cursor over the name of the model in the left column, the full path is shown.



When applying this command, you first have to enter an .html file to export to. The file is created when you apply the command.

Tools > [Export] Graph > Export…

The command saves dependency information in SysML v2 format, for use in other tools. Here is an example:



When applying this command, you first have to enter a .sysml file to export to. The file is created when you apply the command.

Tools > Library Management

The command opens a dialog with three tabs:

- A tab for managing the **File > Libraries** menu for installed libraries (what libraries to show, and in what order etc.).
- A tab where you can manage the environment variable MODELICAPATH from inside Dymola. This environment variable specifies a list of directories where packages will be searched for when for example opening a package.
- A tab for on-demand installation of downloaded libraries. The tab enables you to select a mo/moe file or a zip file and install from this source. You can use partial installation if the zip file contains several libraries. You can select destination of the installation, as well as

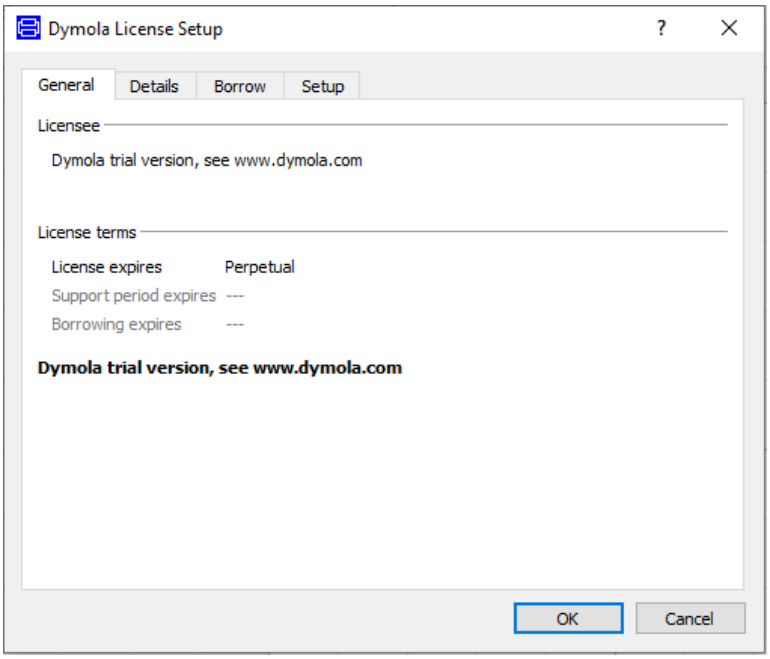
if the libraries should replace existing ones, and if they should be added in the **File > Libraries** menu.

For more information about this command, see the chapter “Appendix – Installation”, section “Installation on Windows”, subsection “Additional setup”.

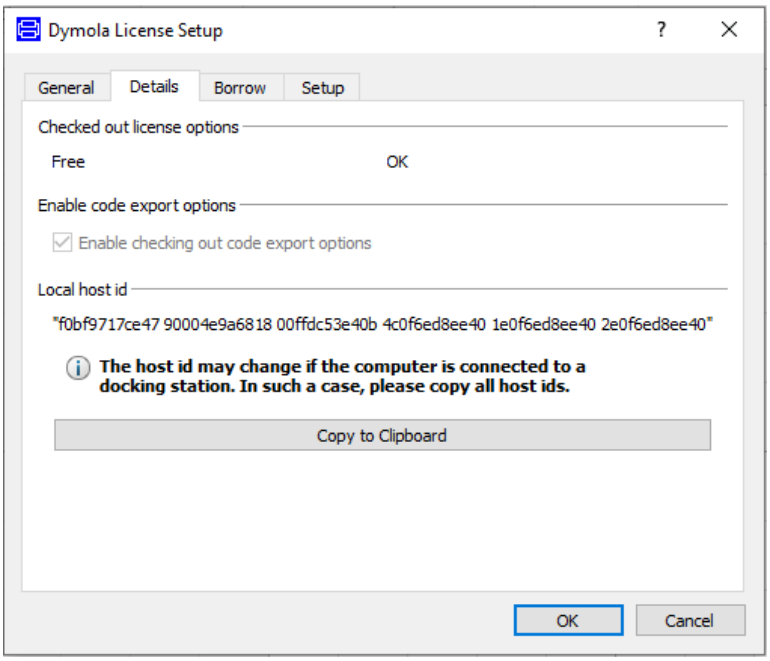
Tools > License Setup

The license handling is presented in the chapter “Appendix – Installation”. Please see this chapter for more information. Below is presented how the tabs would look like with the trial version of Dymola installed.

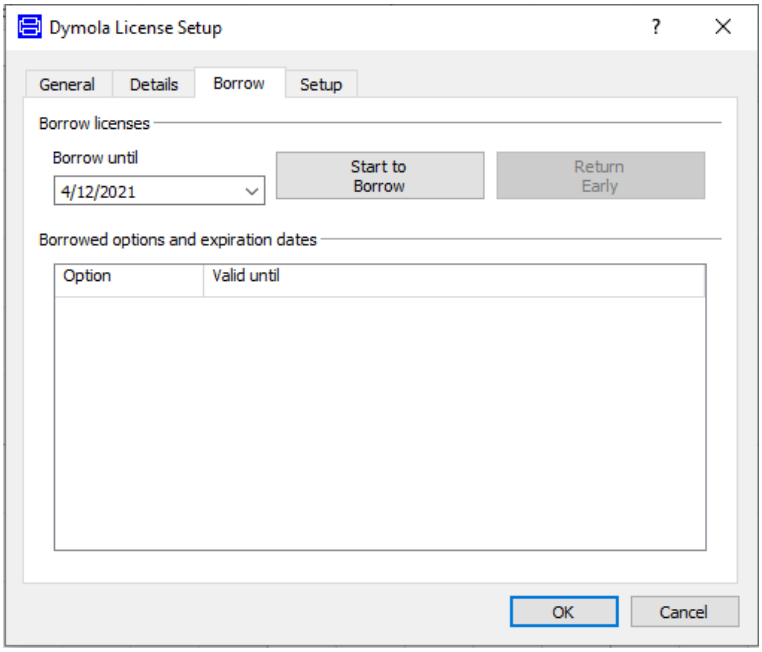
General tab



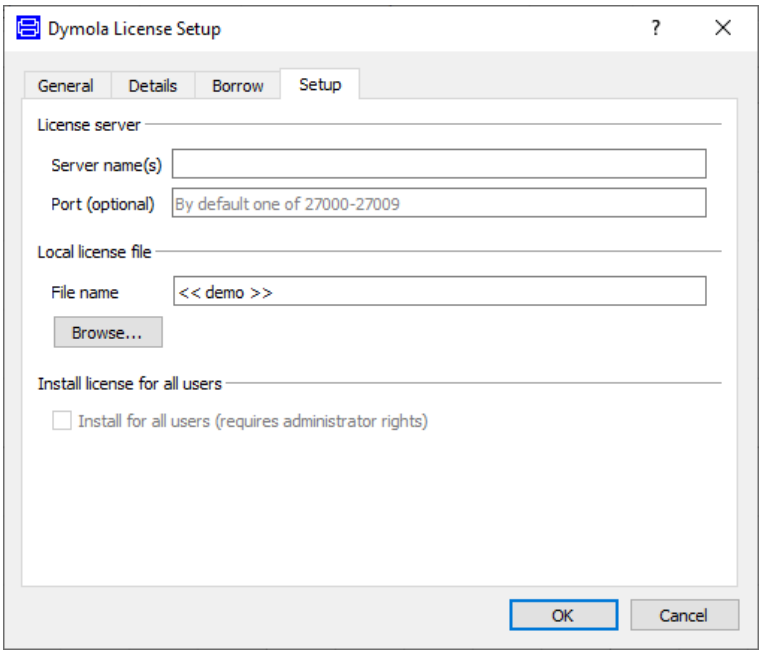
Details tab



Borrow tab



Setup tab

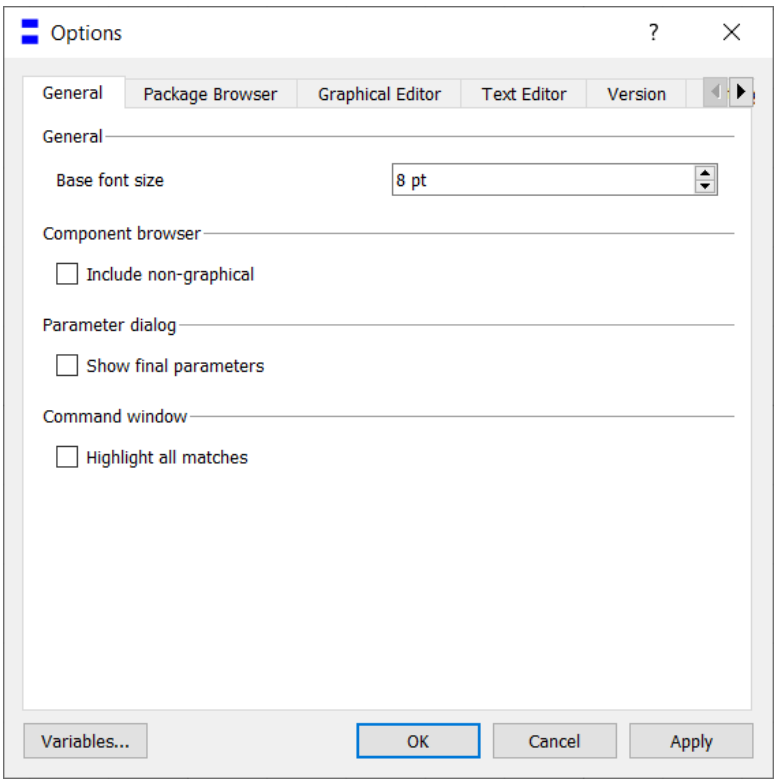

Dymola License Setup
General
Details
Borrow
Setup
License server
Server name(s)
Port (optional)
By default one of 27000-27009
Local license file
File name
<< demo >>
Browse...
Install license for all users
Install for all users (requires administrator rights)
OK
Cancel

Tools > Options

This command is also available as **File > Options**.

General options tab

General user interface settings.



The **General** tab contains graphical user interface settings for windows and the component browser.

General group settings:

Base font size Dymola by default uses the system font size for text in menus, Modelica text etc. If this size is too small when for example presenting Dymola for an audience, the base font size can be changed.

On some computers, it can be difficult to see the difference between the digit “1” (one) and the letter “l” (lower case L). Increasing the font size often helps, but if needed the font used for Modelica text can be changed by a command line option; which font is suitable depends on the computer and which fonts have been installed. Example:

```
"C:\Program Files\Dymola 2023\bin\Dymola.exe" -fixedfont
"Lucida Console"
```

Component browser group setting:

Include non-graphical will, if enabled, display components without graphical elements in the component browser. Such elements are, for example, variables and constants. Please note that only such components in the top level of the tree are shown. This setting can also be applied by a button in the component browser. See “Including top-level non-graphical components in the component browser” on page 213.

Parameter dialog group settings:

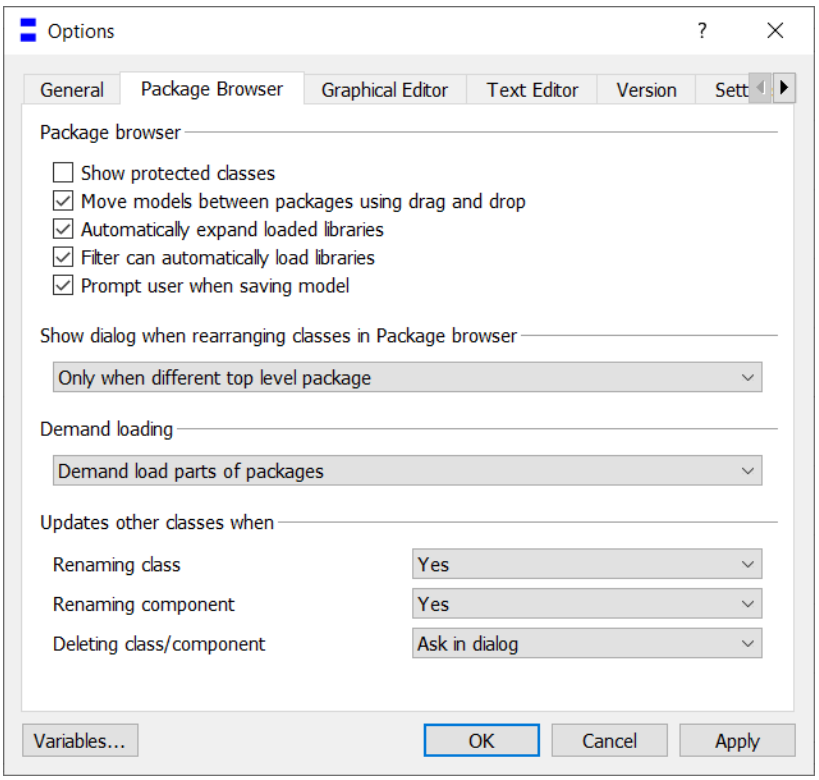
Show final parameters will, if enabled, show final (read-only) parameters in the parameter dialog.

Command Window group settings:

Highlight all matches highlight all matches of whole words when using the **Find** or **Find and Replace** commands, for the Command Window. Note that other graphical editors have another setting with the same name; see the Modelica text editor group settings above.

Package Browser options tab

Package browser user interface settings.



The **Package Browser** tab contains graphical user interface settings mainly related to the package browser.

Package Browser group settings:

The following options affects the package browser.

Show protected classes. Protected classes are by default not included in package browser. This can be changed by ticking the checkbox. For encrypted and read-only packages, protected classes are never shown regardless of the setting of this option.

Reorder using drag and drop. Packages can be reordered using drag and drop in the package browser, for information about this feature see section "Drag and drop of classes in the package browser" on page 234. This setting corresponds to the flag `Advanced.DragDropPackageBrowser = true;`.

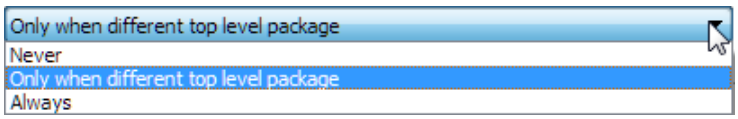
Automatically expand loaded libraries is by default set, meaning that the top level of libraries are automatically expanded when loaded using the **File > Libraries** command. If the option is unticked, the library is loaded, but the top level is not expanded (like e.g. the Modelica Standard Library when starting Dymola). The setting corresponds to the flag `Advanced.ExpandLoadedPackage = true;`.

Filter can automatically load libraries is by default set, meaning that libraries can be automatically loaded when the filtering function in the package browser is used and matches are found in unloaded libraries. See "Filtering in the package browser" starting on page 208. The default setting corresponds to the flag `Advanced.PackageBrowserFilterLoadLibraries=true`. The setting is saved between sessions.

Prompt user when saving model is by default set, meaning that the user will get prompted when saving a model by the command **File > Save > Save**. See "File > Save > Save" on page 455.

Show dialog when rearranging classes in Package browser group setting:

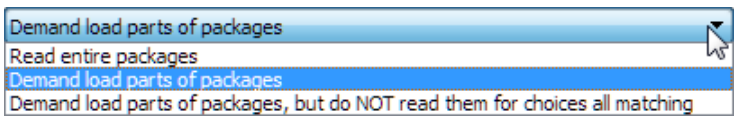
This setting controls when the **Move**/**Copy** dialog is displayed when moving or copying a class in the package browser, either by context commands or by drag and drop. The alternatives are:



The second alternative is the default one. This alternative is to only display the **Move** or **Copy** dialog when moving or copying to another top level package.

Demand loading group setting:

This setting controls how the packages are loaded in Dymola. The alternatives are:



Read entire packages was the default setting for Dymola 2017 and earlier. It corresponds to the flag `Advanced.DemandLoadClasses=0;`.

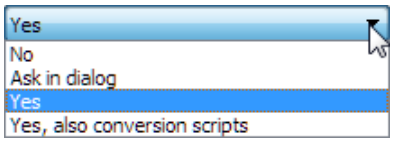
Demand load parts of packages is the default setting for Dymola 2017 FD01 and later. It corresponds to the flag `Advanced.DemandLoadClasses=1;`. In most cases the user will see improved speed, but in some cases delays occur where delays did not occur previously, typically there is now a delay when opening the first dialog in a library (for example Modelica Standard Library) having the setting **Add selector for all matching choices** activated. This setting leads to opening the entire corresponding library, which takes some time.

Demand load parts of packages, but do NOT read them for choices all matching avoids any delay corresponding to the setting **Add selector for all matching choices** but when doing for example a redeclare operation, only classes previously used are displayed as alternatives. This selection corresponds to the flag `Advanced.DemandLoadClasses=2;`.

Update other classes when group settings:

The settings in this group control the smart rename and smart delete features. Note that changing of any of these settings does not require a new translation of the model.

Renaming class This setting controls how other classes are updated when renaming a class. (For more about renaming classes in general, see section "Renaming a class" on page 234.) The alternatives are:



No means that other classes are not updated. This might result in later errors when trying to simulate the model. It corresponds to the flag `Advanced.ActivateSmartRenameClass=0;`.

Ask in dialog means that a dialog is displayed where you can select if you want to update other classes or not when you rename a class. It corresponds to the flag `Advanced.ActivateSmartRenameClass=1;`. Note that you can change the setting to **Yes** for future renaming of classes in the dialog by ticking **Always update without confirmation** and then click **Update**. For an example of a dialog (in this case for deleting a class but the dialog is in principal the same), see section "Deleting parameters, variables, and constants" on page 383.

Yes means that other classes are automatically updated without displaying any dialog. This is the default. It corresponds to the flag `Advanced.ActivateSmartRenameClass=2;`.

Yes, also in conversion scripts means that other classes are automatically updated without displaying any dialog, but also that the name changing operations are accumulated in a conversion-script, to enable later conversion of libraries using the library that were not loaded during the conversion. It corresponds to the flag `Advanced.ActivateSmartRenameClass=3;`.

Renaming component This setting controls how other classes are updated when renaming a component. (For information on how to rename components, see section "Editing name and comment" on page 235.) The alternatives are the same as for renaming classes above, except that the flag is `Advanced.ActivateSmartRename`.

Deleting class/component This setting controls how other classes are updated when deleting a class or a component. (For more about deleting classes in general, see section "Deleting parameters, variables, and constants" on page 383.) The alternatives are:



No means that other classes are not updated. This might result in later errors when trying to simulate the model. It corresponds to the flag `Advanced.ActivateDelete=0;`.

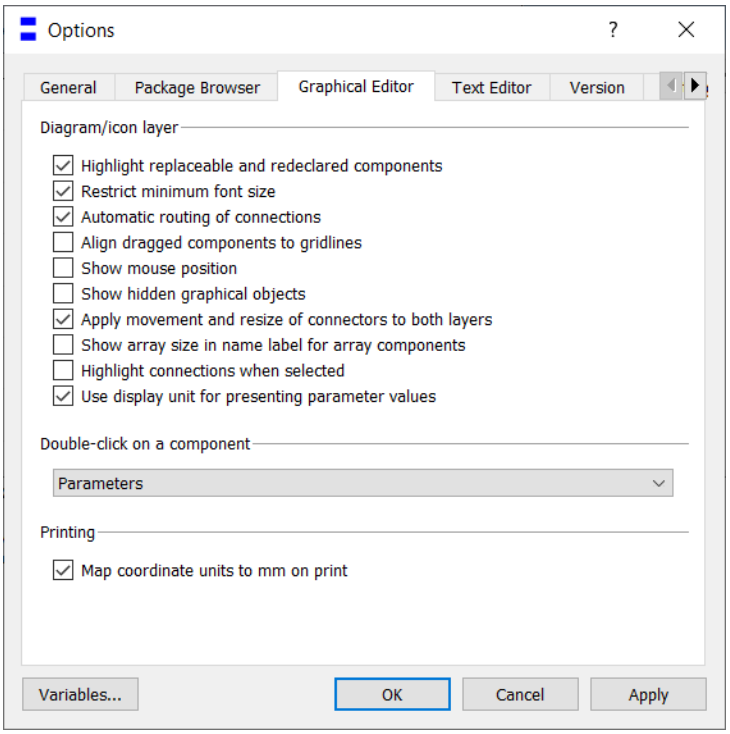
Ask in dialog means that a dialog is displayed where you can select if you want to update other classes or not when you rename a class. This is the default. It corresponds to the flag `Advanced.ActivateSmartDelete=1;`. Note that you can change the setting to **Yes** for future renaming of classes in the dialog by ticking **Always update without confirmation** and then click **Update**. For an example of a dialog and an example of deleting a class, see section "Deleting parameters, variables, and constants" on page 383. See the same section for a list of limitations.

Yes means that other classes are automatically updated, if possible, without displaying any dialog. It corresponds to the flag `Advanced.ActivateSmartDelete=2;`.

The button **Variables...** displays a dialog for displaying and setting setting Boolean, integer, real, and string flags. See "General GUI for displaying and setting flags" starting on page 431.

Graphical Editor options tab

Graphical editor user interface settings.



The **Graphical Editor** tab contains a number of options that affect the operation of the graphical editor user interface.

Diagram/icon layer group settings:

Highlight replaceable and redeclared components Components and connectors that can be replaced are highlighted if this entry is checked. Replaceable components with the default value are shown sunken, as a socket. Redeclared components are shown raised, as mounted in the socket. Components of a replaceable class are outlined with a dashed line. Note that highlighting inherited components are part of the filtering, see section "Filtering on inherited components and connectors" on page 226. For more about indications in the diagram, including also indication of replaceable *classes*, see "Presentation of components and connectors in the diagram layer" starting on page 363.

Restrict minimum font size to make small texts more easily readable. As a consequence, these texts may overflow their bounding boxes. By default, the smallest text size will be 8 pt when this setting is active.

Automatic routing of connections If enabled, connections are automatically routed when created, or when the corresponding compmonent is moved. For more about automatic routing, see section "Routing of connections" on page 252.

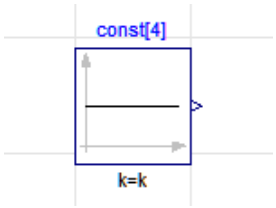
Align dragged components to gridlines If enabled, components are aligned to gridlines when dragged from the browser into the graphical editor (diagram layer of an edit window).

Show mouse position will display the cursor position in the status bar, if the cursor is inside the Edit window.

Show hidden graphical objects will, if enabled, show hidden graphical objects (e.g. hidden graphical connections).

Apply movement of connectors to both layers will, if enabled; result in that movement of connectors will be applied to both the icon layer and the diagram layer if the position of the connectors is initially the same in both layers.

Show array size in name label for array components will, if enabled, display the dimension in brackets [xxxx] for vectorized components. Below the component const vectorized to contain 4 components:



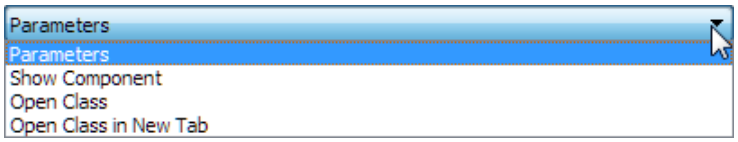
The setting is by default not activated. This means that only empty brackets [ ] are displayed for vectorized components. Note however that you can always use the tooltip to display the full dimension.

Highlight connection when selected If enabled, clicking a connection will highlight it. For more information, see “Highlighting connections” on page 254.

Use display units for presenting parameter values If enabled, display units are shown for values in the diagram layer. By default this setting is activated. The setting corresponds to the flag `Advanced.UseDisplayUnitInDiagram=true`.

Double-click on a component group settings:

Action when double-clicking a component can be selected:



Parameters means that the parameter dialog of the component is displayed at double-click. This is the default.

Show Component means that the command **Show Component** is executed at double-click on a component.

Open Class opens the class in the current tab.

Open Class in New Tab opens the class in a new tab.

The setting corresponds to the flag `Advanced.ComponentDoubleClickAction` with the values 0 – 3. The setting is saved between sessions.

Shift+double-click

Note! Given the above, **Shift+double-click** on a component in the diagram layer by default performs **Show Component**. However, if double-click has been redefined to perform some other action than opening the parameter dialog, **Shift+double-click** on the component will open the parameter dialog.

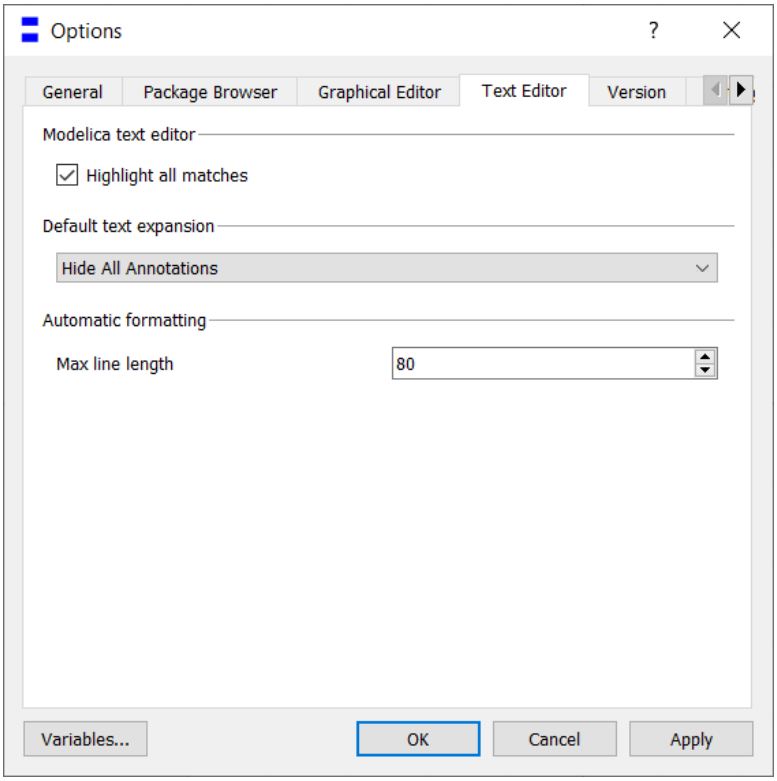
Printing group settings:

Map coordinate units to mm on print Coordinates will be mapped to physical size of one millimeter on paper when printing if this option is enabled. If not enabled, the coordinate system is fitted to the print area.

The button **Variables...** displays a dialog for displaying and setting setting Boolean, integer, real, and string flags. See "General GUI for displaying and setting flags" starting on page 431.

Text Editor options tab

Text editor graphical user interface settings.

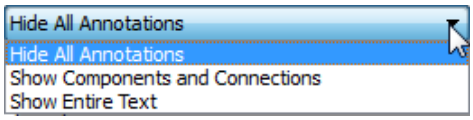


Modelica text editor group setting:

Highlight all matches highlight all matches of whole words when using the **Find** or **Find and Replace** commands. This applies for graphical editors except the command window that has a separate setting, see below, the Command Window group settings.

Default text expansion group:

The expand mode is selectable using a drop-down menu:



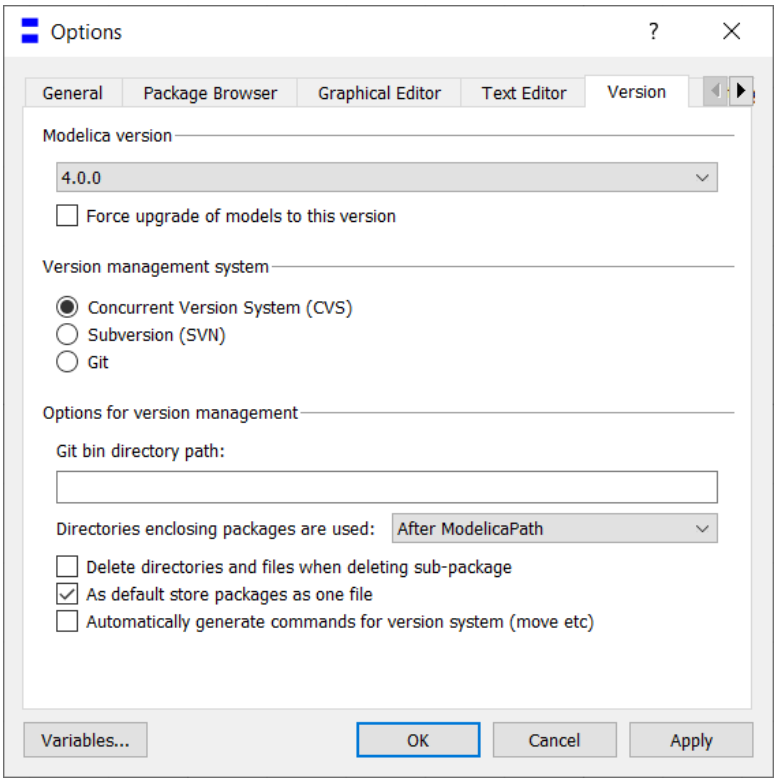
Automatic formatting length:

Max line length specifies the default maximum line length in the Modelica text editor. The default value is 80; values below 50 are not possible to set. The value corresponds to the flag `Advanced.MaxLineLength`. If you change this value, the corresponding dotted line in the Modelica text editor indicating the maximum line length is automatically adjusted to this value. Correspondingly, if you move the dotted line in the Modelica text editor, the **Max line length** value in the above menu will be updated with the new value. For information about how to change the max line length in the Modelica text editor window, see “Adjustable maximum line length” on page 285.

The button **Variables…** displays a dialog for displaying and setting Boolean, integer, real, and string flags. See “General GUI for displaying and setting flags” starting on page 431.

Version handling.

Version options tab



Modelica version group settings:

Sets the default version of the Modelica Standard Library and whether models should be upgraded to this version. This allows the user to quickly switch between some Modelica versions if more than one Modelica version is available/installed in that Dymola version.

If **Force upgrade of models to this version** is not set, the version of Modelica that the model uses is loaded on demand, if available. If the version is not available, Dymola suggest updating to the latest Modelica version available.

By setting **Force upgrade of models to this version**, models are required to be updated to the currently selected version of Modelica. If update to a certain Modelica version is wanted, please select that Modelica version and set **Force upgrade of models to this version** before opening the model. The current version is 4.0.0. If you need to work with an older model that has not been converted to MSL 4.0.0, and you don´t want to upgrade it, you can install MSL 3.2.3 libraries separately. See index entry "*Modelica Standard Library : older than the one in Dymola distribution*" in the manual.

Version management system group settings:

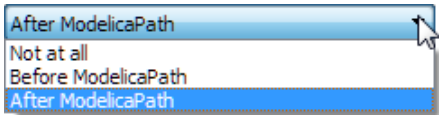
Dymola supports version control of models using three commonly used systems, Concurrent Version System (CVS), Subversion (SVN), and Git. Please see the chapter "Model Management", section "Version management".

Options for version management group settings:

Note that some of the options here can be valuable also if no version management system is used. For the use in version management, see the reference above.

Git bin directory path:

Directories enclosing packages are used can be used, except for version management, to make it easier to find multiple libraries that are stored in the same directory. The possible selections are:

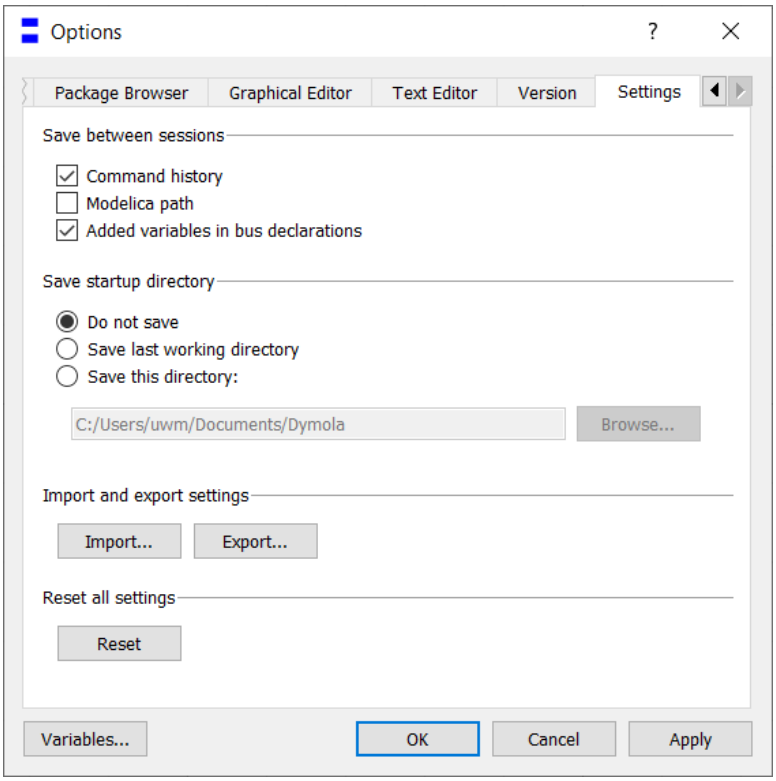


For more about this option, and ModelicaPath, please see the chapter "Appendix – Installation", section "Installation on Windows", subsection "Additional setup" – "Adding libraries and demos to the File menu". Note that the use of Modelica path is wider than the heading of this reference.

The button **Variables...** displays a dialog for displaying and setting setting Boolean, integer, real, and string flags. See "General GUI for displaying and setting flags" starting on page 431.

Settings options tab

Settings for saving.



Settings are saved in an XML file. The settings are saved, for example, when Dymola exits.

The following settings are always automatically saved:

- All settings in the **Tools > Options** dialog.
- Position and geometry of Dymola window. This includes position and size of Dymola window, position and location of command log, and browsers and toolbars inside the Dymola window.
- The search history.
- Custom colors.
- **Plot layout and setup**. This includes both global plot settings; e.g. whether sequence number should be presented in the legends, and the settings of individual plot windows (except what variables are shown). Plot layout and setup can also be saved, including variables shown, using the command **Simulation > Commands > Add Command** or the command **Script Editor > Generate Script** and the built-in function `saveSettings`. For more about these two commands and the difference between them, see next chapter. You can use the index entry "*simulation tab : add command*" in the index in the end of this manual to find the information.

The following settings can be selected to be saved or not by the user (the Save between sessions group settings and the startup directory in the Save startup directory group settings in the figure above)

Save between sessions group settings:

- **Command history**. Saves the command history of the command log if ticked. By default, this setting is active.
- **Modelica path**. Saves the changes in the system variable MODELICAPATH if ticked. By default, this setting is not active. Note that this setting can also be changed from the menu that is used handling the Modelica path. This menu is reached by the command **File > Library Management**, the **Modelica Path** tab.
- **Added variables in bus declarations**. Saves the variables that have been added to bus declarations when connecting expandable connectors. By default, this setting is enabled. If the setting is disabled, the variables added to bus declarations this way will not be present in the bus declaration dialogs when Dymola is restarted. (The added variables will however still be present in the bus locally for the components where they have been added.) For more information about adding variables in bus declarations this way, see section "Expandable connectors" starting on page 355.

Save startup directory group settings:

This group handles the startup directory, that is, the working directory used when starting Dymola again. For more on current working directory /startup directory in general, see section "More about current working directory, startup directory, and related commands" starting on page 462. The alternatives here are:

- **Do not save** No startup directory is saved. This is the default setting. The startup directory by default is `...\Documents\Dymola`.
- **Save last working directory** The last current directory when closing Dymola is saved as the startup directory.
- **Save this directory: <*path*>** The specified directory is saved as startup directory when closing Dymola. The directory can be browsed for, or typed in. Note that this setting corresponds to the command **File > Working Directory > Use as Startup Directory**. If this command is used, the setting described here is activated, and the directory in the input box is changed to the directory specified by the command. For more information about the command, see section "File > Working Directory > Use as Startup Directory" on page 464.

The setup information is by default stored in a system directory associated with the user. Each Dymola version (from 2020x) has its own file. On Windows, the name is typically, for Dymola 2023,

```
C:\Users\<user>\AppData\Roaming\DassaultSystemes\Dymola\2023\
setup.dymx
```

On Linux, the path to `setup.dymx` is, for Dymola 2023, under the user´s home directory:

```
.dassaultsystemes/dymola/2023/setup.dymx
```

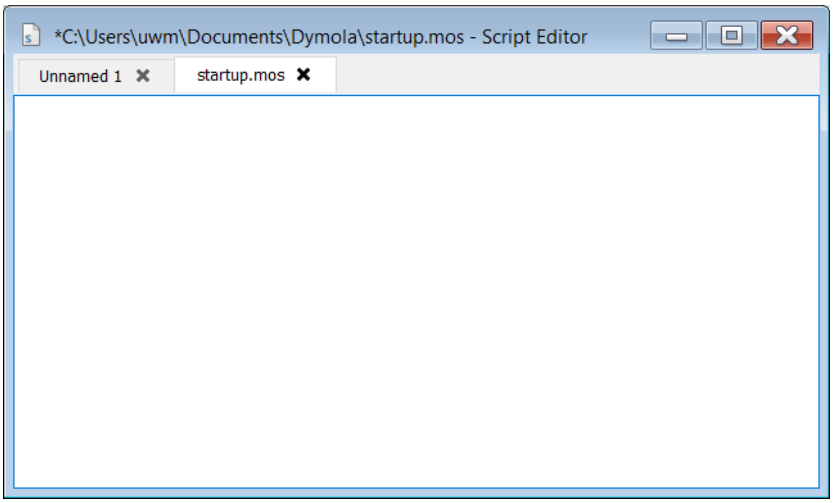
The directory is automatically created by Dymola if needed. If this setup file is not available due to using an older Dymola version, any previous `setup.dymx` or `setup.mos` file will be copied to a new `setup.dymx`.

It is possible to run Dymola with a custom .dymx setup file (for example `mysetup1.dymx`). This enables having different setups for different projects. You can in fact copy the whole Dymola installation to another location and use a custom setup file when starting the copy. This copy will be totally independent from the original, except for the licensing. For information how to manage custom setup files like this, see the index extry "*setup file : custom setup file*".

Important!

If you have added content in the `setup.mos` file, to for example open specific libraries, this content will not be saved in the new `setup.dymx` file. To handle such additions, you can create a customized setup file `startup.mos` by the command **Script Editor > Edit Startup** and add the wanted commands in that script file. (Note that to use that command, you must first open the **Script Editor** tab by, for example, creating an empty script by the command **Simulation > New Script**.)

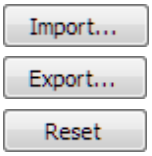
In Windows, the command above opens the file `startup.mos` in the directory `…Documents\Dymola`. If no such file exists, it is created when saving the entered text in the script editor. The script editor looks the following if no such file exists (the script `Unnamed1` is the result of creating a new script to enter the **Script Editor** tab):



If the file `startup.mos` exists, this file will be executed after executing `setup.dymx`. This means that if a setting is present in both files, the setting in `startup.mos` will be the one implemented.

Note. There is a specific possibility to have a personal setup file in .mos format; that is to create such a file and include the path to that file as a command line argument in the shortcut starting Dymola. This file is executed after the two files `setup.dymx` and `setup.mos`. See the index entry "*starting Dymola : startup script*", then look at the example "Creating shortcuts to Dymola" for more information.

(In Linux, the above about customize setup files is valid as well, except that the corresponding directory for the `startup.mos` file in Linux is `/home/<user>/Dymola`. If such a directory is not present when Dymola is installed, the directory is automatically created.)



The command buttons **Import…** and **Export…** makes it possible to import and export the content of the settings file. This makes it possible to, for example, store good settings for later restore.

The command button **Reset** resets all settings to “factory settings”.

In some cases changing of settings when running scripts is not wanted. To prevent saving of settings in `setup.dymx` when running a script, you can in the script set the flag:

```
Advanced.SaveSettings = false;
```

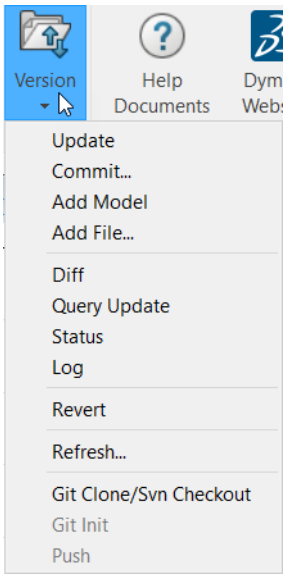
The flag is by default true, meaning that any change in settings is saved, independently if the change is made by command or by script. Do not forget to set the flag to true in the end of the script.

To prevent saving of settings in `setup.dymx` and `startup.mos` you can start Dymola with the command line option `-nosavesettings`. The settings are loaded, but any changes in the settings that you do in Dymola will not be saved.

To prevent *both* reading and saving of settings in `setup.dymx` and `startup.mos` you can start Dymola with the command line option `-nosettings`.

The button **Variables…** displays a dialog for displaying and setting setting Boolean, integer, real, and string flags. See “General GUI for displaying and setting flags” starting on page 431.

Tools > Version



For information about this command, please see the chapter “Model Management”, section “Version Management”.

Tools > Help Documents

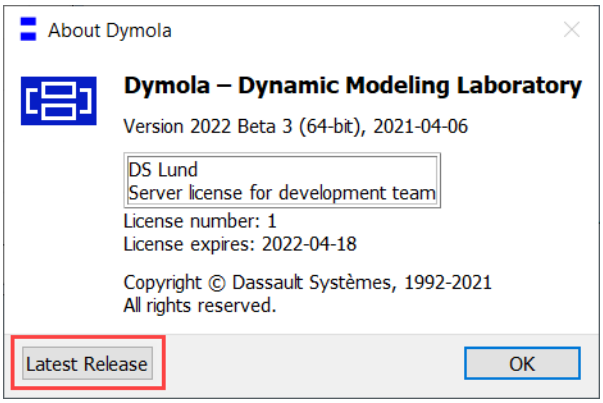
Opens a web browser window with the root of the online documentation, which contains the Dymola manuals, articles and links to other resources.

Tools > Dymola Website

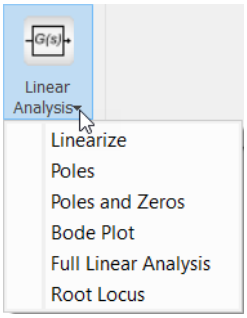
Opens a web browser window with the Dymola webpage.

Tools > About Dymola

Displays version, copyright and license information about Dymola. You can also get information about the latest Dymola release available, by clicking **Latest Release**:



Tools > Linear analysis



The Linear analysis menu contains shortcuts to certain functions in the library Modelica_LinearSystems2. Below the functions are described very short, for further information, please see the documentation of the library.

Note. To select parameters, use the **select** button in the **modelParam** tab in the dialog.

The alternatives are:

Linearize Linearizes a model and returns the linearized model as StateSpace object.

Poles Linearizes a model and plots the poles of the linearized model.

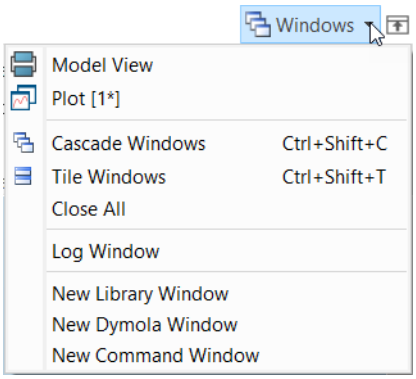
Poles and Zeros Linearizes a model and plots the poles and zeros of the linearized model.

Bode Plot Linearizes a model and plots the transfer functions from all inputs to all outputs of the linearized model as Bode plots.

Full Linear Analysis Linearizes a model and performs all available linear analysis operations.

Root Locus Computes and plots the root locus of one parameter of a model (= eigenvalues of the model that is linearized for every parameter value).

4.6.6 Main window: Window menu



The **Windows** menu button is located in the right upper corner of the Dymola window. The above figure is an example how the menu that appears when you click the arrow next to the command button may look like when working in the **Simulation** tab.

The first part of the commands displays the subwindows present in the Edit window, in this case the Model View window and a plot window. The one you click will be the active on. Note that by clicking directly on the **Windows** command (not the arrow next to it) you toggle between those window, which one should be the active window.

The other commands are:

Window > Cascade Windows



Arranges windows in the Edit window so they overlap.

Window > Tile Windows



Arranges windows in the Edit window so as no overlapping tiles.

Window > Close All

Closes all windows in the Edit window.

Window > Log Window

Displays the log window.

Window > New Library Window

Creates a new library window, see “Library window” on page 213.

Window > New Dymola Window

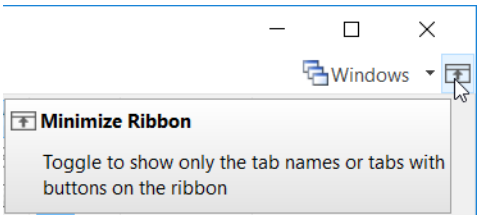
Creates a new modeling and simulation window.

Window > New Command Window

Creates a new command window.

4.6.7 Main window: Toggle button for ribbon information level

At the upper left corner of the Dymola window, the **Minimize Ribbon** button can be used to control what is displayed in the ribbons.

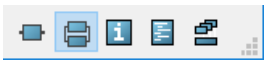


The button toggles between three levels of information:

- Show always tabs with buttons, and group names in the tab (default).
- Show always tabs with buttons, but hide group names.
- Show tabs with buttons only when tab is active (that is, when you click the tab name to give a command), but hide group names.

4.6.8 Main window: Model layer toolbar

In the lower right corner of the Dymola window model layer toolbar is present, to help you quickly select what model layer to work with:



The alternatives are (in bracket the tab where they also can be selected in detail, e.g. what mode of the documentation layer that should be visible/active):

Icon layer (the **Graphics** tab)

Diagram layer (the **Graphics** tab)

Documentation layer (the **Documentation** tab)

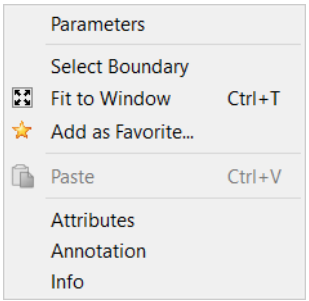
Modelica layer (the **Text** tab)

Used classes layer (the **Text** tab)

4.6.9 Context menu: Edit window (Icon layer)

The icon layer of the Edit window is introduced in “Icon layer” on page 191.

Right-clicking when nothing is selected in the icon layer of an edit window will display a context menu. Since no component/object is selected, the menu will encompass all present components/objects.



The following alternatives might be possible to select:

Parameters shows the parameters in the open layer, that is, the parameters of the package/component that the icon layer is a part of.

Select Boundary will select the coordinate system boundary. Please see section “Coordinate system” on page 197.

Fit to Window will display the whole model in the window. For more information about zooming, see section “Zooming” on page 217.

Add as Favorite… adds the open class to favorites. See the section “The Favorites packages” on page 227 for details.

Paste will paste the content in the clipboard, if any.

Attributes will display a dialog for component attributes. See the description of the command in “Context menu for components in the diagram layer and the component browser” starting on page 551.

Annotation makes it possible to view/edit the annotations of all objects present in the icon layer, and also model annotations. For more information about how to use this command, please see section “Working with annotations using the Annotations dialog” starting on page 426.

Info shows the information of the package/component that the diagram layer is a part of.

4.6.10 Context menu: Edit window (Diagram layer)

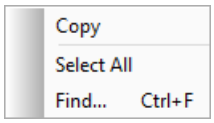
The diagram layer is introduced in “Diagram layer” on page 192.

Right-clicking when nothing is selected in the diagram layer of an edit window will display a context menu. Since no component/object is selected, the menu will encompass all present components/objects. The menu looks exactly as the one in the previous section and is analog to that.

4.6.11 Context menu: Edit window (Documentation layer)

The documentation layer is introduced in "Documentation layer" on page 193. More information, including how to edit the information, is available in section "Documentation" starting on page 300.

Context menu when viewing formatted documentation



Right-clicking in the documentation layer of the edit window presents a context menu with common text operations.

Copy copies text from the documentation layer to the clipboard. Note that when copying text from the documentation layer, the text is now only copied as plain text. The recommended way to copy text with HTML tags is to copy from the HTML source text. The copying can however be restored to the way it worked in previous versions (Dymola 2016 FD01 and older) by setting the flag `Advanced.CopyFormattedHtml=true;`.

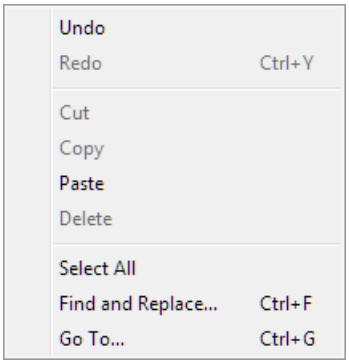
Select All selects all text in the documentation layer.

Find... finds text in the text editor. See the command **Find and Replace** in the section "Context menu: Edit window (Modelica Text layer)" starting on page 533. (Note that only the Find tab is relevant here.)

Context menu for editable info/revisions part

When viewing or editing the information or revision part of the documentation layer, either in the documentation editor or as HTML source code, a context menu is available by right-clicking. Most of the text operations available here are also available in the **Text** tab.

Context menu for editable info/revisions part of documentation layer.



Undo, Redo undoes and redoes the last editing operation in the editor. Same as **Text > Undo** and **Text > Redo**.

Cut, Copy, Paste copies text between the clipboard and the editor. Note that when copying text from the Info editor or Revisions editor, the text is now only copied as plain text. The recommended way to copy text with HTML tags is to copy from the source text (Info source/Revisions source, see below). The copying can however be restored to the way it

worked in previous versions (Dymola 2016 FD01 and older) by setting the flag `Advanced.CopyFormattedHtml=true;`.

Delete deletes selected text.

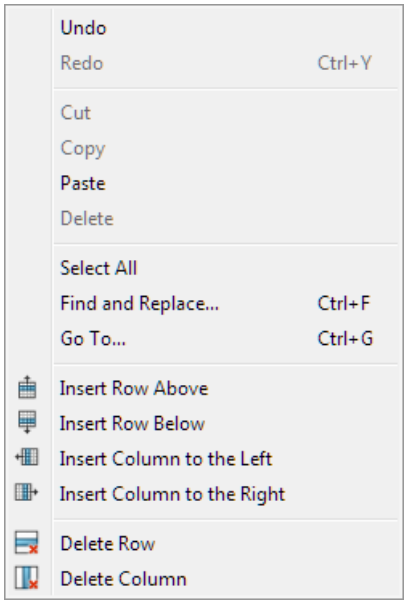
Select All selects all text in the documentation layer.

Find and Replace… finds and replaces text in the text editor. See the command **Find and Replace** in the section "Context menu: Edit window (Modelica Text layer)" starting on page 533.

Go To… sets the cursor at a specified line and scrolls the window if necessary.

Context menu for table cells in the editable info/revisions part

When selecting a cell in a table in the editable info/revisions part, and then right-clicking, the following context menu appears:



The commands are the same as in the general context menu for the editable info/revisions part (see previous section for these), except for the following table commands:

Insert Row Above inserts a new row above the row of the selected cell.

Insert Row Below inserts a new row below the row of the selected cell.

Insert Column to the Left inserts a new column to the left of the column of the selected cell.

Insert Column to the Right inserts a new column to the right of the column of the selected cell.

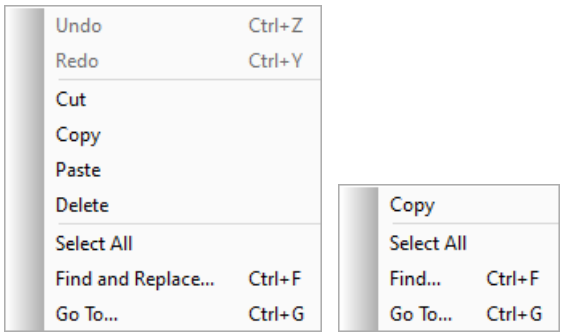
Delete Row deletes the row where the selected cell is located.

Delete Column deletes the column where the selected cell is located.

Context menu for HTML source code for info/revisions part

When viewing a read-only HTML source code information or revision part of the documentation layer, a context menu is available by right-clicking. The context menu depends on whether the class is editable or not:

Context menu for the HTML source code for info/revision part, editable and read-only classes.



For information about the entries, please see previous section. Note however that copying text here, from the source text, also copies HTML tags.

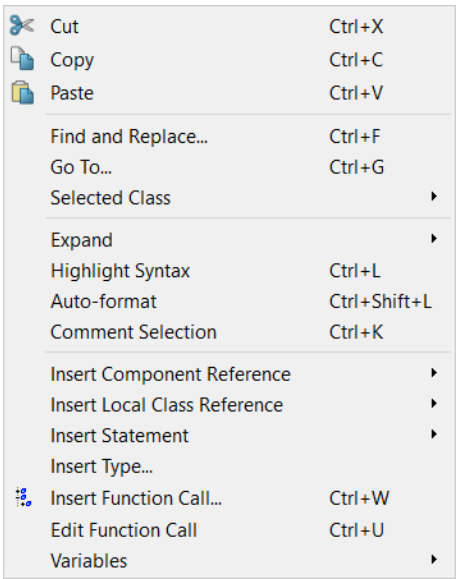
4.6.12 Context menu: Edit window (Modelica Text layer)

The Modelica Text layer is introduced in “Modelica Text layer” on page 194. When it comes to editing, please see “Programming in Modelica” on page 271.

If mathematical notation is activated, another context menu will be presented, see next section.

Right-clicking in the edit window displaying the Modelica Text layer will present a context menu with common text operations. Most of these operations are also available in the **Text** tab.

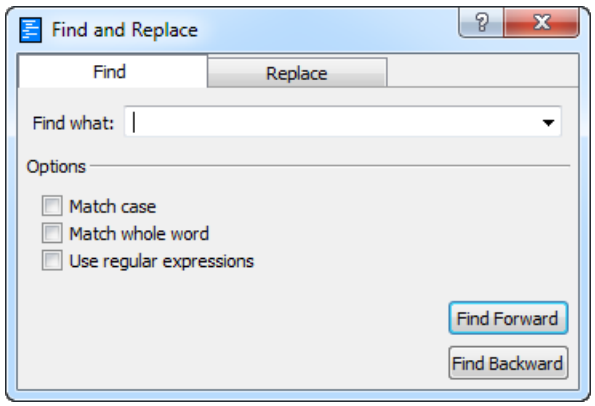
Context menu of the Modelica Text layer.



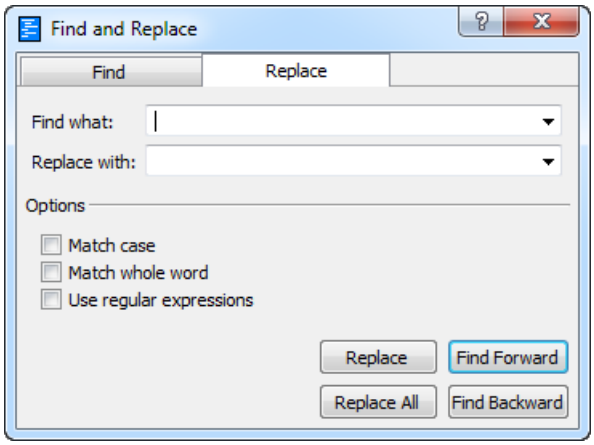
Cut, Copy, Paste copies text between the clipboard and the editor. It is possible to copy text with hidden annotations to clipboard.

Find and Replace finds and replaces text in the text editor. One tab is for find only, one tab is for find and replace.

Find tab.



Replace tab.



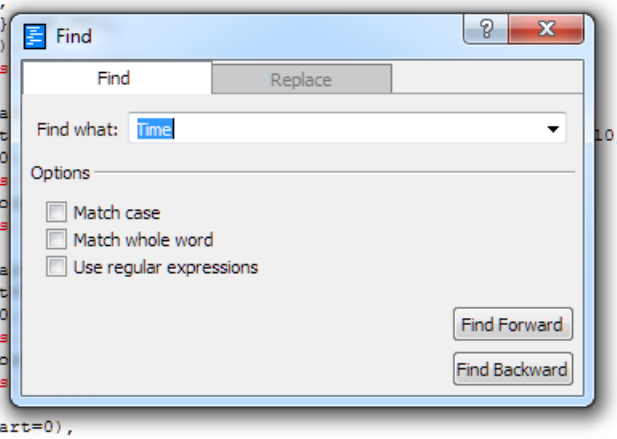
If the text is read-only, only the **Find** tab is accessible; the **Replace** tab is dimmed.

If you have the cursor on a word when activating the command, this word will be prefilled in the **Find what** box.

When you click **Find Forward** or **Find Backward**, you will find the next occurrence of the searched word/text going forward or backward in the text. All occurrences of whole words will also be highlighted in orange; the hit will be in blue.

```
model CoupledClutches "Drive train with 3 dynamically coupled clutches"
  extends Modelica.Icons.Example;
  parameter SI.Frequency freqHz=0.2
    "Frequency of sine function to invoke clutch1";
  parameter SI.Time T2=0.4 "Time when clutch2 is invoked";
  parameter SI.Time T3=0.9 "Time when clutch3 is invoked";

  Rotational.Components.Inertia J1(
    J=1,
    phi(fixed=true, start=0),
    w(start=10, fixed=true)) annotation (Placement(transformation(extent={{
            -70,-10},{-50,10}})));
  Rotational.Sources.Torque torque(useSupport=true) annotation (Placement(
        transformation(extent={{-100,-10},{-80,10}})));
  Rotational.Components.Clutch clutch1(peak=1.1, fn_max=20) annotation (
      Placement(transformation(extent={{-40,-10},{-20,10}})));
  Modelica.Blocks.Sources.Sine sin1(amplitude=10, freqHz=5) annotation (
      Placement(transformation(extent={{-130,-10},{-110,10}})));
  Modelica.Blocks.Sources.Step step1(startTime=T2) annotation (Placement(
        transformation(
        origin={25,35},
        extent={{-5,-5}
        rotation=270)))
  Rotational.Components
    J=1,
    phi(fixed=true, sta
    w(fixed=true, start
            -10},{10,10
  Rotational.Components
      Placement(transfo
  Rotational.Components
    J=1,
    phi(fixed=true, sta
    w(fixed=true, start
            50,-10},{70
  Rotational.Components
      Placement(transfo
  Rotational.Components
    J=1,
    phi(fixed=true, start=0),
    w(fixed=true, start=0)) annotation (Placement(transformation(extent={{
            110,-10},{130,10}})));
```



Note the difference between the **Find** command and the highlighting in orange; to avoid highlighting in orange of e.g. single characters, only whole words are highlighted in orange. As an example, the `Time` occurrence in `startTime` above the Find dialog in the figure above will be found by the **Find** command, but it will not be highlighted in orange since it is not a whole word.

You can disable the feature of highlighting all found occurrences of whole words by unticking the setting **Highlight all matches** in the **Modelica text editor** group, in the **Text Editor** tab of the command **Tools > Options**. This corresponds to setting the flag `Advanced.HighlightAllMatches = false;`.

The key **F3** is shortcut for the command **Find Forward**; the keys **Shift+F3** are shortcut for the command **Find Backward**.

The text window is automatically scrolled to display the found occurrence. If the occurrence is inside an annotation, that annotation will be expanded.

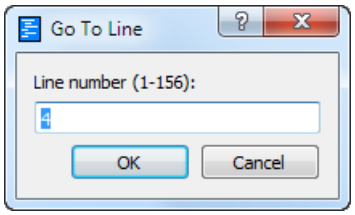
A warning is given if the text is not found.

Search history is available by clicking the arrow in the end of the **Find what** field. Replace history is available clicking the arrow in the end of the **Replace with** field.

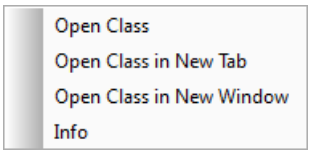
Regular expressions can be used in the search if this checkbox is checked. Special symbols in the regular expression are

*	Match everything.
?	Match any single character.
{ab}	Match characters a or b.
{a-z}	Match characters a through z.
{^ab}	Match any characters except a and b.
E+	Match one or more occurrence of E.
(ab|cd)	Match ab or cd.
\d	Match any digit.
\w	Match any digit or letter.
^	Match from start.
$	Match at end.

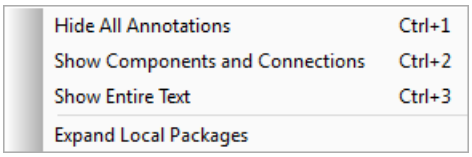
Go To… sets the cursor at specified line and scrolls window if necessary. The number of the current line is shown as default.



Selected Class allows you to open the selected class, or its documentation, by selecting the class name (or putting the cursor inside the name), right-clicking and then selecting the command.



Expand selects the expansion level. The following menu will be displayed when the cursor rests over this entry:



Hide all annotations will hide all annotations, displaying only annotation symbols. Components, and connections, including all annotations (e.g. graphical objects) are collapsed into the symbols and , symbolizing components and connections, respectively.

Show components and connect will expand the components and connections, but leave the annotations collapsed, symbolized .

Show entire text will expand everything, and solely text will be shown in the Modelica text layer, no symbols.

Expand local packages loads local packages; such packages are by default shown in the Modelica Text layer only as symbols, and not loaded. All local packages will be loaded when ticking the entry, including any local packages deeper in the tree structure Then each package can be individually expanded by clicking on the corresponding symbol or the corresponding expand icon . See also section "Local packages" on page 277.

Highlight Syntax scans the text in the editor and applies syntax highlighting for the Modelica language (without reformatting). Any syntax errors are reported in the log window.

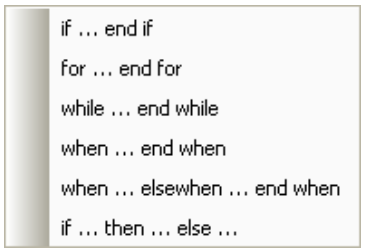
Auto-format makes it possible to auto-format (pretty print) a selection of the text.

Comment Selection can be used to comment out selected rows.

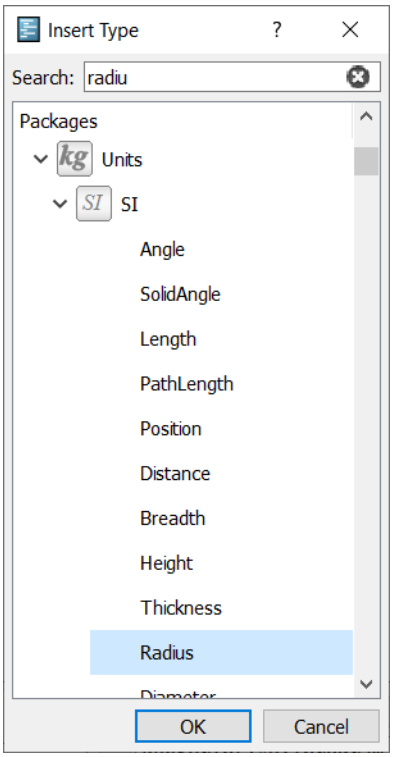
Insert Component Reference enables inserting a component reference in the text.

Insert Local Class Reference enables inserting a local class reference in the text.

Insert Statement presents a sub-menu with common Modelica constructs. This makes it easier to enter Modelica code.

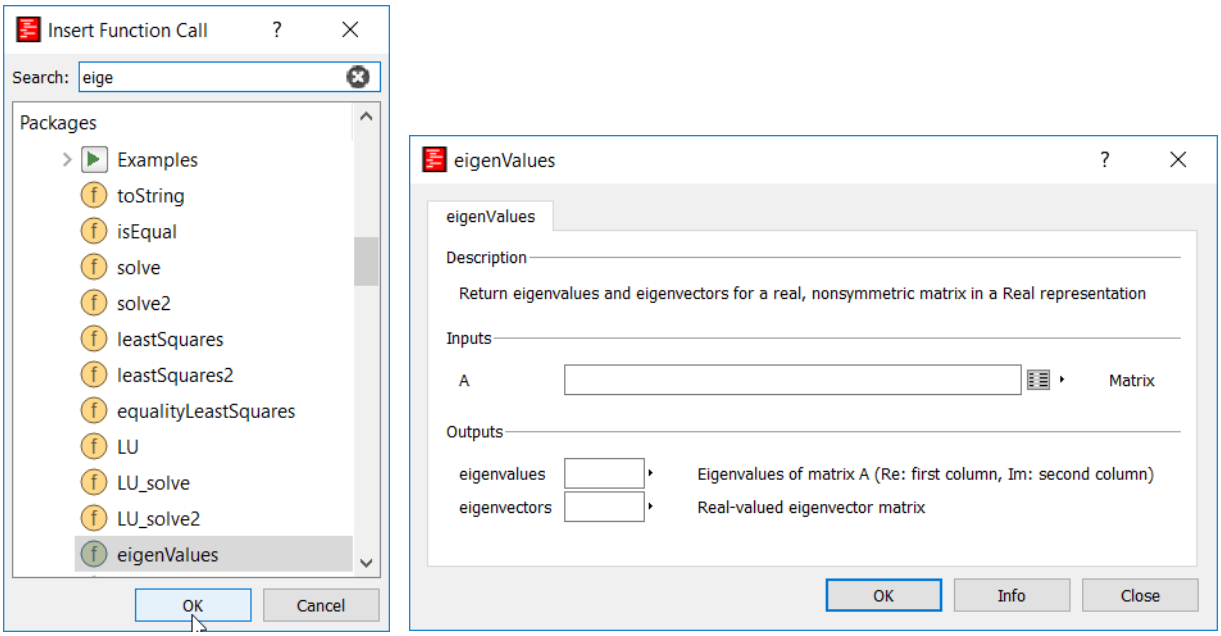


Insert Type enables inserting of a variable type by searching (or browsing). The result will be a text insertion. The example below shows searching by typing `radiu` in the **Search** input field. If **OK** is pressed in this situation, `Modelica.Units.SI.Radius` will be inserted as text.



For more information how to use this function, please see section “Parameters, variables and constants” starting on page 291.

Insert Function Call... enables searching (or browsing) for a function and then entering the arguments, allowing easy insertion of a function call. As an example, entering `eige` in the search input field (in order to find `Modelica.Math.Matrices.eigenValues)` will give the following result (followed by the resulting parameter dialog of the function that will pop when **OK** is clicked):

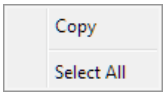


For more information how to use this function, please see section "Handling of function calls" starting on page 294.

Edit Function Call allows you to modify the arguments inside a function call using a parameter dialog. Select a function call up to ending parenthesis (or put the cursor inside name), right-click and select the command.

Variables allows you to define a new variable or edit an existing one using the variable declaration dialog. For details of this dialog, see section "Variable declaration dialog" starting on page 378.

4.6.13 Context menu: Edit window (Modelica Text layer – mathematical notation)



Presenting the Modelica text layer with mathematical notation is introduced in section "Displaying the Modelica Text layer with mathematical notation" starting on page 278.

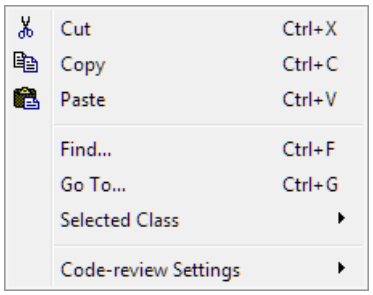
Copy will copy selected parts to the clipboard. Please note that there is no **Paste** in this menu, copied parts can be inserted in e.g. Microsoft Word for further elaboration. Text parts will be inserted as text in Microsoft Word, equations will be inserted as a picture.

(If pasting the copied text into e.g. Microsoft Notepad, the text version will be inserted instead; the formulas will be presented as if the mathematical notation was not activated.)

Select All will select all parts of the layer. See also **Copy** above.

4.6.14 Context menu: Edit window (Used Classes layer)

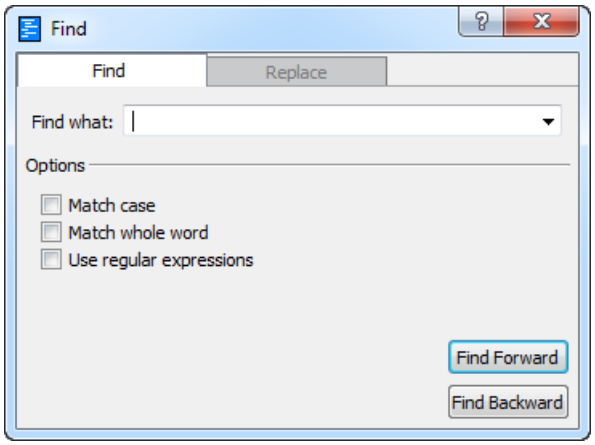
The Used Classes layer is introduced in “Used Classes layer” on page 196. Right-clicking in the layer displays the following context menu:



Right-clicking in the Used Classes layer in the edit window will present a menu with the following choices:

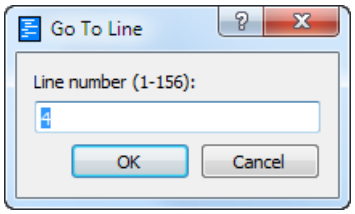
Cut, Copy Paste – copies text between the clipboard and the editor.

Find… – search functionality for a specific text (the **Replace** tab is not available – the layer is read-only). The menu looks like:

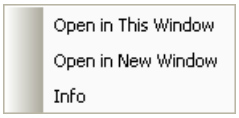


For more information about this command, see the command **Find and Replace** in the section “Context menu: Edit window (Modelica Text layer)” starting on page 533.

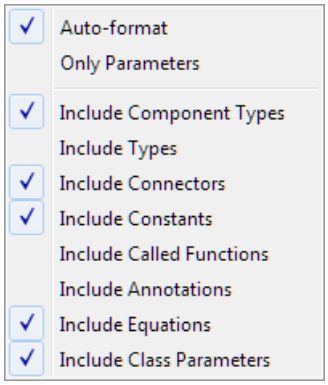
Go To … – use of the following menu. The number of the current line is shown as default.



Selected Class Allows you to open the selected class, or its documentation, by resting the cursor over the class name and then using the command.

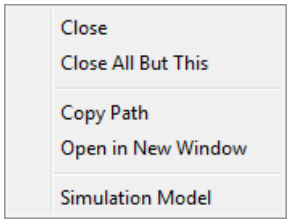


Code-review Settings – selects what should be seen:



Auto-format is by default activated and presents the documentation according to automatic formatting. If the formatting implemented by the writer of the class documentation should be used instead, this alternative should be unchecked.

4.6.15 Context menu: Model tabs



Model tabs are introduced in the section “Model tabs” starting on page 200.

Right-clicking a model tab displays the context menu. The commands available are:

Close closes the tab. If the last tab is closed, Dymola closes.

Close All But This closes all tabs except this tab.

Copy Path copies the module path to the clipboard.

Open in New Window opens the class in a new window.

Simulation Model selects the model in the tab as active simulation model. The command works in toggle mode, by default it is not set.

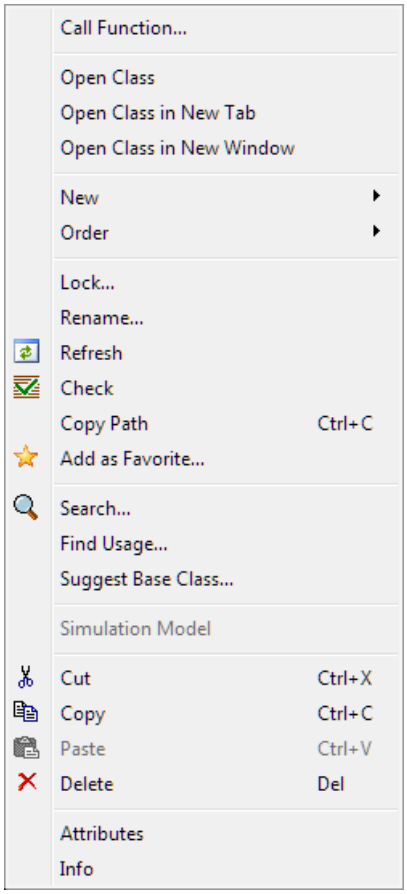
4.6.16 Context menu: Components

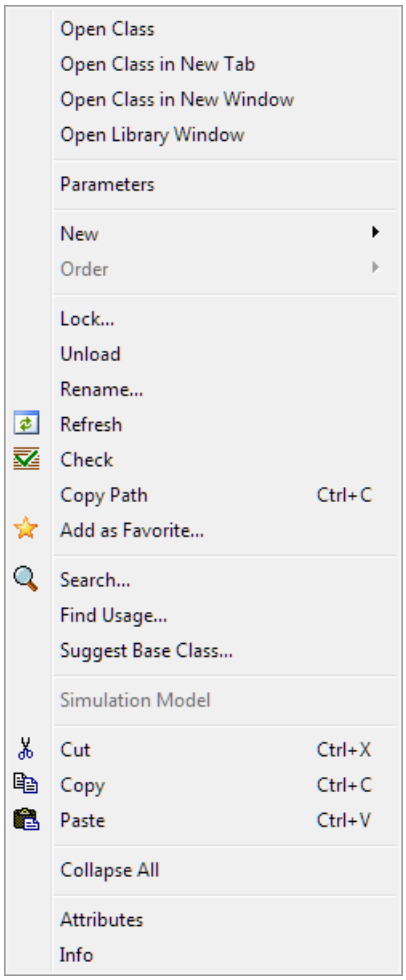
Context menu for components in the package browser

The package browser is introduced in “Package and component browsers” starting on page 202.

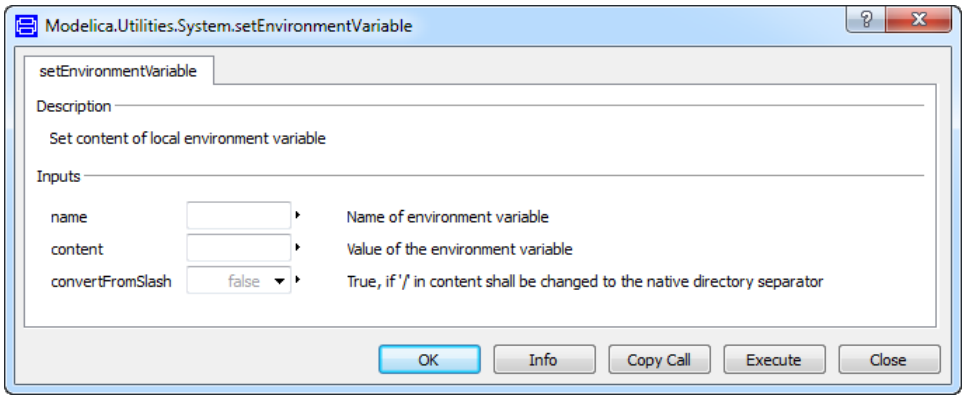
Right-clicking on a component in the package browser presents a context menu with a number of commands. The commands available differ depending on what component is selected, and if it is protected or not.

Context menu for non-protected function and top-level package.

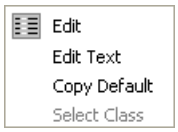




Call Function... (available only for functions) displays the parameter dialog for the function.



The input field can contain a pre-defined value and a drop-down menu for selecting values (values can be entered not using that, however). The arrow after the field displays the context menu of the input field:



Edit will show a suitable hierarchical editor if needed.

Edit text will show a window with a larger input field (line editor). It is needed for modifying e.g. long parameter expressions and re-declarations.

Copy Default copies the text of the default value to the input field for further editing.

Select Class will give the possibility to select a class from a list rather than type it in.

Quote String Having entered a value for a string parameter, the input can be quoted by this command.

The **OK** button will execute the changes and close the dialog window, while the **Execute** button will execute the changes without closing the dialog window. (Executing the function usually also changes the mode from Modeling to Simulation.) The **Info** button will display information about the function and the button **Copy Call** copies the function call including parameter list to the clipboard. The **Close** button will close the dialog window without executing anything.

Open Class opens the class of the selected class in this window (corresponding to the active tab), except if the window/tab contains the active simulation model. If so, the class is instead opened in a new tab.

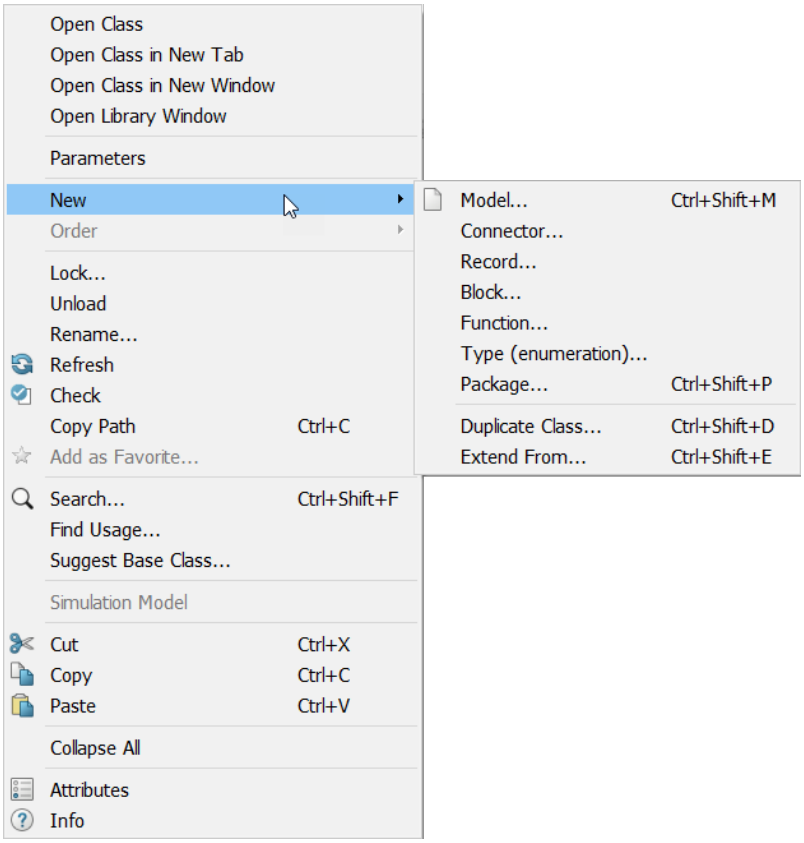
Open Class in New Tab opens the class in a new tab.

Open Class in New Window opens a new main window with the selected class.

Open Library Window creates a new library window. Please note that this command is only available if a suitable component is selected.

Parameters will open the parameter dialog for the selected component (if relevant). For information about the parameter dialog, please see section "Editing parameters and variables" on page 237.

New makes it possible to create new classes, and to duplicate the selected class, or to extend from it.



Model… etc creates a new class of the desired type in the selected package. See also "File > New > Model etc." on page 448.

Duplicate Class… Duplicates a class, see "File > New > Duplicate Class" on page 451. Note that you can also duplicate a class by using the context commands **Copy** and **Paste**, or by keeping **Ctrl** pressed and drag and drop the class to another top level package than where the class resides.

Extend From… creates a new class, which extends from the selected class. See also "File > New > Model etc." on page 448.

Order moves the class up or down in the package browser. This requires that the class and the enclosing package are both stored in the same file. Use "Rename" if you want to move the class to another package (see that command below). Note that you can also drag and drop the class to change the order.

Lock… locks a class For information about locking and unlocking, please see section "Locking and unlocking classes" starting on page 335.

Unlock <*reason for locking*> unlocks a locked class. See previous command.

Unload unloads a model from Dymola. Note that there is a built-in function `eraseClasses` that can be used to, for example, unload libraries by scripting. For more information about this function, see the index entry *"eraseClasses"* in the index in the end of this manual.

Rename… renames a class. This can move the class to a different package.

For overview information about renaming classes, see also section "Renaming a class" on page 234.

Notes:

- Rename can also be done by:
    - In the package browser, select a class; then click it again, or press **F2**. An inline editor opens where you can enter a new name. Finalize by pressing **Enter** (or **Escape** to cancel).
    - Dragging a package to a location in another top level package than the one where the original package is located.
    - Using the context commands **Cut** and **Paste** in the package browser.
    - (not recommended) Changing the name in the Modelica text layer.
- Renaming classes in any way, except editing Modelica text, is by default configured to automatically update the following items. (This setting can be changed by the command **Tools > Options**, see the setting "Updates other classes when renaming class" in section "Package Browser options tab" starting on page 512 for alternatives and details.)
    - All references in all loaded classes to reflect the name-change. There is no undo command for this. **Note!** Dangling references to the unloaded or renamed class may not be detected. Make **Check** (or **Text > Check**) on the enclosing package to detect some potential problems, see "Text > Check" starting on page 491.
    - Directories pointed to by LibraryDirectory and IncludeDirectory are handled, and the references updated. Other resources, e.g. images, are not copied.
- To prevent conflicting names in some operating systems, for example Windows, where file names are not case sensitive, contrary to Modelica names, warnings are given if you rename a file in such a way that it differs to an existing file in the same scope only by upper case/lower case. (For details, see "Warning about potentially conflicting Modelica names when creating or renaming classes" on page 233.)

- For changing the order of classes within a package, drag them, or use the context commad **Order**.

Refresh refresh a package. This is an easier way than unloading the package and opening it again.

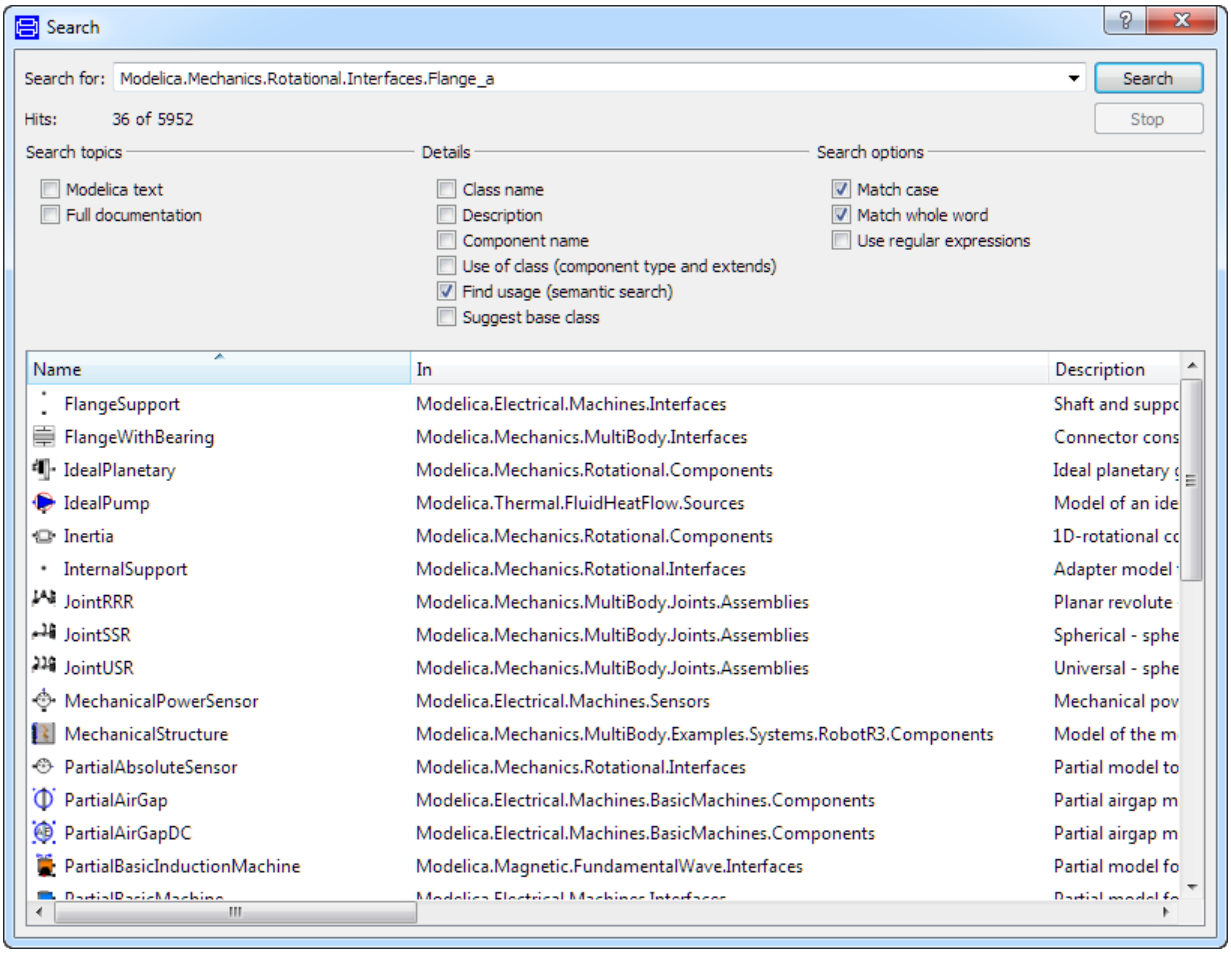
Check checks the selected class for errors. See “Text > Check” starting on page 491.

Copy Path will copy the Modelica path to the class to the clipboard, e.g. `MyPackage.MyModel`.

Add as Favorite… adds the selected class to favorites. See the section “The Favorites packages” on page 227 for details.

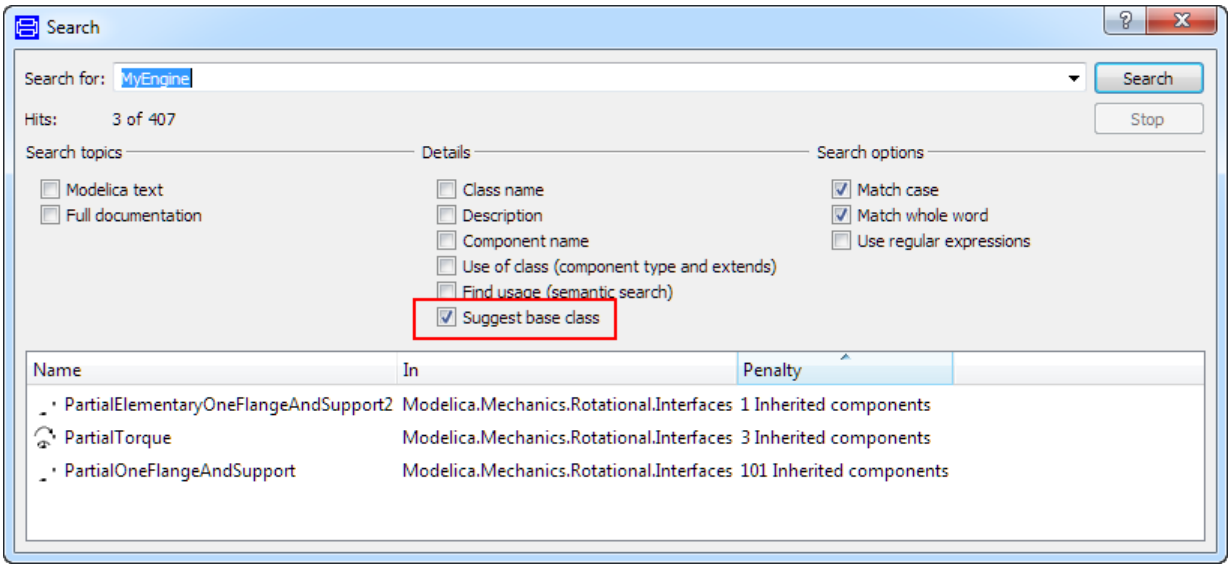
Search... Searches the selected class for all classes or components matching a search pattern. See “File > Search” on page 458. Please compare with the next command.

Find Usage… Performs a semantic search for the usage of the selected class in the current package. This command is actually the previous command with specific settings, as an example:



The advantage with this command is the semantic search, instead of searching for exact strings, the search really displays where a class is used.

Suggest Base Class… Performs a search to find suitable base classes for the selected class. This is actually the previous **Search…** command with specific settings. An example:



By double-clicking any of the results in the pane you can select to adapt the selected class to the found base class. Please see section “Building test-driven template models” starting on page 372 for an example how to do this..

Simulation Model (available for items possible to simulate). Selects the model as active simulation model. See section “Selection of active simulation model” starting on page 325 for more information.

Cut copies the selected class to the clipboard. The operation is concluded when the class is pasted.

Copy copies the selected class to the clipboard.

Paste pastes a cut/copied class to the class that is open. If this class resides in another top level package than the package that the original class where cut/copied from, a **Move**/**Copy** dialog is opened, to let the user conclude the operation.

Delete Deletes the class.

Collapse All collapses an open hierarchy and all lower levels (a package and all sub-packages). The difference between this command and clicking the collapse icon ◢ of the package is how the package is expanded when clicking on the expand icon ▷ afterwards. If the package has been collapsed by the **Collapse All** command, only the first level of the hierarchy is opened when clicking the expand icon. If the package has been collapsed by the collapse icon, all levels previously open will be opened again when clicking the expand icon.

Expand All expands all lower levels of a collapsed hierarchy.

Attributes opens a dialog for the properties of the class. See “Graphics > Attributes” starting on page 479.

Info displays extended documentation for the selected class. HTML documentation will be used if it is available, both for the standard packages and for classes written by users.



What's this?

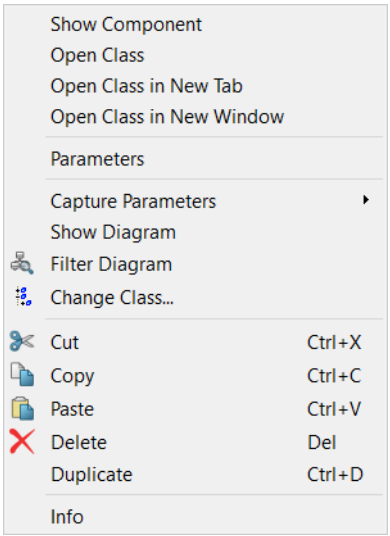
You can also use **What's This?** on each class to display its documentation in a small popup-window inside Dymola.

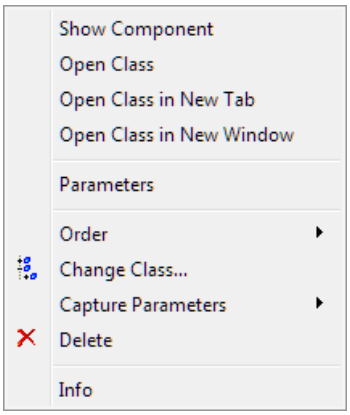
Context menu for components in the diagram layer and the component browser

The context menu of an object in the diagram layer is introduced in "The context menu" on page 236. The component browser is introduced in "Package and component browsers" starting on page 202.

Right-clicking when a component is selected in the diagram layer of an edit window presents a context menu with the following choices (the figure to the left below). A similar menu is presented when right-clicking a component in the component browser (the figure to the right below).

The context menu of components in the diagram layer and in the component browser.







Show Component shows the class of the component or connector in this window. Press the button **Back** in the view toolbar to go back. (You can also use the mouse button Back if you have such a mouse.)

Open Class opens the class in the present window (corresponding to the active tab), if the window/tab does not contain the active simulation model. If so, the class is opened in a new tab.

Open Class in New Tab opens the class in a new tab.

Open Class in New Window opens a new main window with the class of the component or connector.

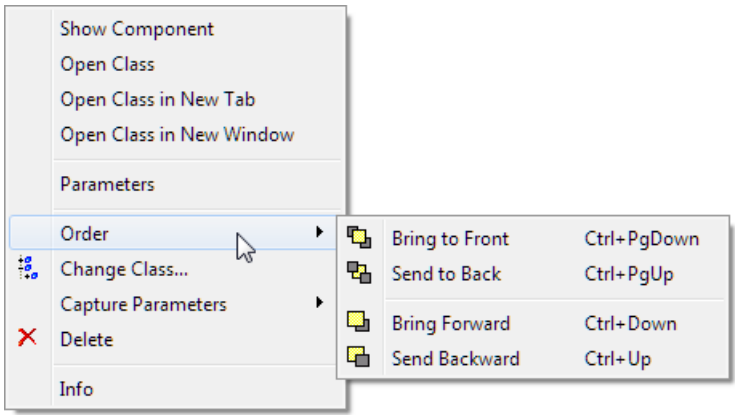
Parameters opens the parameter dialog. For information about the parameter dialog, please see section "Editing parameters and variables" on page 237.

Variable… opens the variable declaration dialog for a selected variable. Note that the setting **Include non-graphical** must be set if variables should be shown in the component browser.

See this setting in section “General options tab” starting on page 511. (You can also double-click the variable in the component browser to display the variable declaration dialog.)

Order Resting the cursor over this entry will display:

Using the Order functionality.

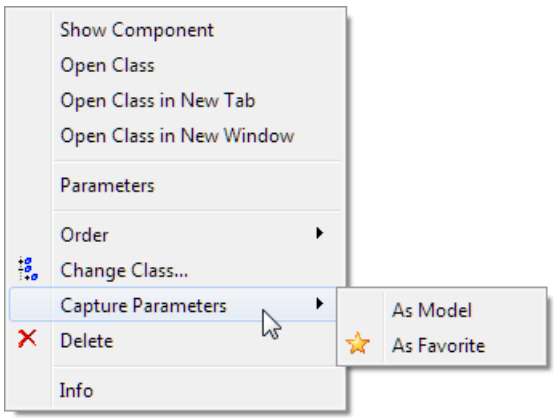


This commands will work on the component browser as well as the diagram layer (by using the command in the **Graphics** tab). As an example **Bring to Front** will bring a component with graphical representation in the diagram layer to the *front* of that layer while at the same time will bring the component to the *bottom* of the component browser. The last two commands work with one layer/level at a time.

Note the specific case when components overlap, it is convenient to use the commands in the Component Browser in such a case.

Capture Parameters Resting the cursor over this entry will display:

Capturing parameters



These commands are available from both the diagram layer and the component browser. These commands do the following:

- **As Model**: Creates a new model from the selected component´s model by extending it, with the connected parameter values and other modifiers included.

- **As Favorite**: Adds a new model, created from the selected component´s model, to favorites, with the connected parameter values and other modifiers of the component included.

For more information about these commands, see “Creating a new model from a component with parameter values included” starting on page 336 (for creating a new model), and “The Favorites packages” starting on page 227 (for adding to favorites).

Show Diagram displays the diagram layer of the component, instead of the icon layer. Note the difference between this command and the command **Show Component**; the latter displays the diagram layer of a component in the whole window, while **Show Diagram** only displays the diagram layer for the selected component, without changing the content in the rest of the window.

Filter Diagram Filter to dim component from selected packages and/or not matching a name. See section “Filtering objects” on page 222 for more information.



Change Class… It is possible to change class of one or several replaceable components (of the same class) using the context menu for the component. Please see section “Working with replaceable components” on page 346 for more information.

Cut, **Copy**, **Paste**, **Delete** and **Duplicate** corresponds to the commands in the **Graphics** tab. Please see these commands in section “Main window: Graphics tab” starting on page 467. Concerning **Delete**, if multiple selections should be made, the only way to do that is to do it in the diagram layer.

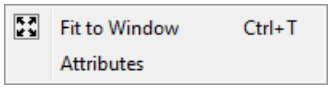
(**Set Unit** appears as entry if you have selected a connector of a component in the diagram layer instead of the component itself. See “Adding unit by using the context menu of the connector” on page 360.)

Info displays extended documentation for the class of the component or connector. HTML documentation will be used if it is available, both for the standard packages and for classes written by users.

4.6.17 Context menu: Coordinate system boundary

The coordinate system is introduced in section “Coordinate system” on page 197.

Right-clicking when the coordinate system boundary is selected displays the context menu.



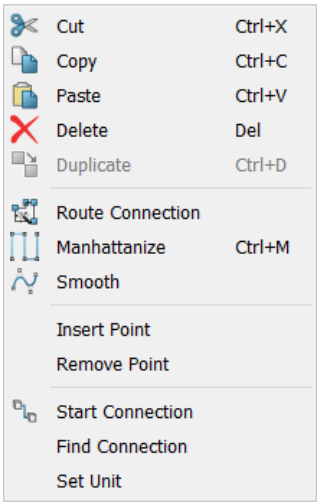
Fit to Window will zoom to display all objects of the model in the window. In most cases this corresponds to a zoom factor of 100 %, but if e.g. objects are placed outside the coordinate system, the fit to window zoom factor will be different.

Attributes displays the Graphics tab of the command **Graphics > Attributes**. See section “Graphics tab” on page 480 for more information.

4.6.18 Context menu: Connections

Connections are introduced in section “Connections” starting on page 247.

Right-clicking a connection displays the context menu:



General entries

Most of the commands are also available from the **Graphics** tab. For such commands, see section “Main window: Graphics tab” starting on page 467.

Specific commands in the context menu for connections

The below commands are only available in the context menu, or are they are specific for connections:

Route Connection will route the selected connection.

Insert Point will insert a new point in the selected connection. The point is inserted in the point closest to the cursor in the selected connection.

Remove point will remove the point closest to the cursor in the selected connection.

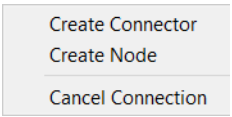
Start Connection will insert a new connection starting from the location of the cursor on the connection line. See “Start connection from connection / Connect to connection” on page 250 for details.

Find Connection will present the connections and connected components in the displayed diagram layer, using a prepopulated menu. See “Searching connections and connected components” on page 257 for details.

Set Unit will make it possible to add a unit to the connector(s) of the connection line if any block is connected to it. For details, see section “Adding unit by using the context menu of the connection line” on page 363. Note that if unit is already set, the menu entry is not visible in the context menu.

4.6.19 Context menu: Connection while connecting

Connections are introduced in “Connections” starting on page 247.



Right-clicking or double-clicking while the connection is being draw presents a context menu with these choices.

Create Connector creates a new connector at the cursor position and the connection is completed. This connector has the same type as the connector at the start of the connection. This operation is typically used to draw a connection from a component which should end with an identical connector in the enclosing class.

Create Node creates a new protected connector (internal node) at the cursor position and the connection is completed. This connector has the same type as the connector at the start of the connection. The size of the new connector is 1/10 of the normal component size.

Cancel Connection cancels the connection operation; no connection is made.

4.6.20 Context menu: Connectors

The context menu of a connector is the same as the corresponding context menu of the component itself in the diagram layer, except the below item.

For the context menu of a component, see section “Context menu for components in the diagram layer and the component browser” starting on page 551.

Set Unit will make it possible to add a unit to the connector of a block. For details, see section “Adding unit by using the context menu of the connector” on page 360. Note that if unit is already set, the menu entry is not visible in the context menu.

4.6.21 Context menu: Graphical objects

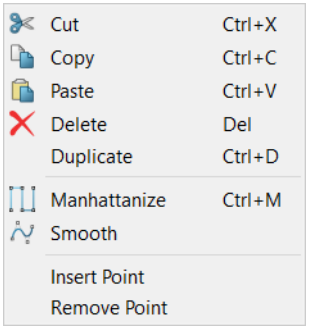
Editing of graphical objects is introduced in “Changing graphical attributes” on page 270.

A context menu is available by selecting a graphical object and right-clicking. (It is also possible to get the menu for an object by right-clicking when the cursor is resting on the contour of the object.) The following might be available, depending on what type of graphical object is selected:

Please note that not all possibilities for graphical objects are covered by this menu! Line and area color etc. must be edited using other commands. Please see section “Line style and Fill style” on page 271 for more details.

General entries in the context menu of graphical objects

As an example, consider the context menu for a line:



Most commands are the same the ones in the **Graphics** tab. For such commands, please see section “Main window: Graphics tab” starting on page 467.

Specific commands in the context menu of various graphical objects

Line

The commands specific for the line context menu are:

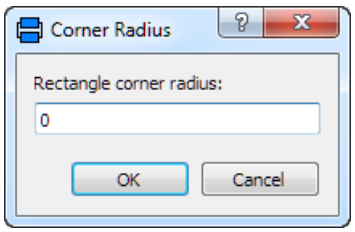
Insert Point will insert a new point in the selected object. The point is inserted in the point closest to the cursor in the selected object.

Remove point will remove the point closest to the cursor in the selected object.

Rectangle

The command specific for the rectangle context menu is

Corner Radius… will display a menu that looks like:

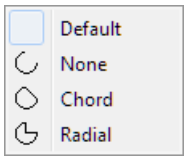


Using this menu the rectangle can be given rounded corners. Please see section “Rectangles” on page 262 for more details.

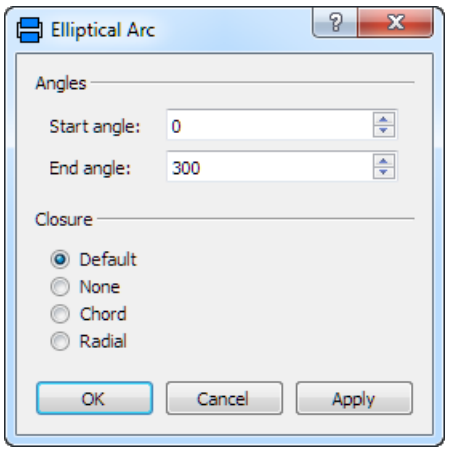
Ellipse

The commands specific for the context menu of an ellipse are:

- **Ellipse Closure** that can change how an elliptic arc is closed:



- **Elliptical Arc…** that displays the following for an elliptic arc 0-300:



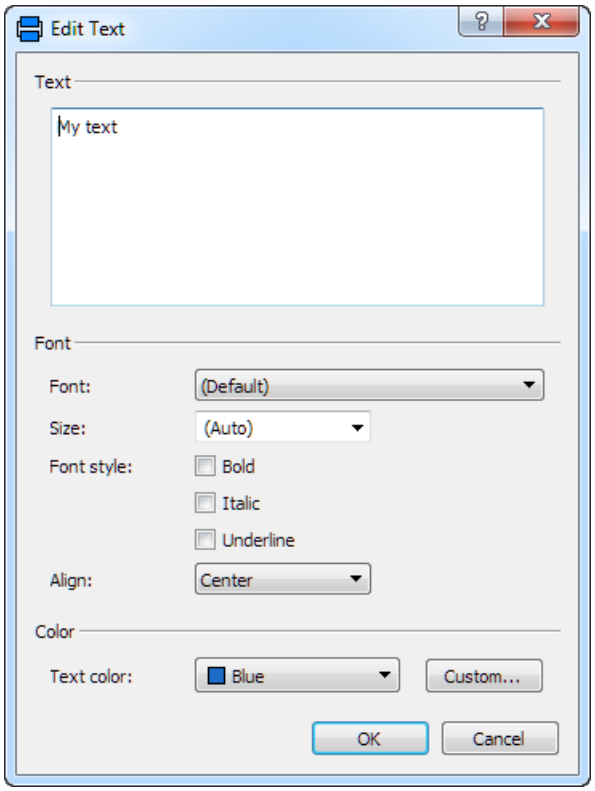
For details on these commands, please see section “Ellipses” on page 263.

Polygon

The commands specific for the context menu of the polygon are the same as for the line; please see above.

Text

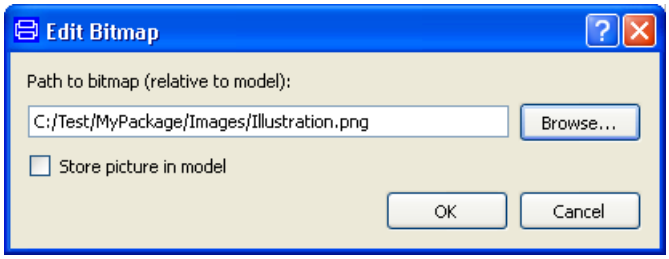
The command specific for the context menu of the text object displaying e.g. “My text” is **Edit text…** that displays the following:



For more information, please see section “Text” on page 265.

Bitmap

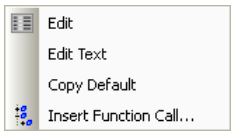
The command specific for the context menu of the bitmap is **Edit Bitmap…** that will display the following:



For more information, please see section “Bitmap” on page 268.

4.6.22 Context menu: Variable declaration dialog; Value input field

The variable declaration dialog is described in “Variable declaration dialog” starting on page 378.



Right-clicking in the **Value** field in the Declaration tab in the variable declaration dialog or clicking on the arrow after the field will pop up a context menu.

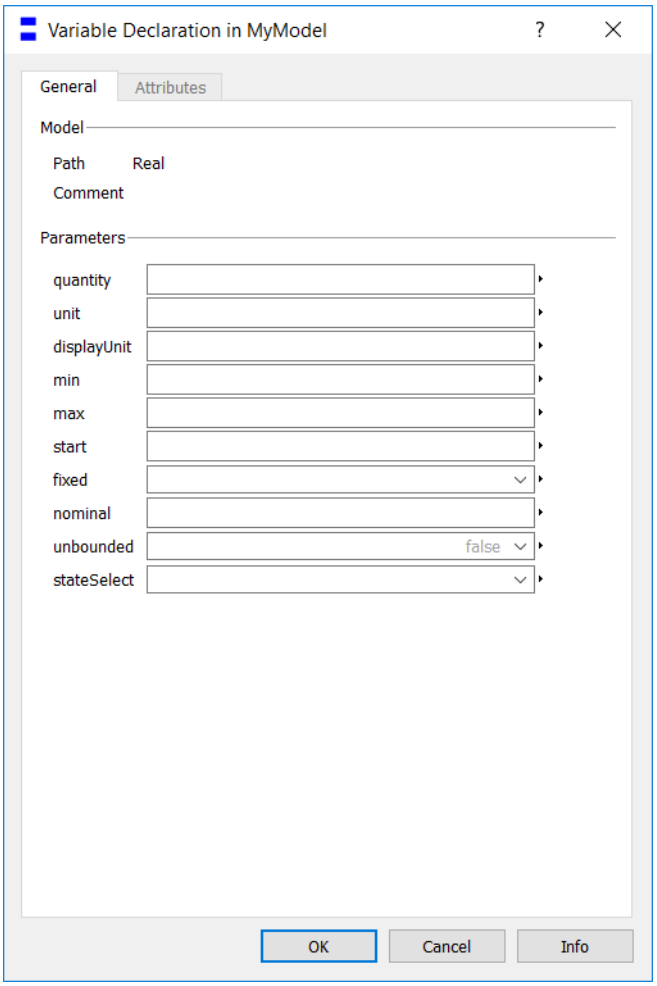
The **Edit** command (or pressing the **Edit** button symbol) will open an editor for the variable value. The editor will look different depending on if the value is just a single value or if it is a matrix of values (in the latter case the number of rows and columns are present in the **Array dimensions** field in the variable declaration menu).

(**Edit Text**, **Copy Default** and **Insert Function Call...** will be presented in a section when **Edit** has been dealt with.)

Edit menu of a single value

If the value to edit is a single value the following will pop up (this example is a declaration of a simple Real variable):

Edit menu of a single value.



The **Model** group contains information about the **Path** and **Comment** of the variable.

The **Parameters** group contains a number of attributes that can be added.

quantity – the value.

unit – the unit of the variable used in calculations.

displayUnit – the unit that will be displayed in e.g. plot windows.

min – the minimum value of the variable.

max – the maximum value of the variable.

start – the start variable of the variable.

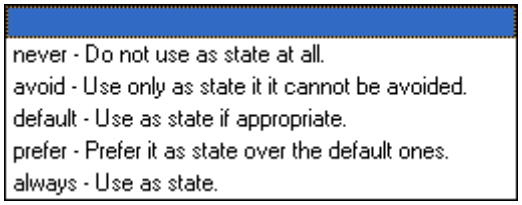
fixed – whether the parameter/variable is fixed or not (could be selected as **false** or **true**, default is true for parameters, false for variables). This controls if the start-value is fixed (i.e. will be satisfied during init) or not (start-value is a guess-value).

nominal – indicate the magnitude of the value

unbounded – if true, it means that the value may grow without bound, and only the absolute error should be controlled

stateSelect – this gives the preference state selection. (For more information about state selection please see the chapter "Advanced Modelica Support", section "Means to control the selection of states".) The choice is made using a drop-down menu:

stateSelect alternatives.



By not entering anything the default value is selected.

Each of these fields also has a context menu.

Edit menu of a matrix of values (matrix editor)

Dymola has specialized array and matrix editors for entering structured parameters. The array and matrix editor allows editing of both elements and array size (if allowed). A context menu is available to insert rows (before the selected entry) and delete rows and columns. Insertion after the last entry is performed by increasing the size.

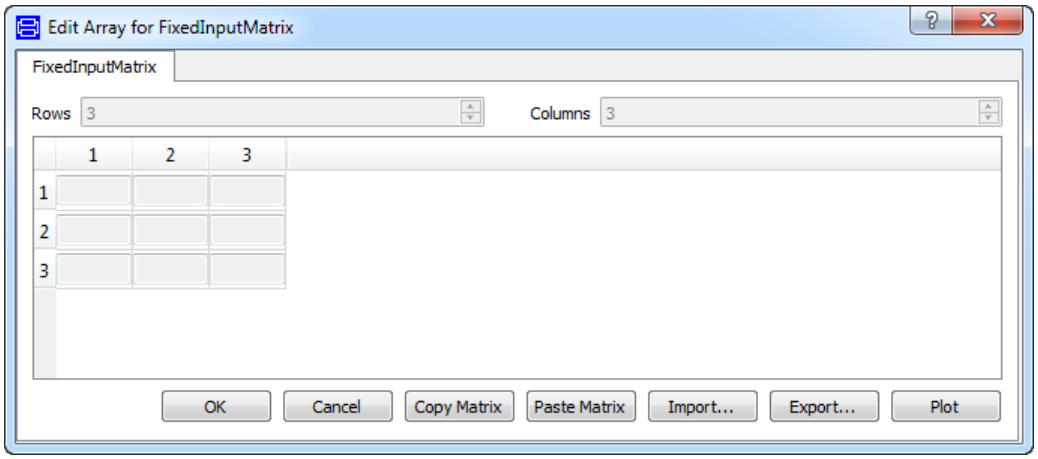
For handling of large matrices (more than 100 000 cells) in the matrix editor, see "Handling of large matrices (more than 100 000 cells)" on page 569.

For general information about using data from files, see section "Using data from files" starting on page 420.

Fixed size of the matrix

When the values of a fixed-size matrix should be specified (e.g. a parameter declared [3, 3]) the edit menu will look like the following:

Matrix editor – fixed size of matrix.



Concerning the values, only numbers can be entered – no attributes of the values can be given.

A context menu is available by right-clicking in any value field. That menu is described in the next section (where more alternatives can be used).

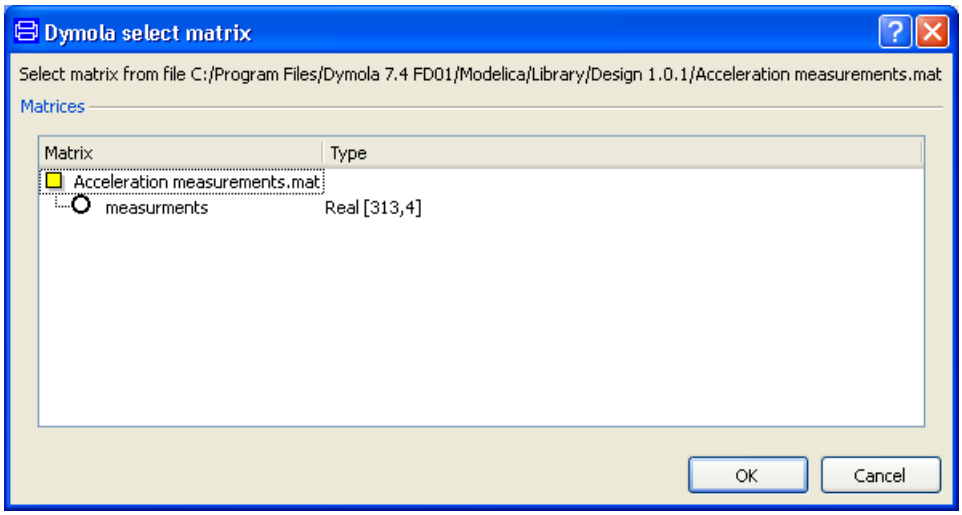
The table can be copied from the matrix editor by using the **Copy Matrix** button. The copied table can then be pasted in e.g. Microsoft Excel. The whole table is copied; it is not possible to select a part to copy.

Data for the table can be pasted from copied cells from for example Microsoft Excel or Matlab, by using the **Paste Matrix** button. If the number of rows and columns to paste is not the same as the number of rows and columns in the existing matrix, a warning is issued.

Data for the table can also be loaded using the **Import...** button. The data can be imported from external Matlab 4 (.mat files), Scientific Data Format (.sdf) or Comma separated values files. Comma separated values files can have extensions .csv or .txt. Files in .txt format can have tab, space, semicolon or comma as separators.

When importing from a .mat file, the name of the matrix needs to be specified. The following is an example of a window that will be displayed:

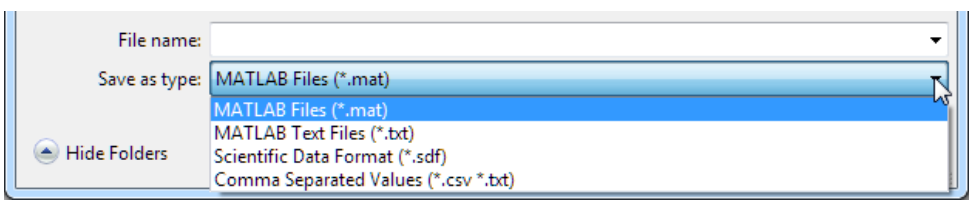
Matrix selector.



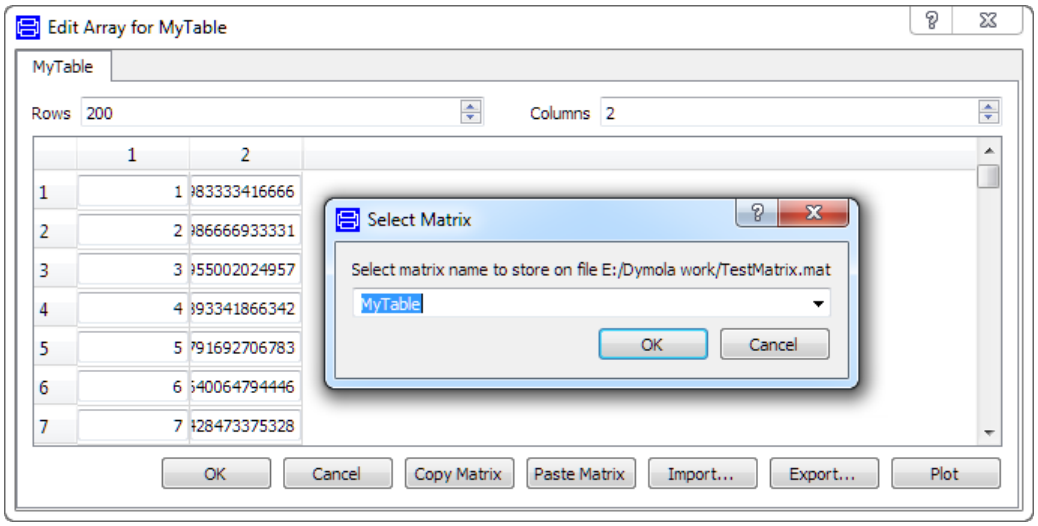
Please note that the selection can only be done by clicking in the "circle" before the table name!

The **Export...** button enables saving the matrix in the same format as **Import...**.

Menu for selection of file format when saving.



A matrix name has also to be specified if saving in .mat format, an example:



(Note that history of names is available using the arrow in the end of the line.)

Comma separated values in .txt format will however always have tab as delimiter. (If such a file should be saved, the extension .txt has to be added to the file name in the dialog.)

Please note that the above is the way to create .txt files for most users - the **MATLAB Text files (*.txt)** alternative is a specific alternative that creates files in a specific text format. By saving a 3x3 table as “ytab” in this format and setting the table name to “ytable” in the dialog that follows a file that looks the following when opened in Notepad is saved:

Example of the specific Text files format.

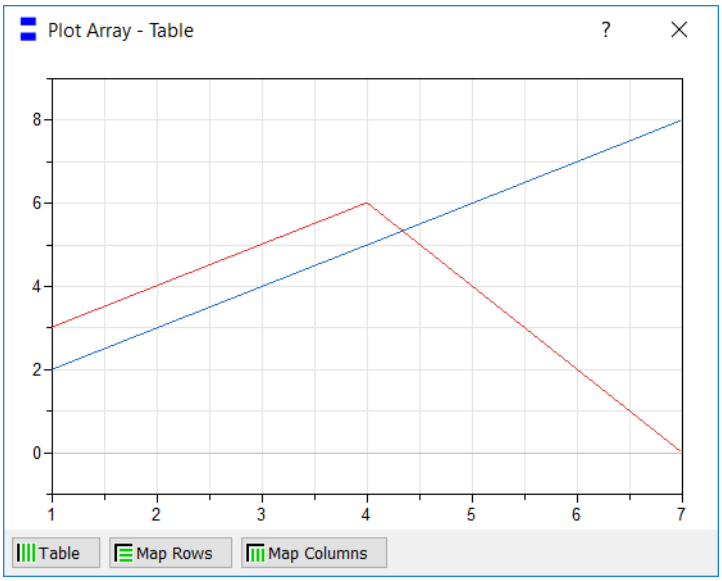
ytab.txt - Notepad

File Edit Format View Help

```
#1
double ytable(3,3)
  1  2  3
  4  5  6
  7  8  0
```

The matrix editor has extended dialog for plotting one- and two-dimensional table functions. The plot is displayed by pressing the **Plot** button and then pressing the **Table** button. If we do that for the ytable above, we get:

Matrix editor plot.



The contents of the table can be interpreted in three different ways when plotted. The type of plot is selected by pressing the corresponding button. Maps are available if the matrix size is greater than 2x2.

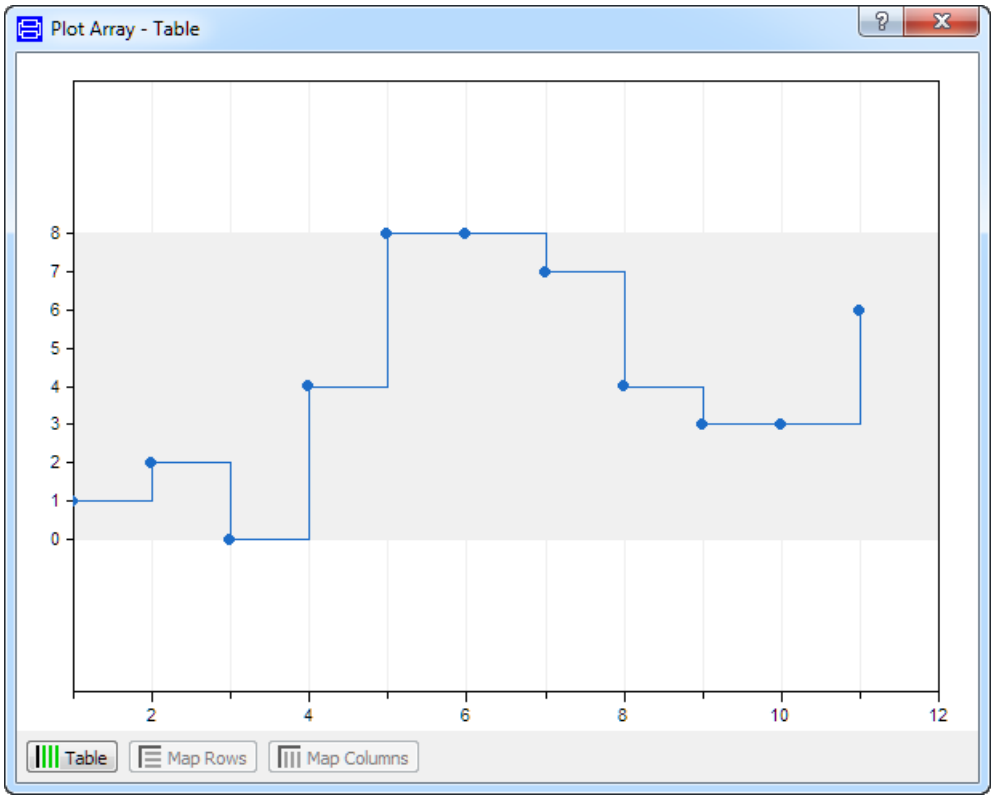
Table plots columns 2..n versus first column. Vectors are plotted against index number.

Map Rows plots rows of a 2D map. Plots rows 2..n versus first row, except the first element of each row.

Map Columns plots columns of a 2D map. Plots columns 2..n versus the first column, except first element of each column.

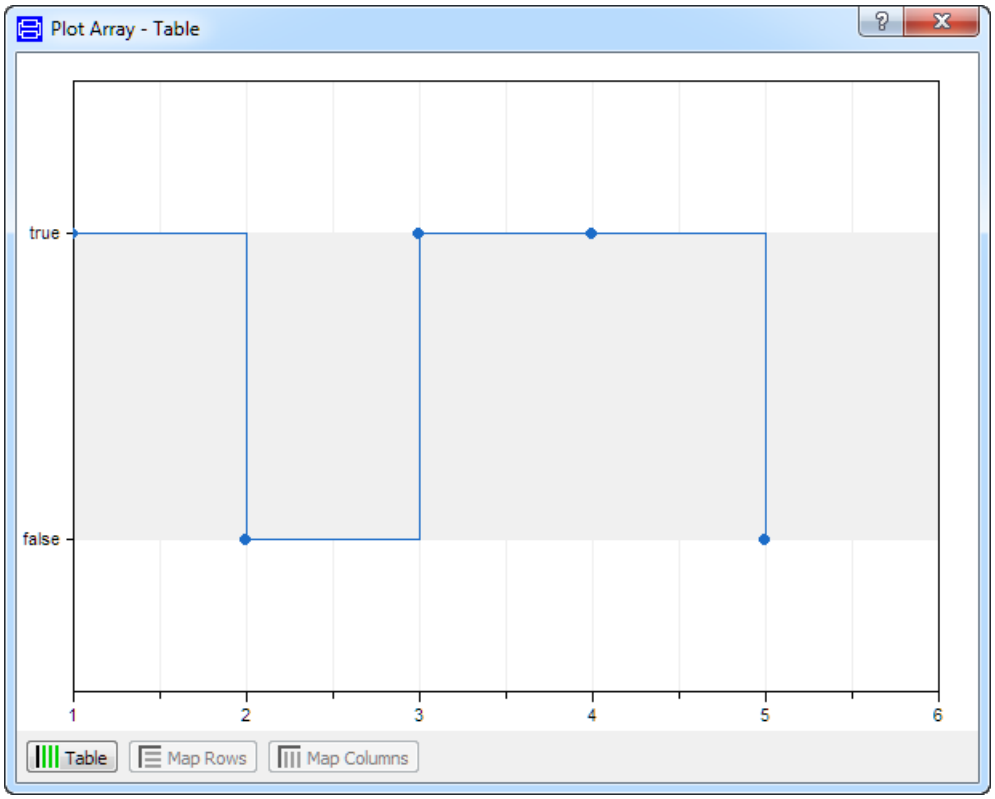
Plotting Boolean arrays and short integer arrays are supported. Short integer arrays are by default defined as integer arrays where the span between minimum and maximum value is less than 20. An example of a plot of the following short integer array:

```
Integer i[:]={1,2,0,4,8,8,7,4,3,3,6}
```

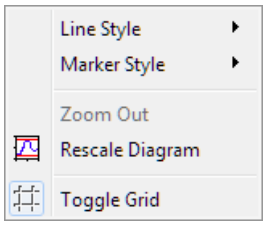


An example of a plot of the following Boolean array:

```
Boolean b[:]={true, false, true, true, false}
```



A context menu is available by right-clicking in the window:

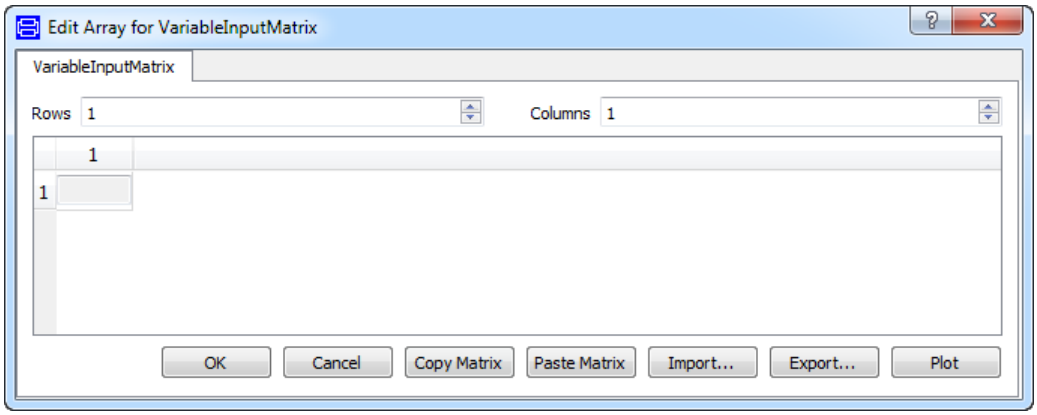


For information, please see section “Context menu: Parameter dialog; plot window” on page 571.

Variable size of the matrix

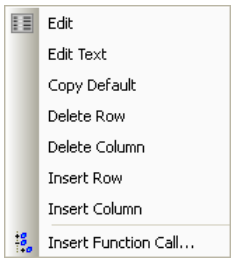
It the case the matrix is of variable size (e.g. a parameter declared [:, :]), the edit menu will look like the following when entered the first time:

Matrix editor – variable size of matrix.



In this case the number of rows and columns can be manipulated.

One way is to use the context menu of any value field in the matrix.



Edit and **Edit Text** will in this case both show a text editor for longer strings, here not of much use.

Edit Combined gives a special editor for array of records if such a construction is present.

Copy Default copies the text of the default value to the input field for further editing.

Delete Row and **Delete Column** will delete the row/column where the cursor is located when displaying the context menu.

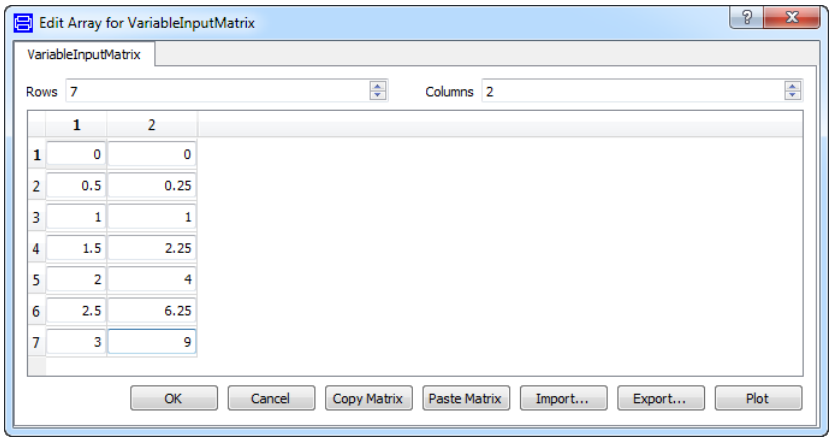
Insert Row and **Insert Column** will insert a row/column before the row/column where the cursor is located when displaying the context menu.

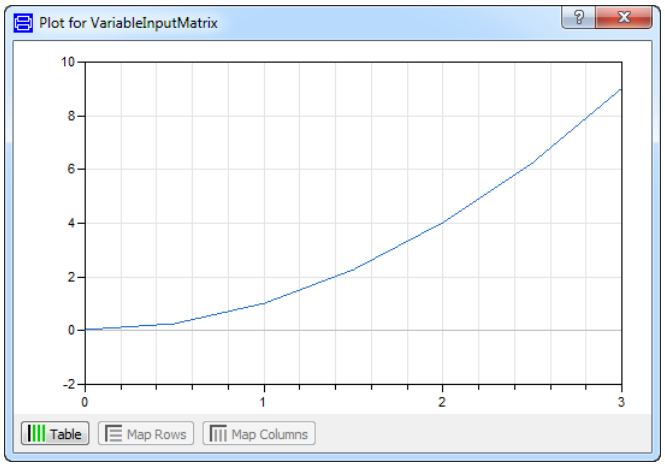
The other way is to use the arrows for rows and columns that are located in the header of the matrix. These arrows expand/shrink the matrix by working on the last row/column. (Increasing the number of rows will create new rows after the last one, decreasing will take away the last row etc.)

The buttons are described in the previous section.

An example of a matrix and the corresponding plot:

Example of matrix and corresponding plot.

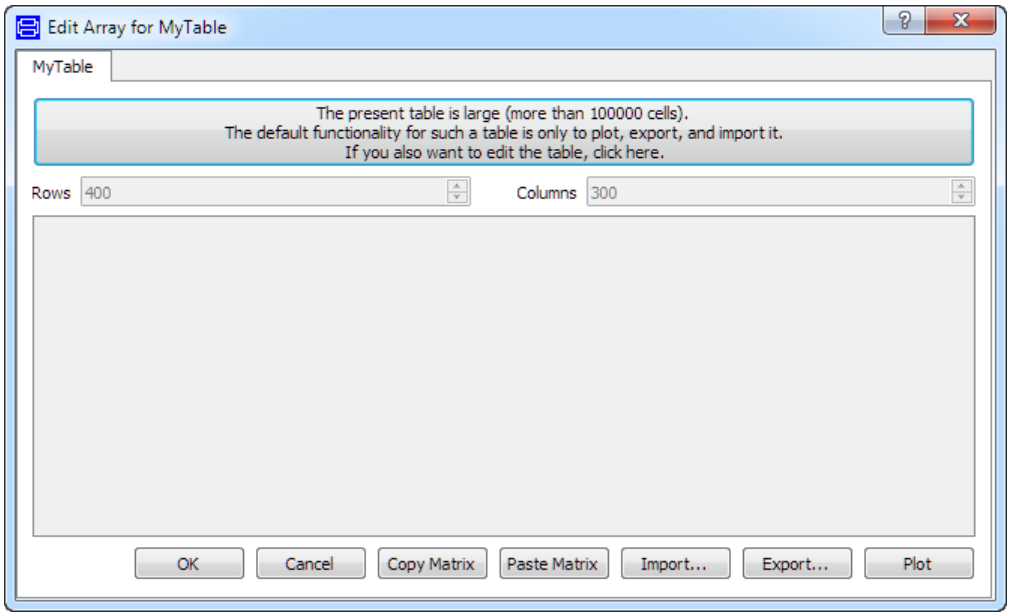




Handling of large matrices (more than 100 000 cells)

If a matrix is large (more than 100 000 cells), the default functionality of the matrix editor is only to import/export and plot the matrix. This improves the handling of large matrices by the matrix editor.

If editing is wanted, a command button has to be clicked (see figure below), and a corresponding warning that this will be a time-consuming operation is given.

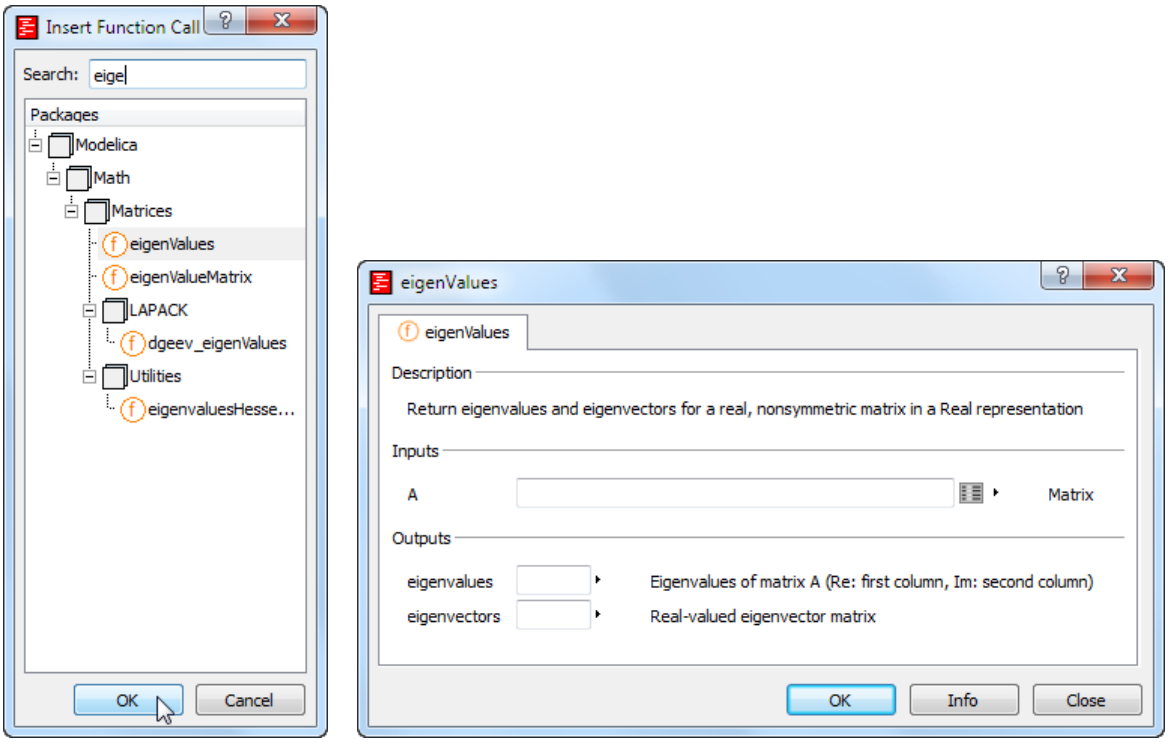


Edit Text, Copy Default and Insert Function Call

Edit Text will show a window with larger input field (line editor). It is needed for modifying e.g. long parameter expressions and re-declarations.

Copy Default copies the text of the default value to the input field for further editing.

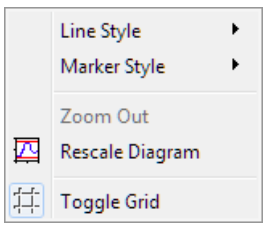
Insert Function Call... enables searching (or browsing) for a function and then entering the arguments, allowing easy insertion of a function call. As an example, entering `eige` in the search input field (in order to find `Modelica.Math.Matrices.eigenValues)` will give the following result (followed by the resulting parameter dialog of the function that will pop when OK is clicked):



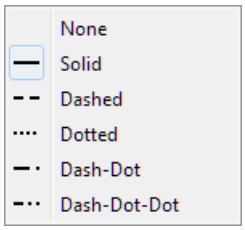
For more information how to use this function, please see section “Handling of function calls” starting on page 294.

4.6.23 Context menu: Parameter dialog; plot window

A context menu is available by right-clicking in the plot window in the parameter dialog:



Line Style has the following alternatives:



Marker Style has the following alternatives:



By using **Zoom Out** the diagram is zoomed out if earlier been zoomed in into by dragging a rectangle.

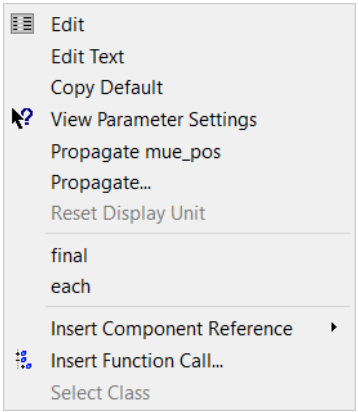
Rescale Diagram will reset the curve to the default size.

Toggle Grid can be used to hide/display the grid in the plot window.

4.6.24 Context menu: Parameter dialog; parameter input field

The parameter dialog is introduced in “Components and connectors” starting on page 235.

The context menu of a parameter input field in the parameter dialog can be displayed either by right-clicking in the input field or by clicking on the arrow button after it.



Note that the commands in the menu differs depending on what type of parameter, below are listed the possible commands.

Note also that code completion by **Ctrl+Space** can be used to display a list of alternatives to select from when working with parameter input fields. See section “Code completion” on page 283 for more information on code completion.

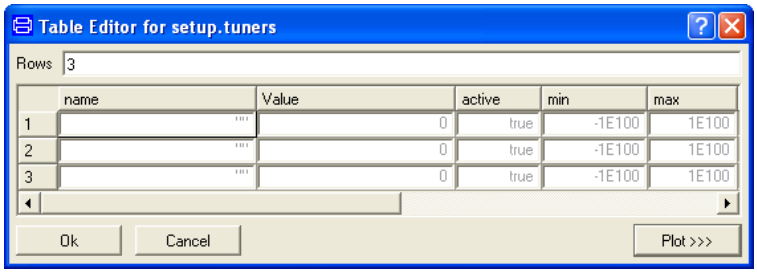
Edit allows hierarchical inspection and editing. This uses a matrix editor for matrices, a parameter dialog for structured modifiers, re-declarations and function calls, and sometimes what the previous edit button showed. For alternatives, please see section “Context menu: Variable declaration dialog; Value input field” starting on page 559.

Edit Text will show a window with a larger input field (line editor). It is needed for modifying e.g. long parameter expressions and re-declarations.

Copy Default copies the text of the default value to the input field for further editing.

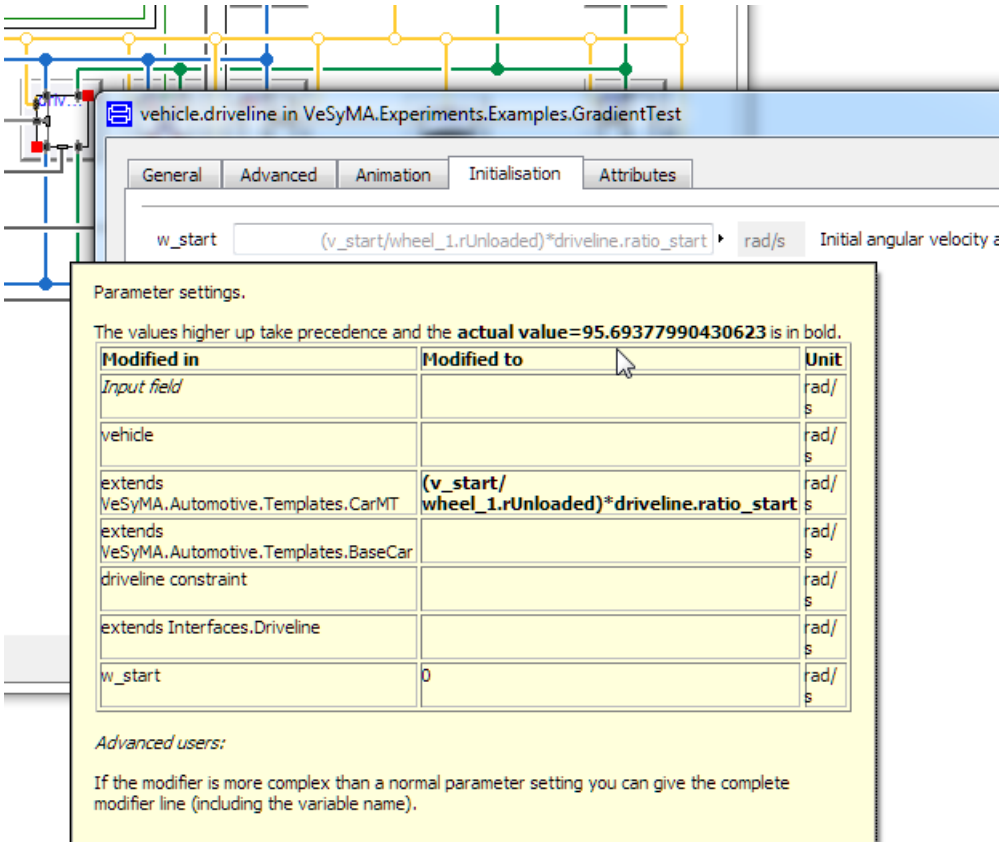
Edit Combined edits an array of calls to functions or record constructors.

Editor for array of records.



View Parameter Settings will show the parameter settings, e.g. the following (yellow) window:

View parameter settings.



The window shows where the default originates from, and all modifiers applied between the original declaration and the current one.

Note that the value is calculated (at the cursor); this helps you to see if complicated formulas give reasonable value without having to simulate, which might be excessive.

To display the above example, do the following in the **VeSyMA** library:

- Open `Experiments.Examples.GradientTest`.
- Right-click the vehicle component and select **Show Components**.
- Right-click the driveline component and select **Parameters**.
- In the parameter dialog, select the **Initialization** tab.
- Right-click the input field of **w_start** and select **View Parameter Settings**.

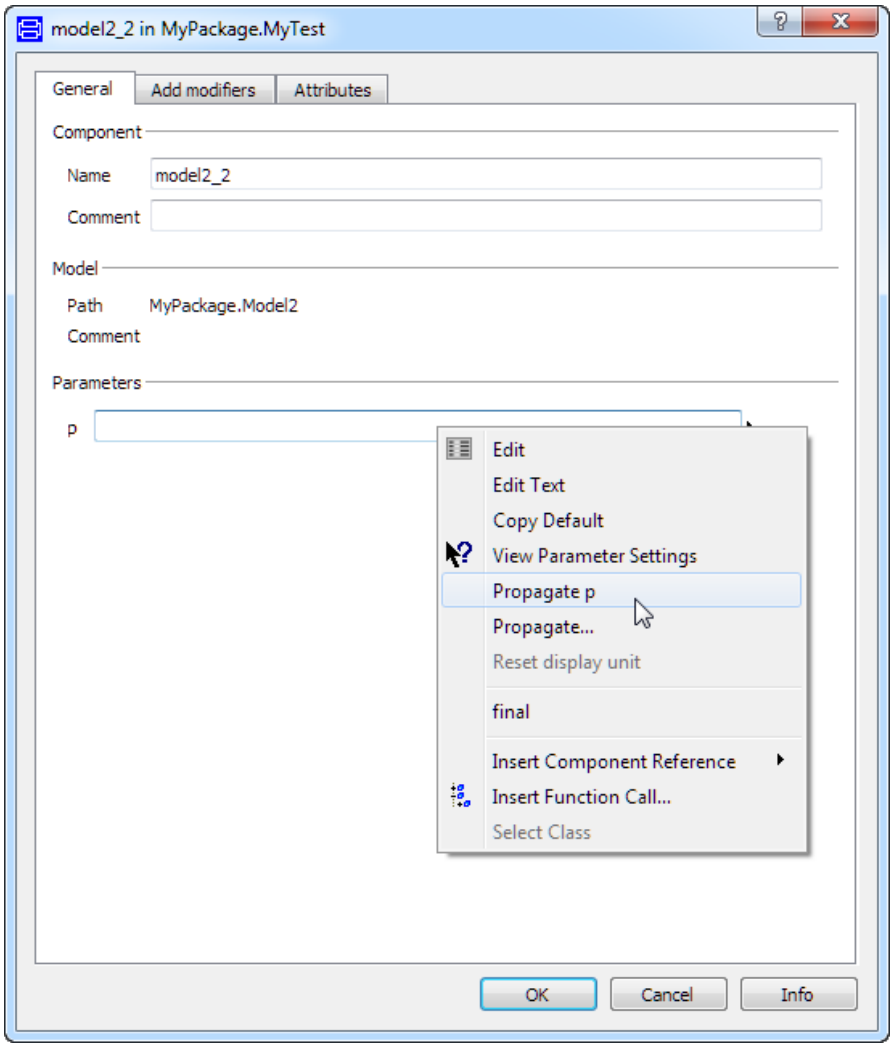
Propagate <*parameterName*> (compare next command) inserts a parameter declaration in the enclosing model and binds the component's parameter to it by introducing a modifier of the type `p=p`. A variable declaration dialog is displayed to declare the new parameter. Default values are not propagated to the new parameter declaration.

It is possible to automatically create the propagations with attribute `final`, that is, the modifier introduced will then always be `final p=p` instead. This is done by setting the flag

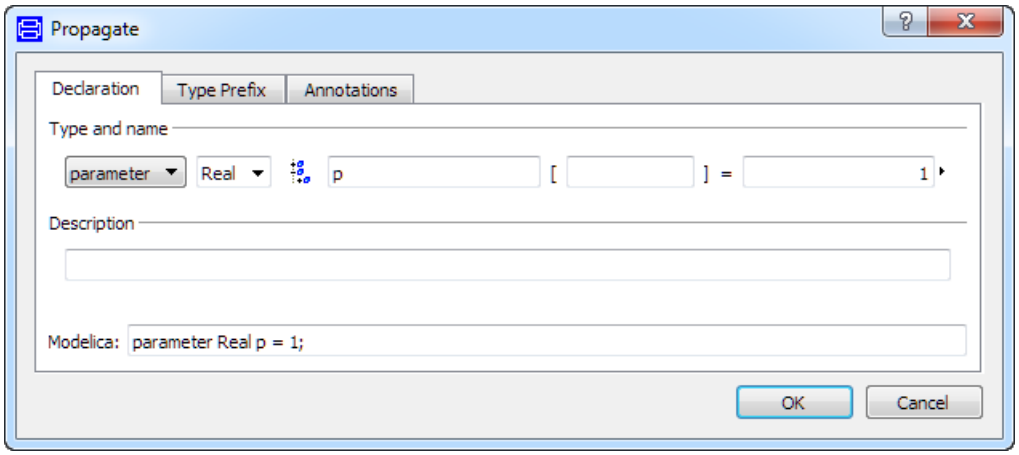
```
Advanced.Editor.FinalPropagate=true.
```

(The default value of the flag is `false`.) To have the propagations as final is often wanted. For more information about the attribute `final`, see the context command **final** below.

As an example of propagation (without using the flag for final), consider the following simple propagation of the parameter p from the component `model2_2` by the **Propagate p** command:



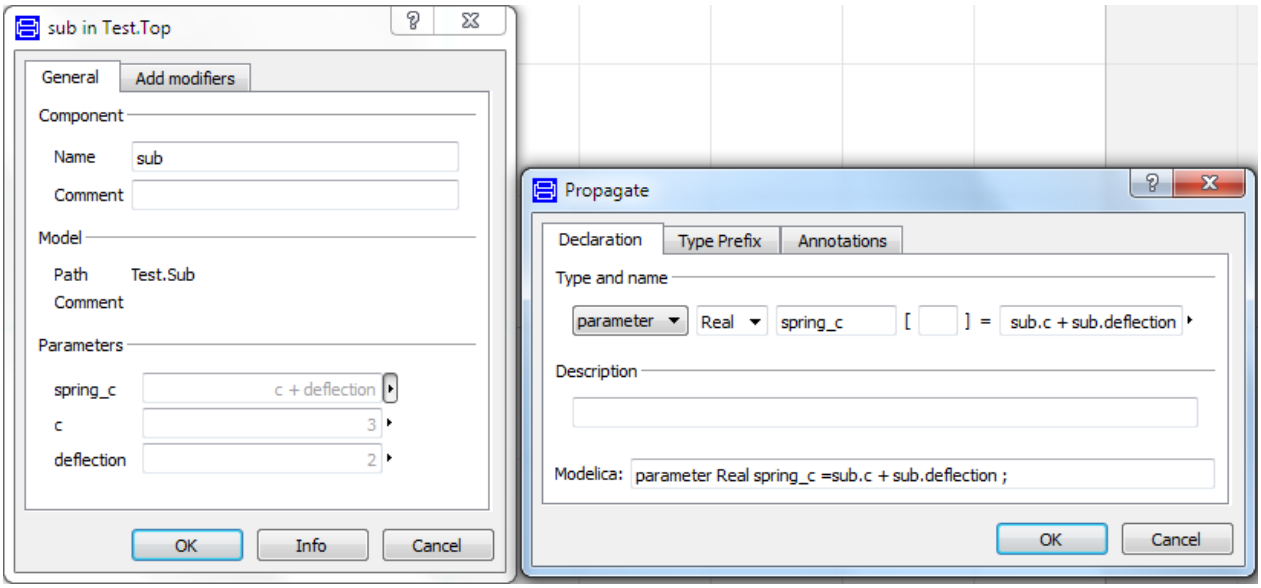
If p is not propagated before, using the name p, from another component of the same class (located in the same enclosing module), the command will open the following variable declaration dialog:



Clicking **OK** will suggest a new parameter p in the model enclosing model2_2, and insert p in the parameter input line of the previous image. The operation is finalized when clicking **OK** in that dialog.

However, if p is already propagated to a parameter p from another component of the same class (in the same enclosing module), the command just adds p in the parameter input line of the previous image. To propagate with a different name, see the **Propagate…** command below.

Propagation of parameters bound to other (local) parameters is handled using hierarchical names of the resulting parameters.



Note that this will cause issues if the local parameters are protected. In that case the default value has to be manually converted to a new scope.

For information about the variable declaration dialog, please see section "Variable declaration dialog" starting on page 378. Note the option to select tab for the GUI of the propagated parameter, see "Tab selection for propagated parameters" on page 388.

Propagate… performs the same action as the previous **Propagate** command, unless if the parameter has already been propagated with the same name previously from another component of the same class (in the same enclosing model). If so, the **Propagate…** command enables changing the name of the parameter to propagate to, by still opening the variable declaration dialog. (This is not the case when using the **Propagate *<parameterName>*** command, see above.)

Reset display unit will reset the selection of display unit if changed.

final If a modifier is final it cannot be modified from an enclosing class. Use **final** if the value should not be modified. It is useful for e.g. propagating parameters (note the possibility to automatically create propagations as **final** by using a flag, see the context command **Propagate *<parameterName>*** above). A checked checkbox will be shown in the context menu if **final** has been selected. A change will however not be executed until **OK** has been pressed. If the component is write-protected, the change will not be implemented.

each For an array **each** means that each element has this value; otherwise the value is split among the array elements. A checked checkbox will be shown in the context menu if **each** has been selected. A change will however not be executed until **OK** has been pressed. If the component is write-protected, the change will not be implemented.

replaceable For a replaceable variable you can use **replaceable** in a redeclaration to keep it replaceable. A checked checkbox will be shown in the context menu if **replaceable** has been selected. A change will however not be executed until **OK** has been pressed. If the component is write-protected, the change will not be implemented.

Quote String Having entered a value for a string parameter, the input can be quoted by this command.

Insert Class Reference inserts a reference to a class in the model.

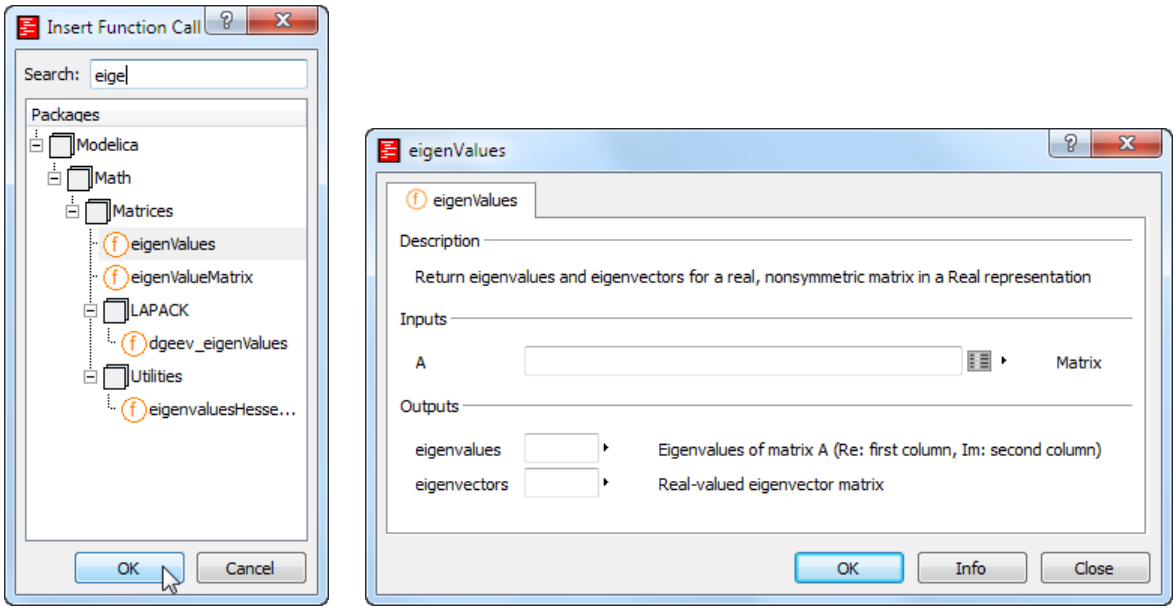
Insert Component Reference inserts a reference to a component in the model at the insertion point of the input field.

Select Class provides a class selector for re-declarations with all matching classes.

Select Record selects a record constructor or constant among suitable candidates.

Any changes become effective when pressing the **OK** button, and are discarded by pressing the **Cancel** button. If the shown model is read-only no changes are possible and only a **Close** button appears.

Insert Function Call… enables searching (or browsing) for a function and then entering the arguments, allowing easy insertion of a function call. As an example, entering `eige` in the search input field (in order to find `Modelica.Math.Matrices.eigenValues` will give the following browser (followed by the resulting parameter dialog of the function that will pop when OK is clicked):



For more information how to use this function, please see section “Handling of function calls” starting on page 294.

5 SIMULATING A MODEL

5 Simulating a model

This chapter describes how to simulate a Modelica model in Dymola. Please also see the chapter “Getting started with Dymola”, which includes several examples.

The content is the following:

In section 5.1 “**Windows available when working with simulation**” starting on page 584 the windows and tabs available in Dymola when working with simulation are described as an overview. More information about the use of some windows and tabs is given in the following sections.

Section 5.2 “**Model simulation**” starting at page 614 begins with the sub-section “Basic steps” that describes the basic steps in setting up a simulation, running it, plotting/animating the results, documenting the simulation and basic scripting. (Please also see the chapter “Getting started with Dymola”, which includes several examples.) The following sections describe more in detail how to perform the basic steps, but also some extended functionality. These sections are centered on how to use the available windows and tab when working with simulation, for browsing signals, displaying model layers when simulating, plotting, animation, scripting and documentation. The section is concluded by a section about simulation settings.

Section 5.3 **“Editor command reference when working with simulation”** starting on page 790 is a reference section, describing the commands available when working with simulation. The menus are ordered in the order they appear in certain tabs and windows, starting with Dymola Main window. The menus of other windows follow, followed by context menus for the windows.

Section 5.4 **"Dynamic Model Simulator"** on page 892 describes Dymosim, the executable generated by Dymola.

Section 5.5 **"Scripting"** starting at page 904 describes the scripting feature. The section is in practice divided in three parts; one describing scripting in general; one describing how to use function calls in scripting and one how to use script files.

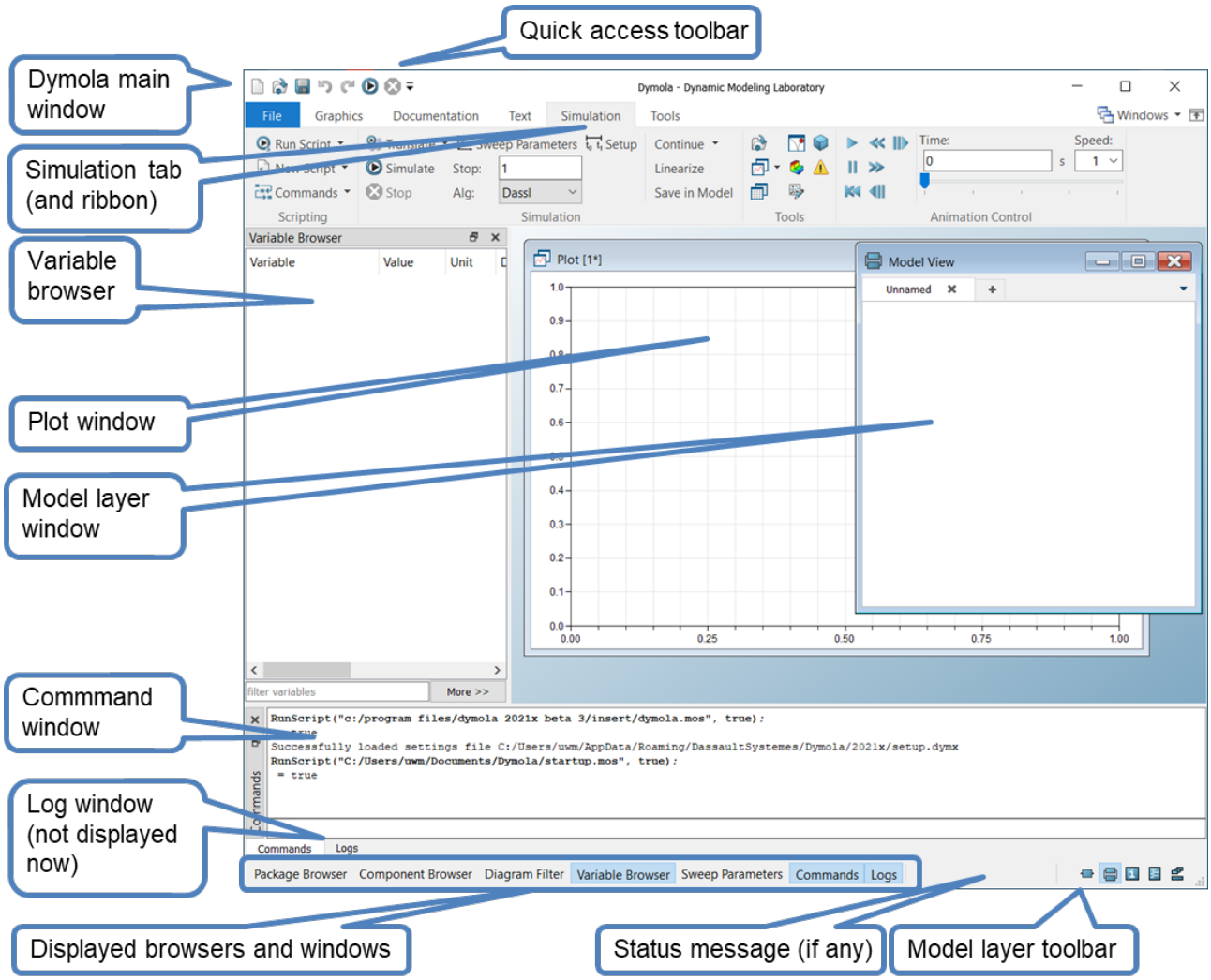
Section 5.6 **"Debugging models"** starting on page 974 lists some important tips for avoiding errors, as well as a number of useful tools how to find errors in the models.

Section 5.7 **"Improving simulation efficiency"** starting on page 1005 describes methods to improve simulation efficiency (dealing with e.g. chattering, profiling and inline integration).

Section 5.8 "**Initialization, steady state, and start values**" starting on page 1024 describes how to handle the **Continue** command, over-specified initialization problems, discriminating start values, how to debug failed initialization, and finally steady state initialization.

5.1 Windows available when working with simulation

A number of window types and tabs are available in Dymola when working with simulation. If you start Dymola and go to **Simulation** tab without opening any model, the Dymola window will look like:



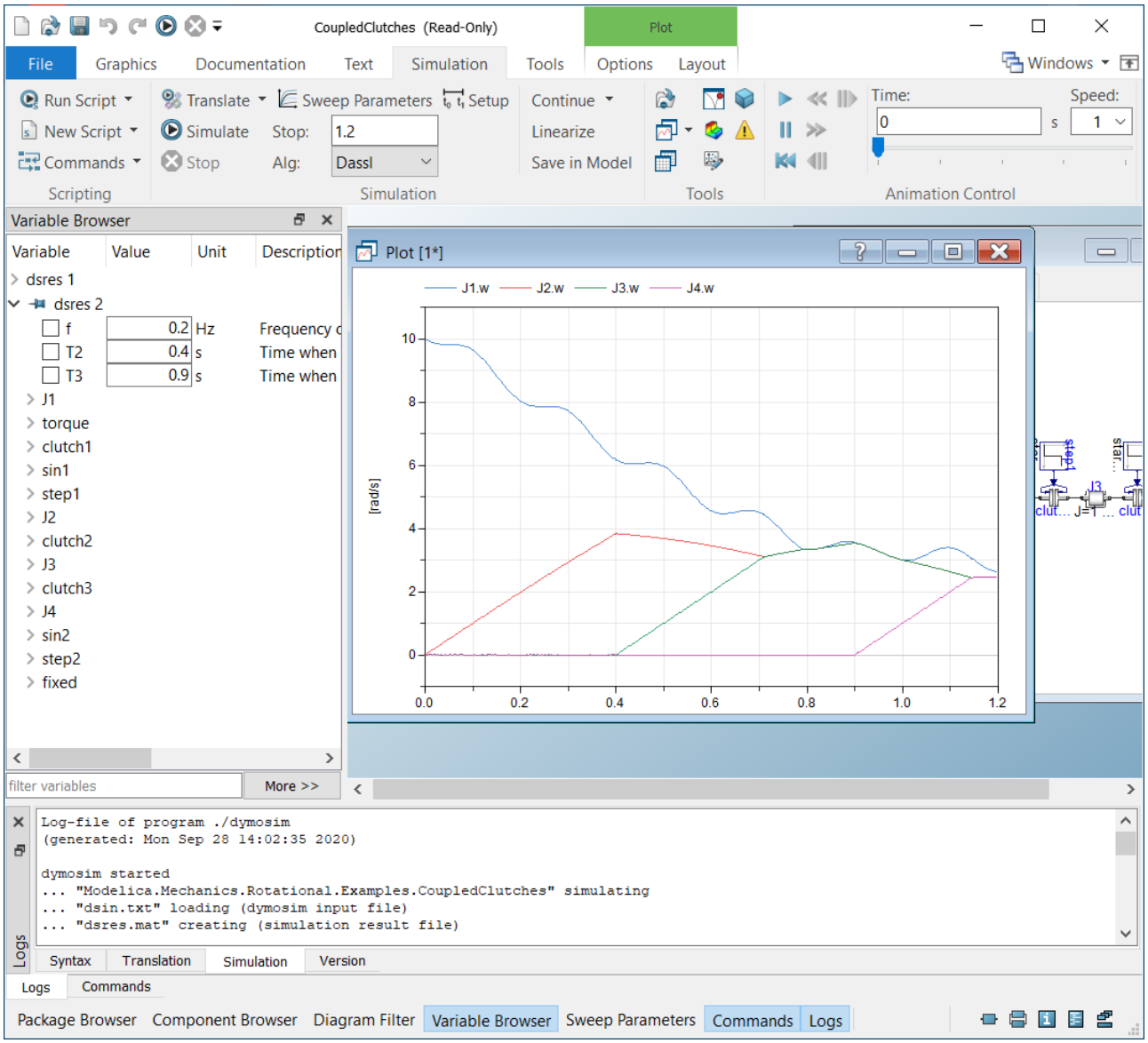
What is seen is the Dymola main window that contains a number of browsers and windows.

5.1.1 Dymola Main window

Default content in Dymola Main window when simulating a model

What sub-windows that are shown when selecting the **Simulation** tab depends partly on the model simulated, e.g. if it is prepared for showing animation. “Default” here should be interpreted as simulating a very basic model without animation etc.

To display a simple example, use the command **File > Demos > Coupled Clutches** to open that demo, select the **Simulation** tab, and perform the command **Commands > Simulate and Plot**. You will get:



Dymola Main window contains by default the following in the **Simulation** tab (see figure above):

- A number of menus and toolbars in the upper part of the window. All menu alternatives and toolbars are presented in the section “Editor command reference when working with simulation” starting on page 790.
- Five sub-windows – one variable browser, one plot window, one model layer window, one command window, and one log window. They are presented below. Note that the by

default both the log window and the command window are present but by default they are docked and only one is on top (active). You can click on any of the **Commands** or **Logs** tab so select which one to be the active one. If displaying the Logs tab, you can select from a number of logs. In the figure above, the simulation log is displayed. For more information about the log window, see section "Log window" on page 603:

- A quick access toolbar in the upper left corner of the Dymola main window to enable quick selection of, for example, the simulation command. What commands that are presented here can be configured. For more information, see "Shortcut to Simulation tab commands in the top left toolbar" on page 791.
- A model layer toolbar in the lower right corner of the Dymola main window, to select what layer to present in the model view window. For more information, see "Displaying different model layers in the Model View window" on page 689.

Additional content available

Additional content available depends on:

- The model being simulated; if it contains animation or visualization, animation or visualization windows will appear when the model is being simulated.
- Command selection, for example, scripting commands opens a script editor window.
- User selection of window; for a number of windows additional tabs opens if you work in a window:
    - Working in a plot windows opens the **Plot** tab. See "Plot window and Plot tab" on page 591.
    - Working in an animation window opens the **Animation** tab. See "Animation window and Animation tab" on page 596.
    - Working in a script editor window opens the **Script Editor** tab. See "Script Editor window and Script Editor tab" starting on page 602.
- User selection of content; by clicking the buttons in the lower part of the staus bar at the bottom of the Dymola window, the content of the Dymola window can be changed. Features by default only available in other tabs can be added. This selection is individual for each ribbon tab. The settings are saved between sessions.
- Progress of translation/simulation. If errors are detected, they are displayed in the log window.
- When using any **Info** button or **Info** command an information browser appears.

Shortcut keys to tabs

For the tabs, you can use the following shortcut keys, listed in tab order (the first shortcuts are for developing a model, see that chapter for more information about them):

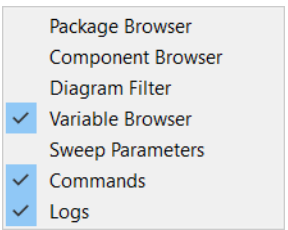
Note that you can only go to tabs visible, that is, you can, for example, never be in the **Graphics** tab and go to the **Plot** tab by **Alt+P**, the **Plot** tab is not visible when in the **Graphics** tab.

Shortcut keys	Action
Alt+F	Displays the **File** menu
Alt+G	Displays the **Grapics** tab
Alt+D	Displays the **Documentation** tab
Alt+T	Displays the **Text** tab
Alt+S	Displays the **Simulation** tab
Alt+O	Displays the **Tools** tab
Alt+P	Displays the **Plot** tab
Alt+A	Displays the **Animation** tab
Alt+C	Displays the **Script Editor** tab

Context menu

Right-clicking in an empty space in the ribbon part section of the tab will display a menu where the content of the main window (available sub-windows) can be changed.

Context window of ribbon part in Simulation tab.



5.1.2 Variable browser

General

The variable browser displays a tree view of all variables available. Each sub-model represents one level of the hierarchy (a “node” in the browser), and all variables of a sub-model instance are grouped together. Components are by default sorted according to order of declaration in the enclosing class. Clicking on the header line toggles alphabetical order.

The top-level nodes in the variable browser represent simulation result files; other nodes represent the component hierarchy and variables and parameters at the lowest level.

Vector/array elements are collected under expandable vector/array nodes .

What variables are available in the variable tree can be controlled by programmer using annotations for variable selections.

Concerning plotting, if more than one variable is selected by the user for plotting, they will be plotted in a single diagram with a common scale. Variables that contain data for 3 D animation are not shown.

Display units for signals can be selected in the variable browser, if display units are defined for the signals, and if the signals are editable.

Modified initial conditions and parameter values entered in the variable browser when working with simulation can be saved to the model.

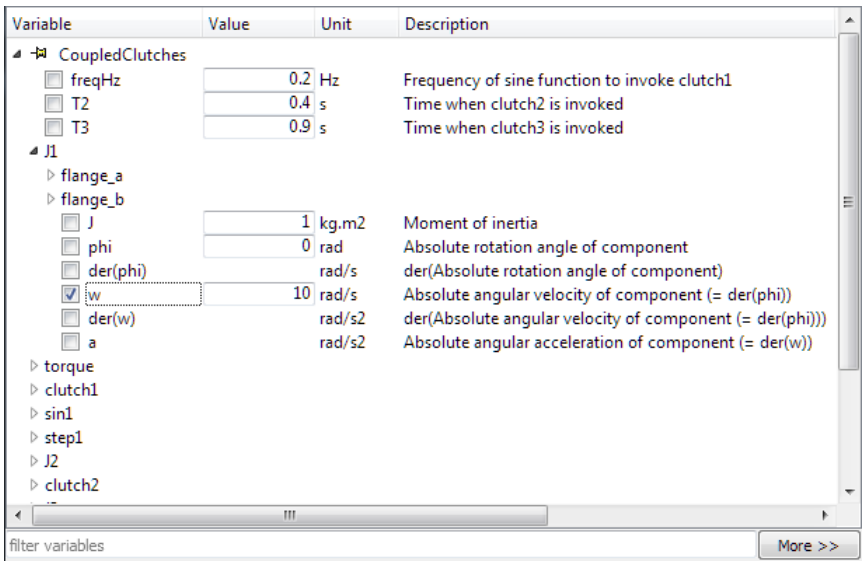
Dependencies, equation incidence, numeric integration, event logging, and timers can be analyzed.

Filtering variables can be done using the bottom line in the variable browser.

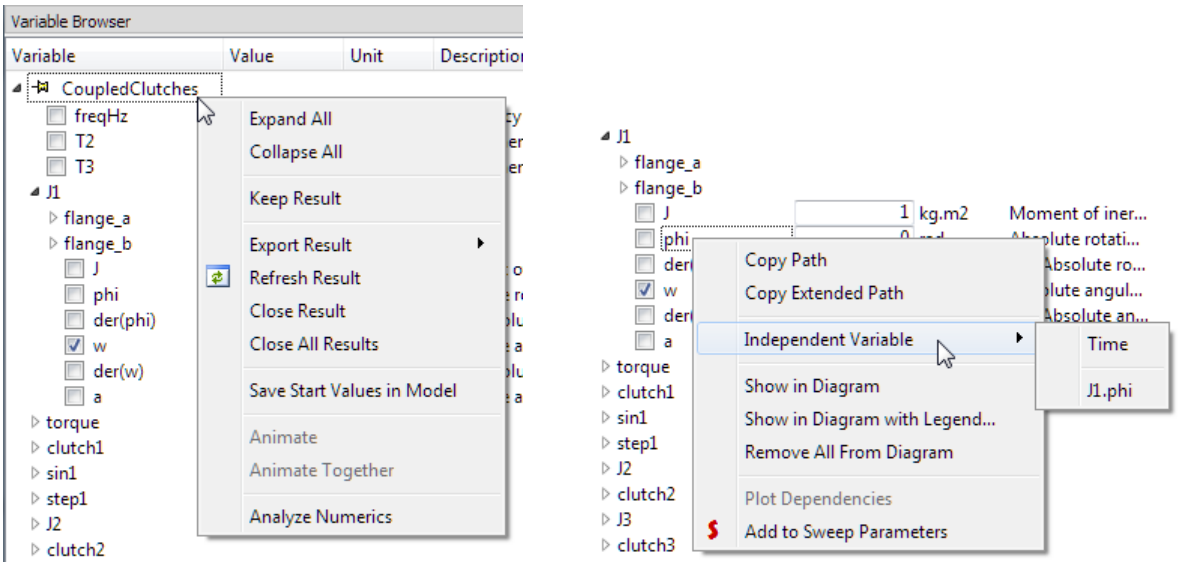
An advanced mode gives the possibility to, for example, display changed variable values and synchronize values in the variable browser with the animation time.

Please see the section “Variable browser interaction” on page 619 for more information.

The variable browser (expanded to show the description).



By right-clicking context menus can be used. The context menus are different depending on where the cursor in when activating the context menu. The menu to the left below is an example of the context menu of the top level of the variable browser, the one to the right is a context menu of a signal. Please see the sections “Context menu: Variable browser – nodes” and “Context menu: Variable browser – signals” on page 869 and 873, respectively, for more information.



Browser handling



The variable browser can be undocked using the **Undock** button or by double-clicking on the browser header or dragging it.

To dock the browser again, the best way is to double-click on the browser header. It can also be dragged to its original location (you will then see how space for it is created automatically before releasing it).

It is possible to dock also to the right side of the main window.

If you have displayed a diagram filter browser, you can have the variable browser and the diagram filter browser as two tabs by dragging one on top of the other.

Note that you can dock and undock also the command window, but you cannot dock it to the browsers.

5.1.3 Model View window

A Model View window containing the diagram layer of the model is by default opened when you enter the **Simulation** tab the first time.

The diagram can be used for selecting what should be visible in the variable browser and/or for presenting and changing values. You can navigate in the instance hierarchy of the diagram. The diagram can also be filtered.

You can also display other model layers (Modelica text, icon, documentation and used classes) of the selected object in the diagram. You do this by using the model layer shortcut buttons in the lower right corner of the Dymola window.

Notes

- The window header changes name to indicate what layer is displayed. “Model View” corresponds to “diagram layer”.
- All these layers are read-only when displayed in the Model View window.

For more information for how to use the diagram layer in the Model View window to select what variables to show in the variable browser, see section “Using the diagram layer in the Model View window to select what to show in the variable browser” on page 620.

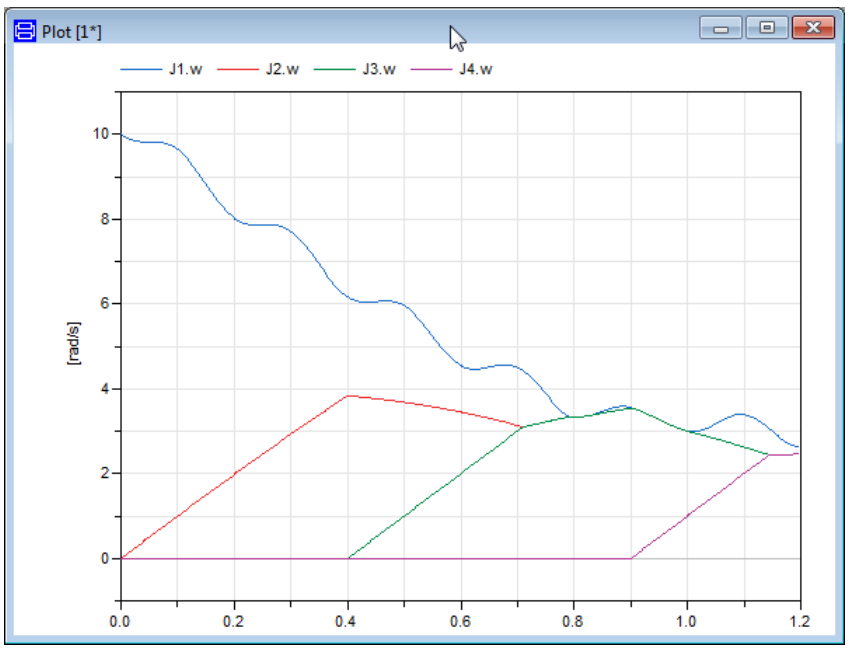
For more general information about the Model View window and how to use it, see section “Using the Model View window” starting on page 679.

5.1.4 Plot window and Plot tab

Dymola supports plotting of any variable. Multiple plot windows may be created. Each plot window may contain several diagrams. Multiple curves in each diagram are allowed. Multiple diagrams in a plot window allow the user to lay out the curves nicely with aligned time axis and different heights and y-scales of the diagrams.

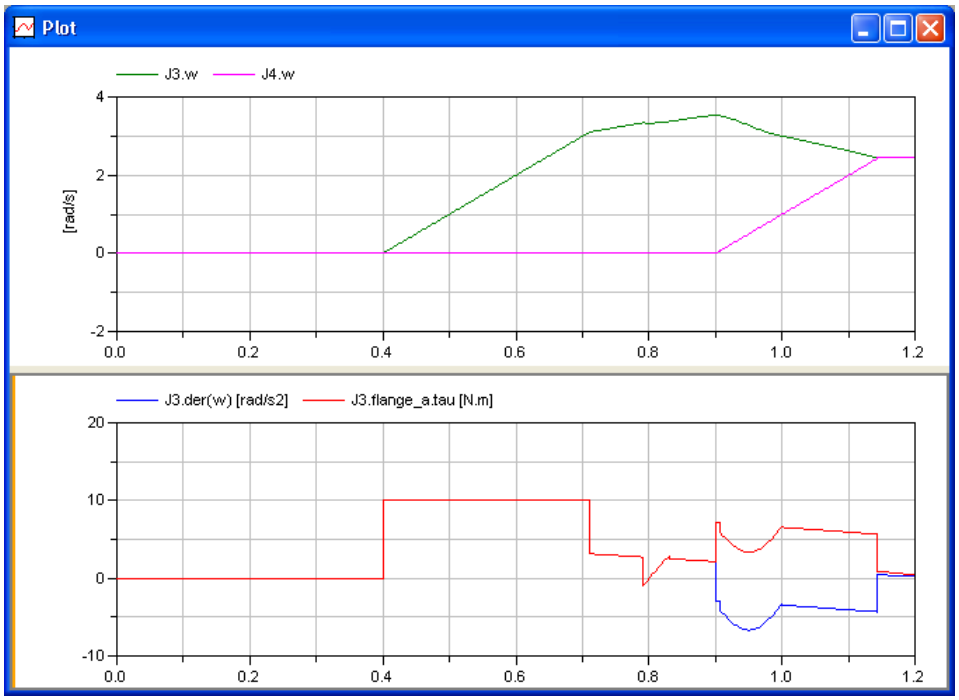
The default layout of plot window displays the names of plotted signals in a legend above the diagram, and the unit of the signals (if available) along the vertical axis.

Default plot window.



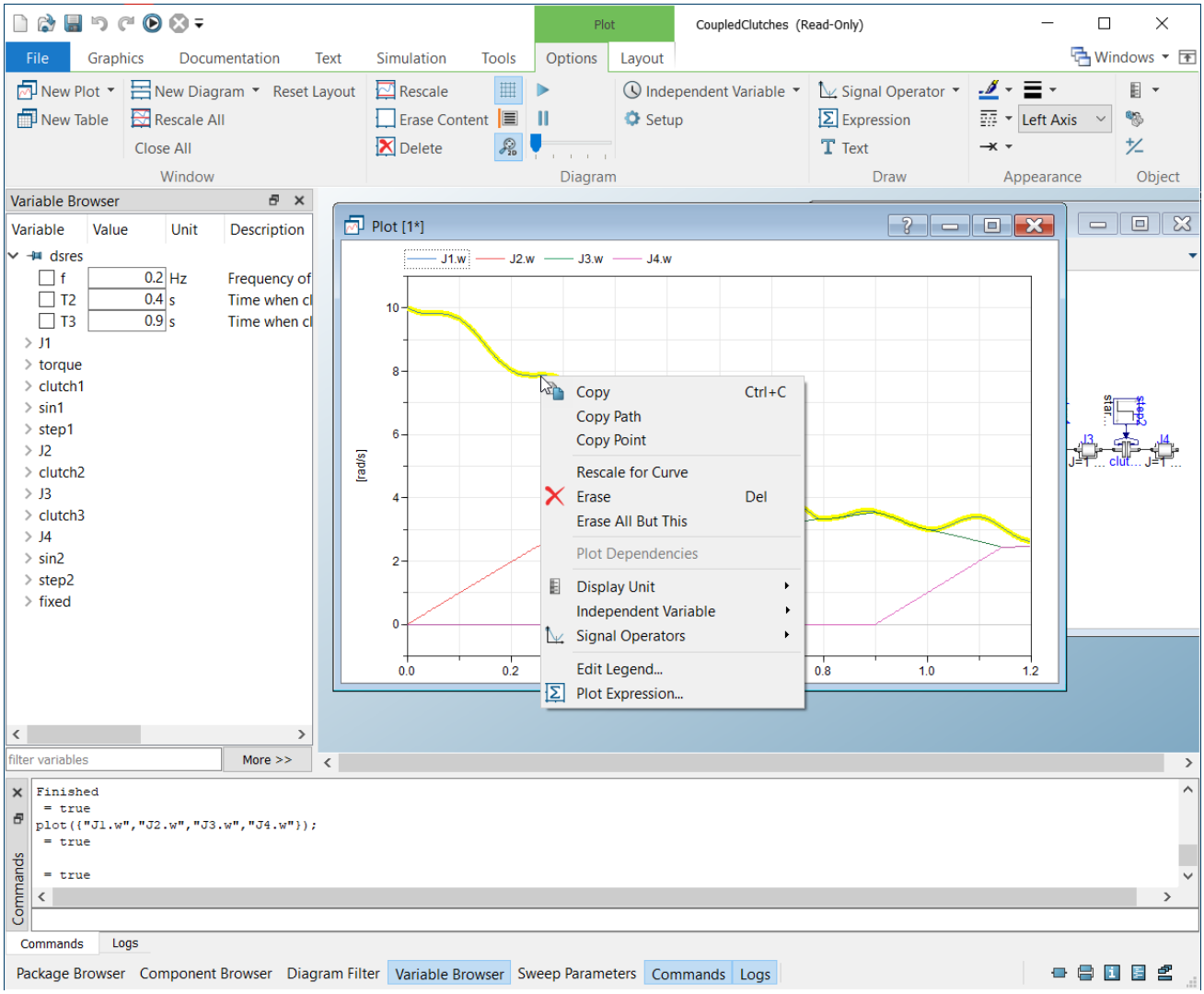
When plotting, there must be an active plot window. If the active plot window has multiple diagrams, one of the diagrams is active, indicated by an enclosing grey rectangle, and a tiny orange bar to the left. The plot commands are directed to this diagram. In the figure below the lower diagram is active.

A plot window where the lower diagram is active.



You may have one or several plot windows in the Dymola main window, in the **Simulation** tab, like in the first figure in section “Default content in Dymola Main window when simulating a model” on page 585 above.

However, as soon as you start working in a plot window (for example, selecting a curve), the **Plot** tab is activated to display commands for working with the plot:



The **Plot** ribbon tab has two subtabs, **Options** and **Display**. In the figure above the **Options** subtab is active. This subtab contains all plot commands except the ones for managing 2D plot layout changes; this tab is opened by default. The **Display** subtab is for working with 2D plot layout, and contains commands to, for example, merge and split diagrams in the plot window.

As you can see in the figure above, you also have a number of commands available in context menus; in the above example, the context menu for a curve when right-clicking it.

You can use the commands in the ribbon and the context menus to do, for example, the following:

- Display several diagrams and work with them in the same plot window, in one column or as 2D layout.
- Display Boolean and enumeration signals, separated or overlapping.
- Display curves negated.
- Duplicate diagram to new plot window.
- Display dynamic tooltips for signals (and copy the values to clipboard if needed).
- Display tooltip for the legend to see more information about the corresponding signal.
- Display tooltip for the x-axis and y-axis.
- Move and zoom in the window.
- Apply time sliding window.
- Change time unit on x-axis.
- Change the display unit of a signal.
- Using a measurement cursor.
- Drag and drop curves between plots/diagrams.
- Drag and drop curves between plots and tables.
- Lock plots and tables.
- Add a y-axis to the right (secondary axis).
- Assign different colors to curves from different simulation result files
- Display signal operators.
- Plot general expressions.
- Plot 1D discretizations.
- Plot dependencies.
- Select multiple curves in a diagram (for e.g. copying the curve values to Excel).
- Display a table instead of curves.
- Display constant arrays as matrices.
- Create tables from curves.
- Apply color-coding to simulation result file tables.
- Plot parametric curves.
- Create scatter plots.
- Change color, line/marker style and thickness of a signal.
- Plotting bar charts and area charts.
- Plotting of external files.

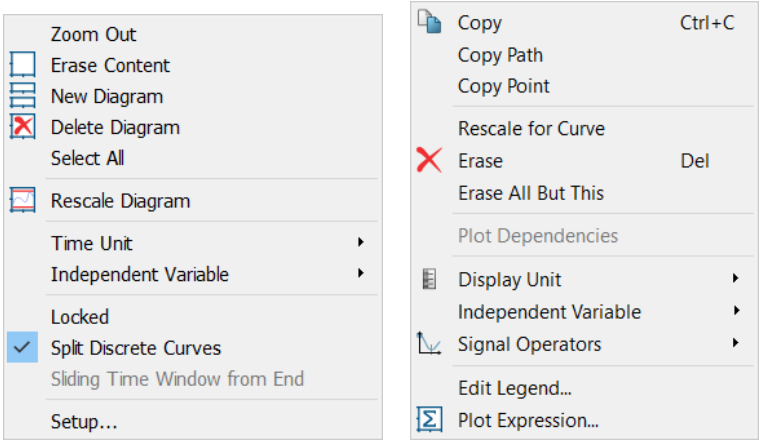
- Adding titles and documentation to plot windows.
- Set diagram heading and axis titles and ranges.
- Insert and edit text objects in the plot.
- Change the appearance, layout and location of the legend. (The legend can be located in several different places, for example inside the diagram.)
- Display the component where the signal comes from in a diagram layer.
- Go back to previously displayed plot window.
- Insert plot and its corresponding command in command window.

Please see section "Plot window interaction" starting on page 692 for more information on interaction in the plot window and plot tab.

For an overview of where to find information in the manual concerning plot functionality, please see "Plotting" starting on page 690.

For information about the tab ribbon commands of the two subtabs of the plot tab, see the sections "Main window: Plot tab: Options subtab" on page 839, and "Main window: Plot tab: Layout subtab" on page 858.

A number of different context menus are available. Below two examples, the context menu that is the result of right-clicking in an empty area, and (to the right) right-clicking on a curve or legend.

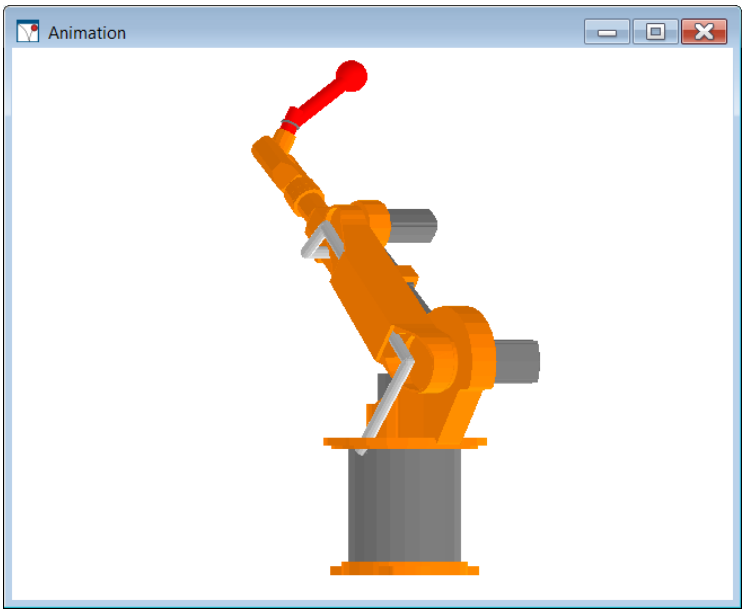


These context menus are described in the sections "Context menu: Plot window" on page 874 and "Context menu: Plot window – curve and legend" on page 876. Other context menus related to plot features are described after them (e.g. the context menu for a text object in a plot, context menus for signal operators and parametric curves, context menu for table window).

5.1.5 Animation window and Animation tab

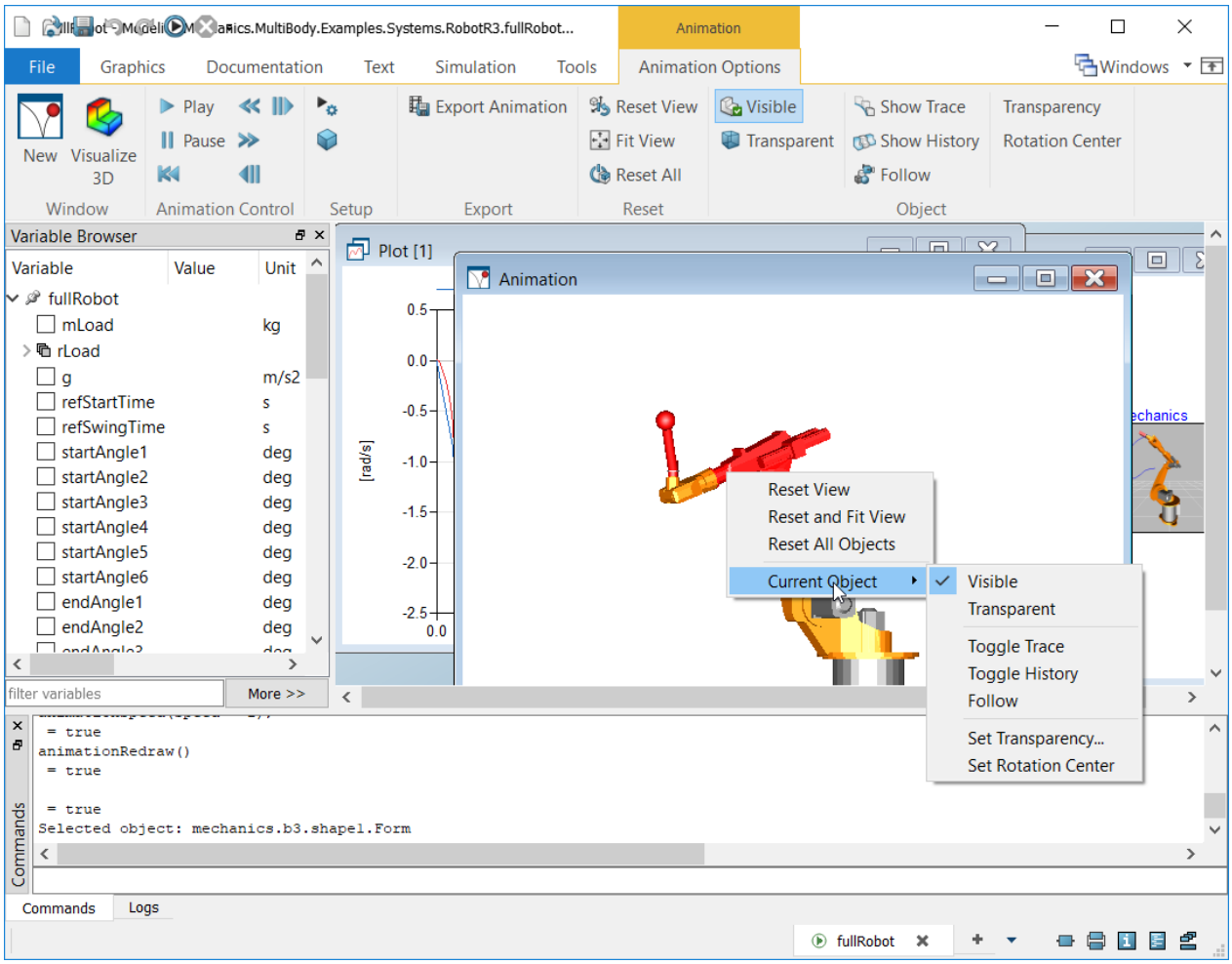
Dymola supports a 3-dimensional model-view built up by graphical objects such as boxes, spheres and cylinders, rendered with shading and hidden surfaces removal.

An animation window.



You may have one or several animation windows in the Dymola main window, in the **Simulation** tab.

However, as soon as you start working in an animation window (for example, selecting an object in the window), the **Animation** tab is activated to display commands for working with the animation:



Visual modeling

Dymola supports visual modeling in addition to dynamic modeling with a library of graphical objects. When a model class is described in Dymola with equations and sub-models, it is also possible to define its visual appearance. This is done by including predefined graphical objects of various shapes. Any model variable can be used to define the changes of the visual appearance.

Graphical objects

Examples of supported 3D graphical objects are: Box, Sphere, Cylinder, Cone, Pipe, Beam, Gearwheel, Spring and Vector arrows

Parameters such as size can be specified. Coordinate systems can be defined by a complete 3-dimensional transformation (3x3-matrix+translation). The information can either be parametric or depend on the dynamics of the model.

For more information about the graphical objects, please see section “Defining Graphical Objects for the animation window” starting on page 757.

More on animation

For details about animation, see section “Animation” starting on page 757.

Interaction

Please see the section “Animation window interaction” starting on page 760.

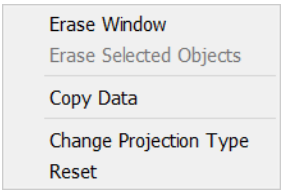
For the commands of the Animation tab, see section “Main window: Animation tab” starting on page 860.

A context menu is available in the window, as seen in the figure above. For more information about this menu, please see section “Context menu: Animation window” on page 880.

5.1.6 Visualizer window

The visualizer window is part of the Plot 3D package, a part of the Visualize 3D concept. For more information about this concept (including how to build and handle objects) please see the chapter “Visualize 3D”.

A context menu is available in the visualizer window:

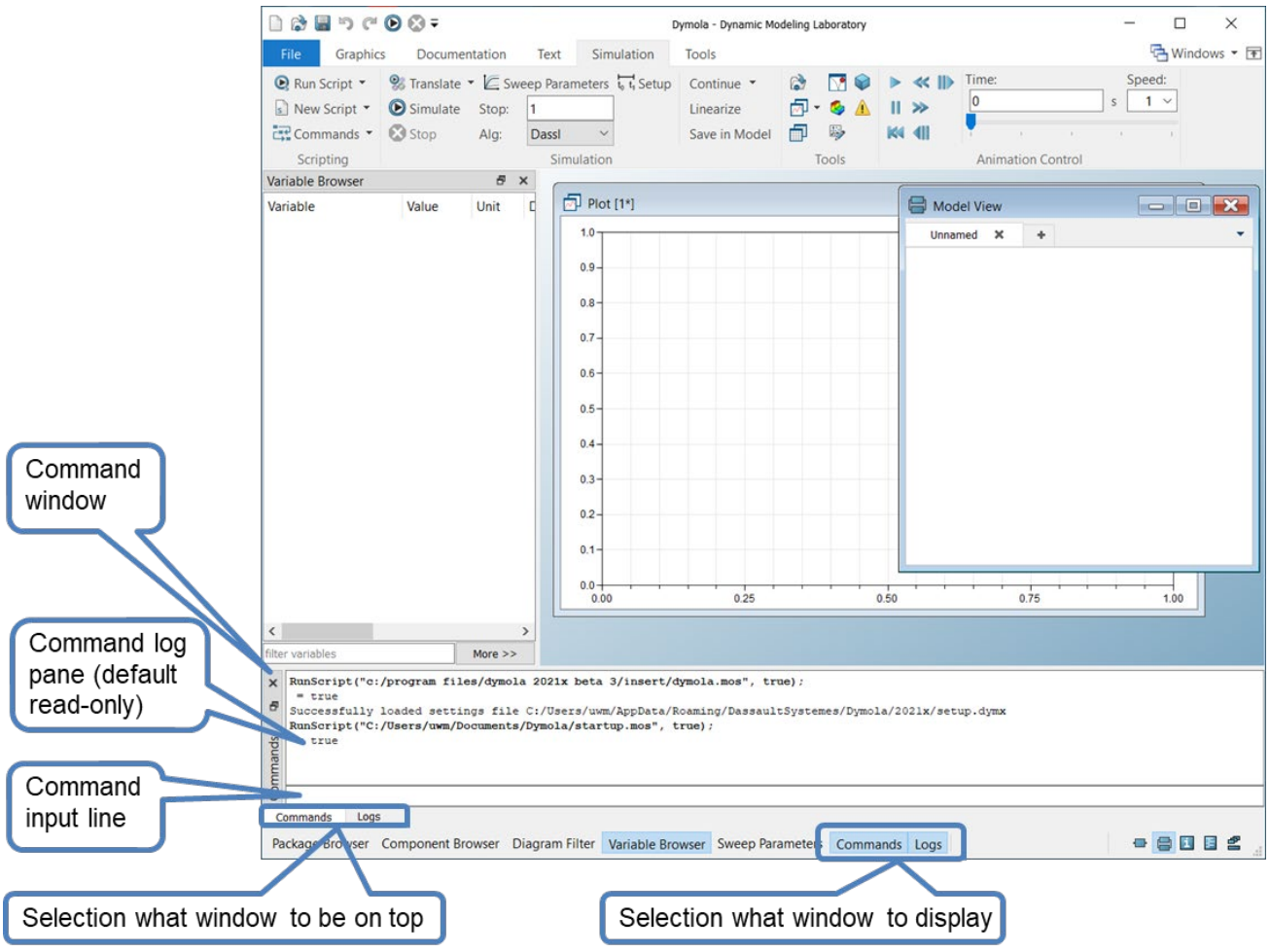


For more information about this, please see the section “Context menu: Visualizer window” on page 882.

5.1.7 Command window

General

In the **Simulation** tab the command window is available by default:



However, it might not be active (on top); the Logs window might be on top instead. To activate the command window, click the **Commands** tab just under the command input line (see figure above). **Note** that to be able to see it at all, it must be selected to be displayed (blue) in the status bar in the lower part of the Dymola window, see figure above.

The command window contains the command log pane displaying the command log and a command input line, where commands can be entered.

The command window can be displayed also in other tabs than the **Simulation** tab, and as a separate window as well. Please see section "Displaying the command window" starting on page 764 for more information about this.

The command log pane

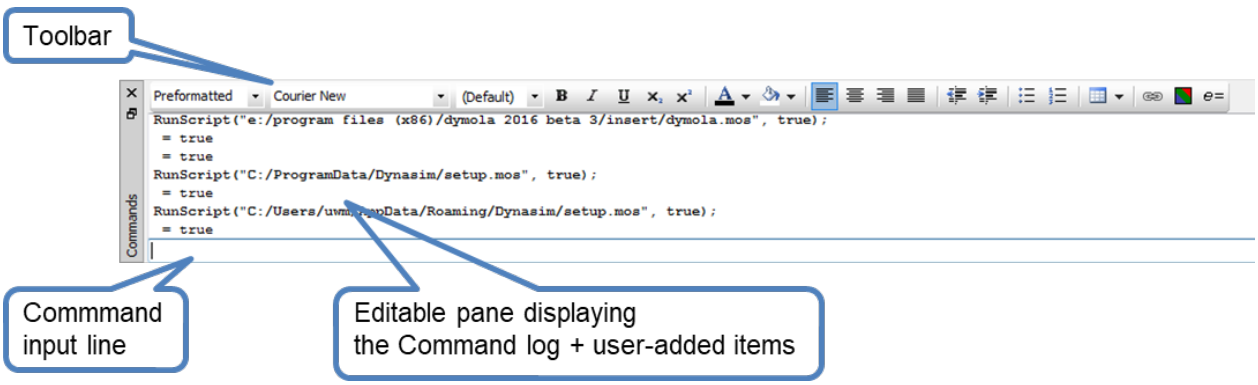
The command log pane can be used in two modes.

Default mode (read-only command log pane)

By default, the command window shows the command log in a read-only mode (as in the image above). The command log contains commands given and the result of these commands. A valuable advantage is that the command log can be saved as e. g. a simple list of commands given (to be used as a script file). This is possible by using the different options for saving the command log (see next section below).

Documentation mode (editable command log pane)

Right-clicking the command log pane and activating **Show Toolbar** turns the command log pane into an editor that can be used to document the simulations. You can add texts, links, images etc. using the commands in the toolbar:



In this mode, the command log pane reflects looking at the command log as a documentation resource, enabling the creation of documentation of simulations containing also formatted descriptions of e.g. simulation results, plots, images, links etc – much more a documentation of a simulation than a pure command log. Math rendering of equations and formulas (with indexing) and rendering of Greek letters is also possible. Of course, it is possible to copy sections of this editor to e.g. Microsoft Word for further work.

More information about the documentation editor in this mode is given in section "Working with documentation in the command window" starting on page 771.

The command log file

The command log is what is displayed in the command log pane of the command window. The command log can be saved to a file using e.g. the command **File > Save Log…**).

The saved command log file can be of three types:

- A list of given commands and the result of these commands.
- A documentation of simulation according to previous section.
- A list of given commands without results that can be used as a script file.

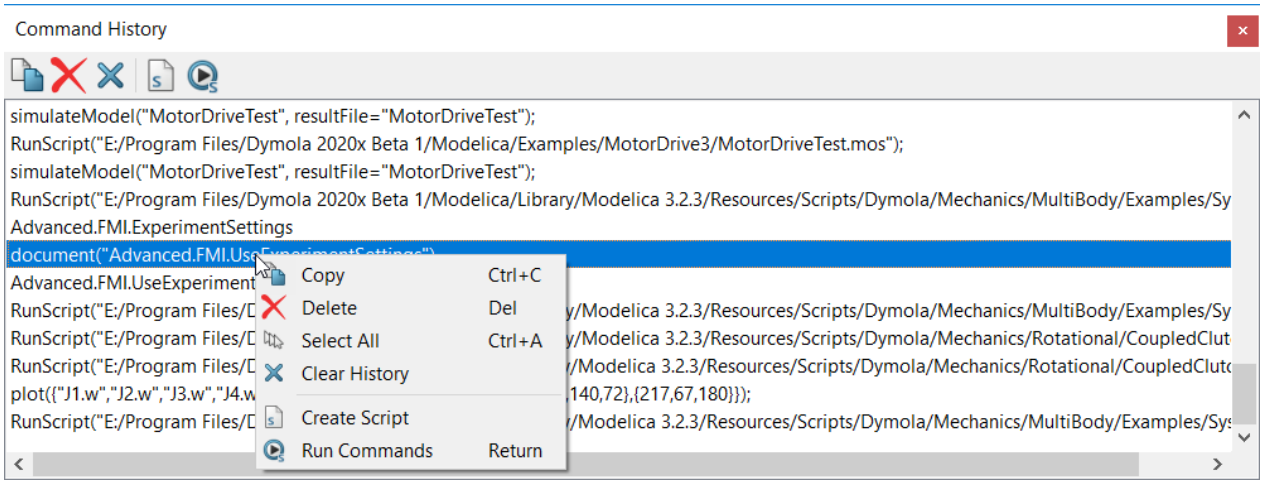
The reason that the saved command log file can have different content is that the content depends on what file format the file is saved. For more information, see section "The command log file" starting on page 767.

The command input line

The command input line is referred to in many places in this manual as a way of manually entering commands (e.g. setting flags). But it can be of more value than that when used for scripting. More information about the command input line is given in section "The command input line" starting on page 768.

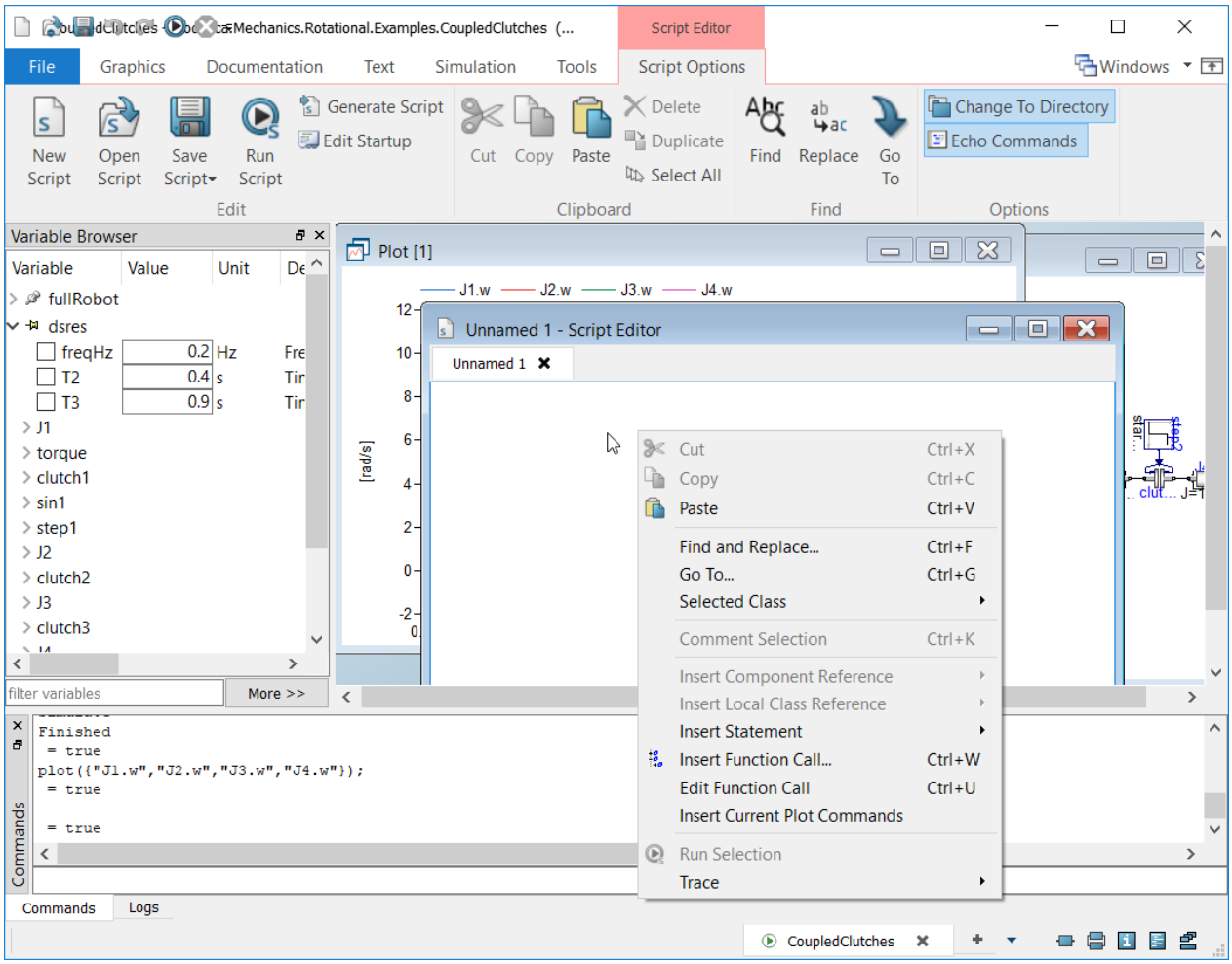
The command history window

The command history window can be displayed from the command input line. It can exist as a framless window displaying only the command history, or it can be displayed as a separate command history window with a number of options. See section "Working with the command history" starting on page 780.



5.1.8 Script Editor window and Script Editor tab

The Dymola script editor is displayed when you want to create a new script or open an existing script by the commands **Simulation > New Script**, or **Simulation > Open Script**. At the same time as the script editor opens, the **Script Editor** tab appears as well:



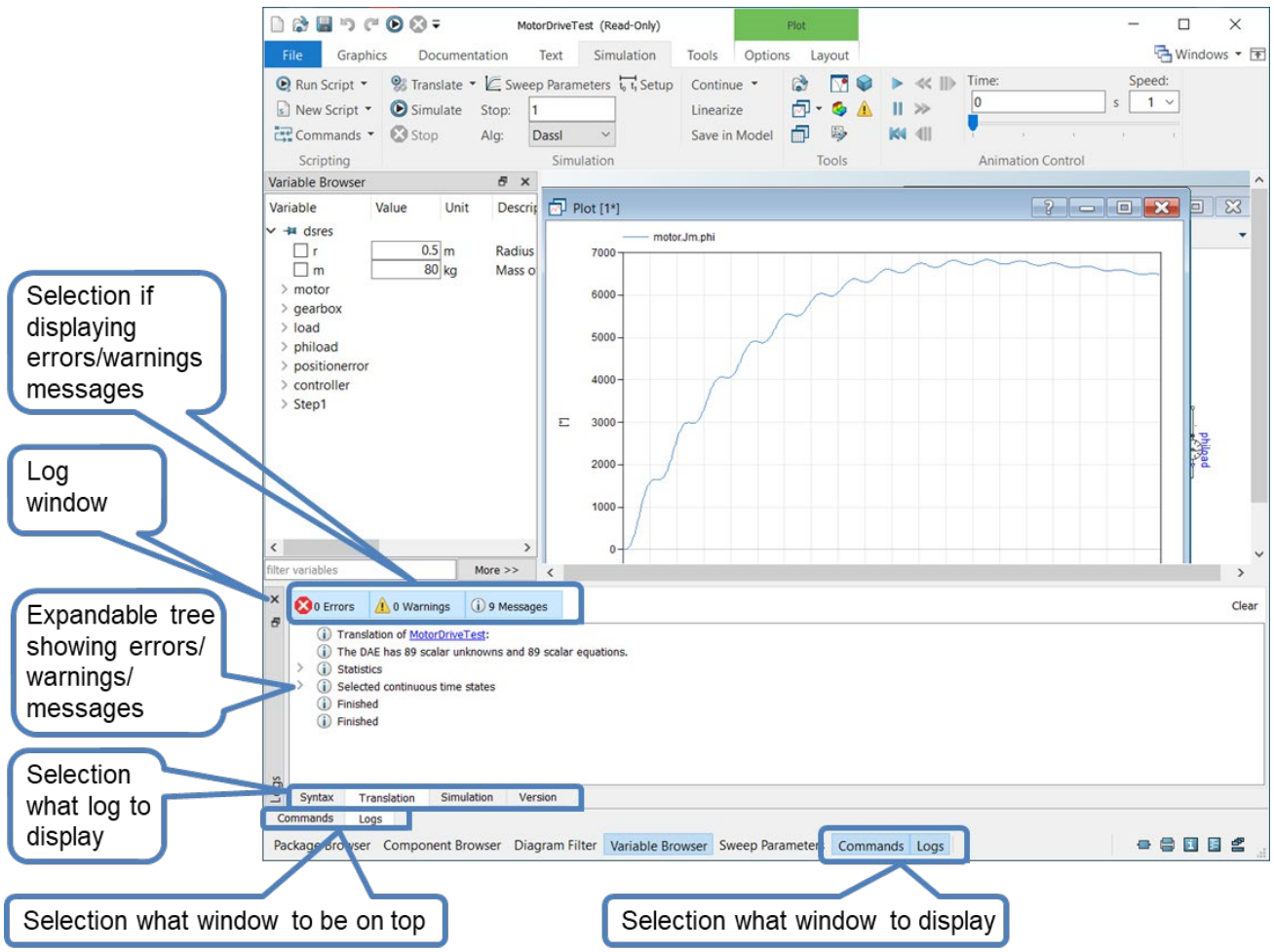
Several scripts can be handled in the same script editor window using tabs. A script or a selected part of a script can be executed. Global variables, function calls, and variables in functions can be traced.

Please see section “The Dymola script editor” starting on page 926 for more information.

By right-clicking in the script editor window, a context menu is available, as seen in the image above: Please see section “Context menu: Script editor” on page 885 for more information.

5.1.9 Log window

The log window is present by default.



However, it may not be displayed if it is docked with the command window, and the command window is active (on top). If not active, you can click the Logs tab just under the list of the different logs (see figure above) to activate it. **Note** that to be able to see it at all, it must be selected to be displayed (blue) in the status bar in the lower part of the Dymola window, see figure above.

Note that in many cases it might be good to see both windows, it may be a good idea to undock the log window by double-clicking the vertical header bar or by clicking in the vertical header bar. You can also drag the header. Below figures will be of undocked windows. To dock the window again, you can drag the window header or double-click it.

The log window has a number of tabs that can be used in different situations.

Syntax Error tab

The **Syntax Error** tab displays syntactical errors generated when reading incorrect Modelica source files or when writing incorrect Modelica code in the text layer of Dymola.

Consider the following example of a syntactically incorrect model (a semi-colon is missing after `der(x)=-a*x`):

```
model MyModel
  parameter Real a=5;
  Real x(start=1);
equation
  der(x)=-a*x
end MyModel;
```

Check the model by pressing the **Check** button. The check detects that the syntax is incorrect and an error message will be displayed in the Syntax Error tab. Syntax errors are also detected when syntax highlighting is performed by taking up the context menu by right-clicking and selecting **Highlight Syntax,** or by pressing **Ctrl+L**. Syntactical errors are also detected when trying to switch layer, for instance by pressing the Diagram layer button.

Translation tab

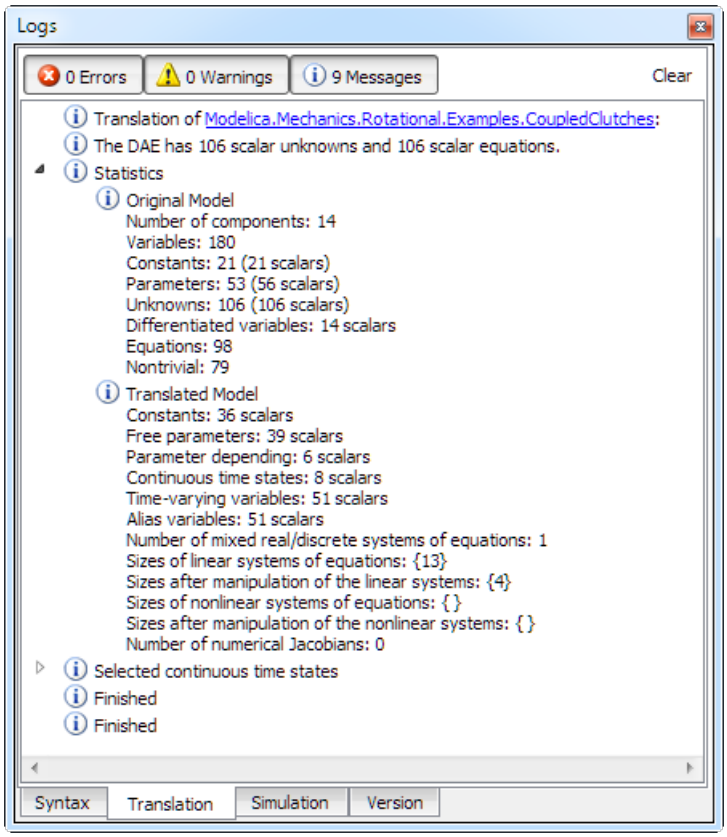
The **Translation** tab shows the translation log; information and statistics related to translation and symbolic manipulation of a model. The translation tab can be very useful when debugging and improving models, for examples to find iteration variables missing start values and to see if there is any dynamic state selection.



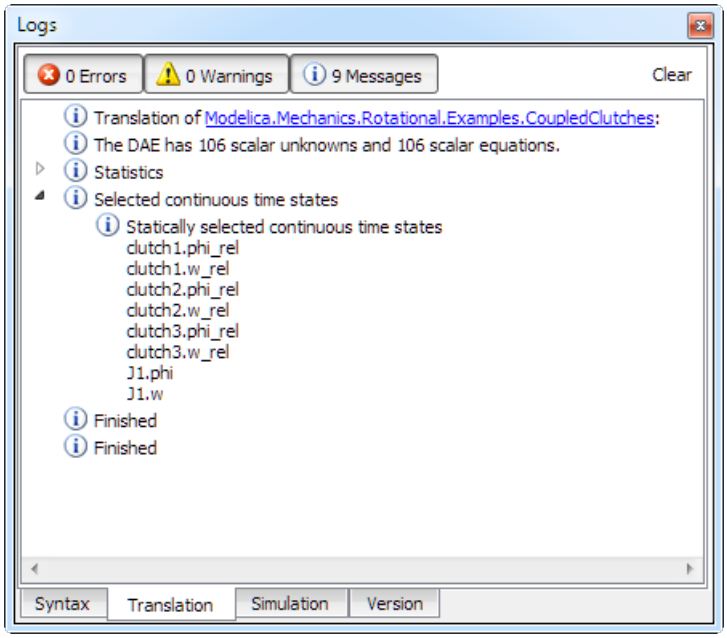
What information is displayed depends on the settings in the **Translation** tab in the simulation setup. The simulation setup is reached by clicking the **Setup** button or by the **Simulation > Setup…** command. Please see the section “Translation tab” on page 805 for more information.

There are two cases; information about errors/warnings when the translation failed, and information about a successful translation. For the former case, please see section “Errors and warnings when translating” on page 753, and “More about information in the Translation tab of the log window” on page 976.

In the sample screenshot below of a successful translation, the translation tab is shown for the demo model `CoupledClutches` (accessible through **File > Demos > CoupledClutches**). The `Statistics` message group has been expanded. Here we can see, for example, that the model has 8 continuous time states. We can also see some statistics on the number of linear and nonlinear equation systems and the sizes of these systems before and after symbolic manipulation.



Selected continuous time states can be displayed by expanding this item:



The states are by default listed as scalar entities. This corresponds to the flag:

```
Advanced.LogStateSelectionScalarized = true;
```

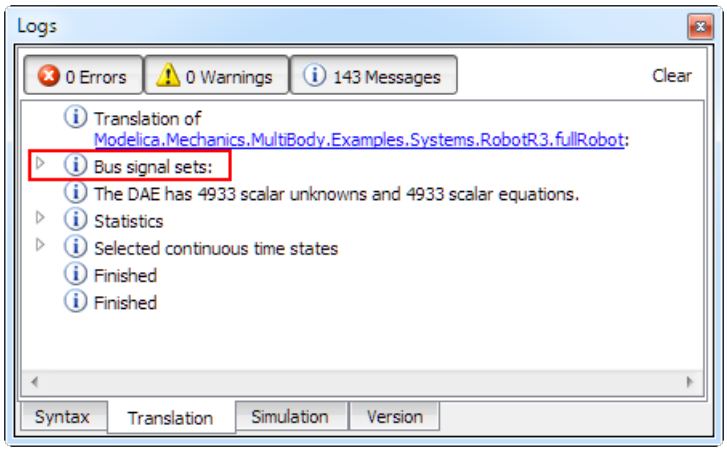
By setting this flag to `false`, vectors and matrices are listed if all their elements are states.

Bus signal tracing

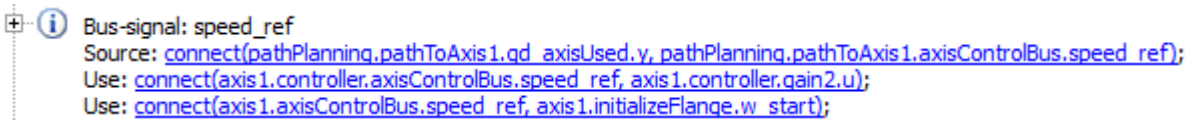
From the Translaton tab, bus signal (expandable connectors) tracing can be performed. To get such results, tick **Log bus signal sets** in the in the **Translation** tab in the simulation setup.

This setting is by default not activated. Activating this setting corresponds to setting the flag `Advanced.LogBusSignalSets=true`.

When the setting is active, you get a new entry in the **Translation** tab of the Command log (if the model has expandable connectors, like for example the Robot demo).

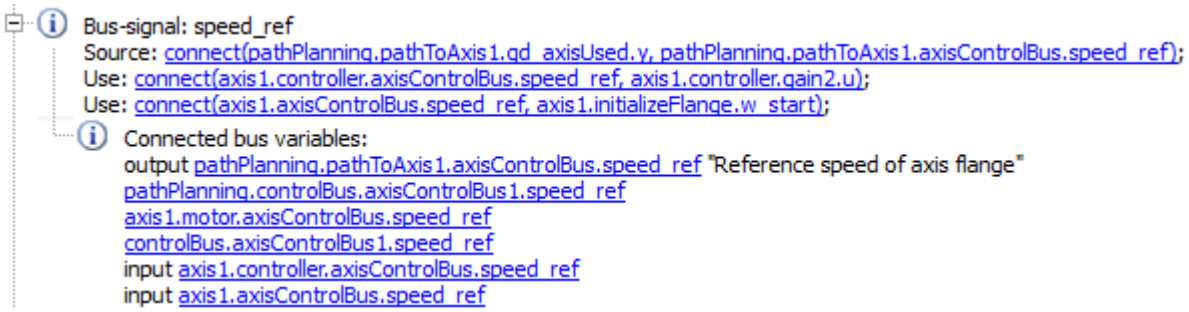


Expanding this display several instances like:



Source means that a causal signal originates at that place, and **Use** that the signal is used at that place. Clicking on a link highlights both components of connect-statement in the diagram layer.

The node can be opened to show detailed content (this information is also shown for some errors in order to help with localizing the error).



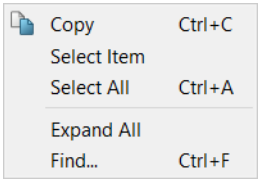
Notes:

- **Source** without **Use** means that the signal is not used.
- **Use** without **Source** means:
    - For a physical signal (`pin`, `flange`) this is normal.
    - For a causal signal this means that the model in incomplete (except that it the model has a public top level bus it will automatically get one external input).

- Neither **Source** nor **Use**: The signal is only present in the connector definition and is not used.
- Another useful tool when it comes to search for connections and connected components is the command **Edit > Find Connection**. For more information about this command, please see the documentation for that command in previous chapter. Note that the diagram must be displayed to use this command.

Context menu

By right-clicking, a text in the tab, a context menu is displayed:



Please see section “Context menu: Log window” on page 890 for more information.

The content in the tab can be erased by clicking **Clear** in the header.

Simulation tab

The simulation log

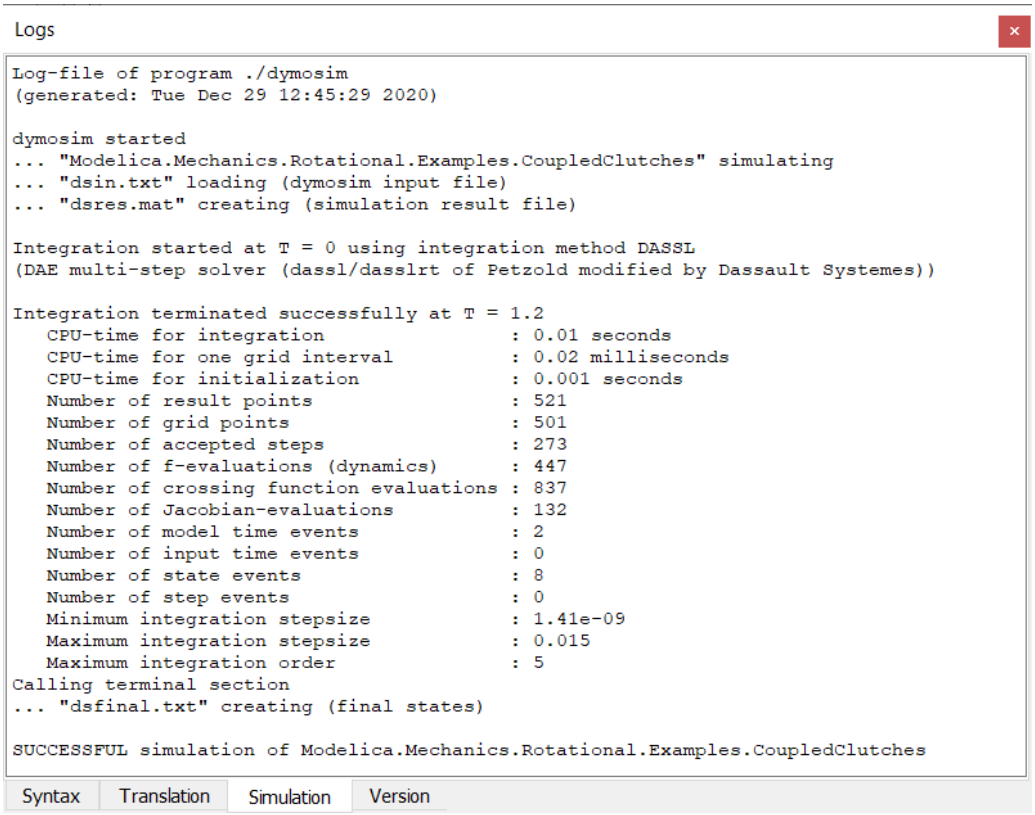
The simulation tab displays the simulation log; information regarding the simulation of a model. The layout and available information may differ from the example image below depending on the selected integration algorithm.

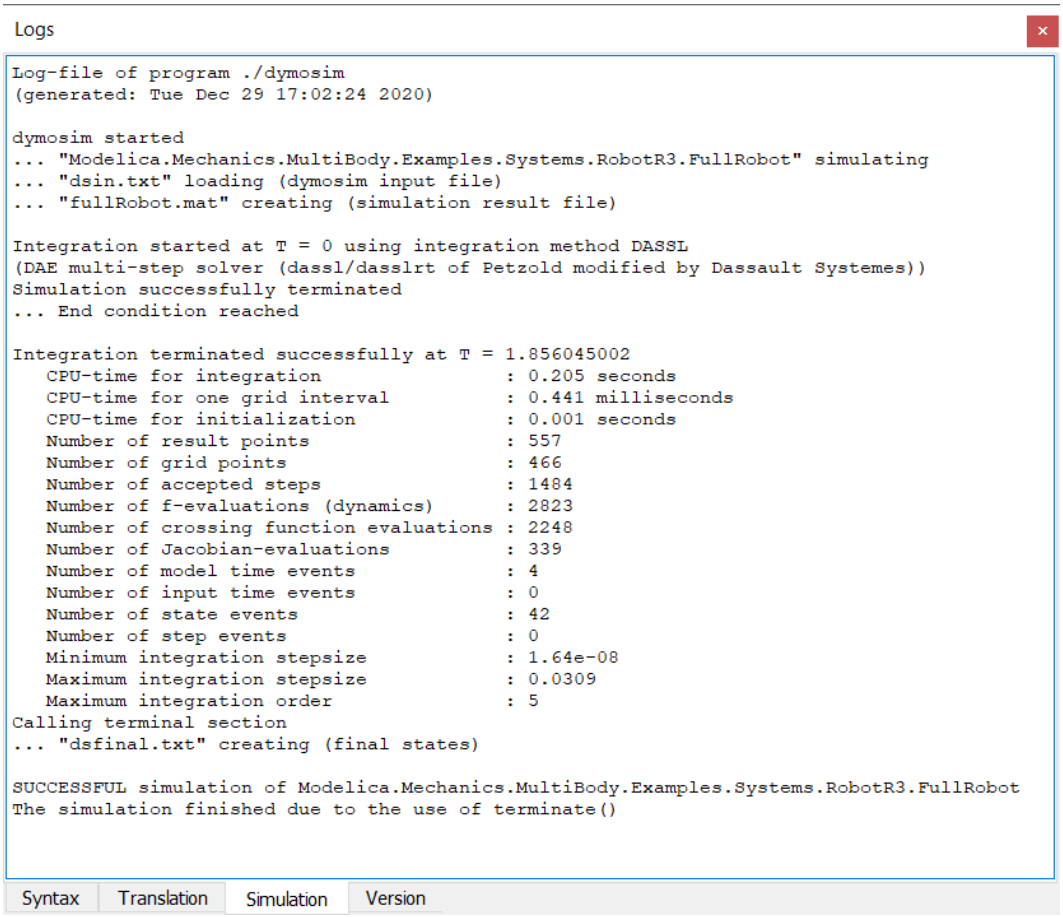
From the simulation tab we can see, for example, start and stop time, CPU time for integration and initialization, and the number of result points.

At the end of the simulation log a message is printed, clearly stating whether the simulation was successful or if it failed. The message also states if the simulation was terminated by the operator `terminate ()`. The message also contains the full path to the model being simulated.

The full path to the model is also given when the simulation in started.

The simulation tab contains useful information to determine if the model has events, and also performance related statistics as the number of function- and Jacobian-evaluations. This can be important for debugging and for improving performance of models. For more information, see “More about information in the Simulation tab of the log window” on page 977.


Logs
Log-file of program ./dymosim
(generated: Tue Dec 29 12:45:29 2020)
dymosim started
... "Modelica.Mechanics.Rotational.Examples.CoupledClutches" simulating
... "dsin.txt" loading (dymosim input file)
... "dsres.mat" creating (simulation result file)
Integration started at T = 0 using integration method DASSL
(DAE multi-step solver (dassl/dasslrt of Petzold modified by Dassault Systemes))
Integration terminated successfully at T = 1.2
CPU-time for integration : 0.01 seconds
CPU-time for one grid interval : 0.02 milliseconds
CPU-time for initialization : 0.001 seconds
Number of result points : 521
Number of grid points : 501
Number of accepted steps : 273
Number of f-evaluations (dynamics) : 447
Number of crossing function evaluations : 837
Number of Jacobian-evaluations : 132
Number of model time events : 2
Number of input time events : 0
Number of state events : 8
Number of step events : 0
Minimum integration stepsize : 1.41e-09
Maximum integration stepsize : 0.015
Maximum integration order : 5
Calling terminal section
... "dsfinal.txt" creating (final states)
SUCCESSFUL simulation of Modelica.Mechanics.Rotational.Examples.CoupledClutches
Syntax
Translation
Simulation
Version



Note that the command **Simulation > Show Log** will display the log window with the simulation tab active.

Interpretation of simulation log entries

The following is an interpretation of the lines in the simulation log between the lines `Integration terminated successfully…` and `Calling terminal section`. The log from simulating the robot above is used for figure examples.

`CPU-time for integration`: Shows how long the CPU was occupied (wall-clock time) with the task of computing the result for the model. Still this time is a bit unreliable; in particular, if the simulation time is in range of milliseconds, as the CPU time is influenced by other tasks that the CPU has to take care of simultaneously. Note that to get an approximate value for the full CPU time used, you must add the `CPU-time for initialization`. To get the exact CPU time used, you can activate the flag `Advanced.GenerateBlockTimers`. For mote information about using this flag, see “Basic profiling” on page 1018.

`CPU-time for one grid interval`: This value is calulated by dividing `CPU-time for integration` with `Number of grid points` (see below).

`CPU-time for initialization`: Shows how long the CPU was occupied (wall-clock time) with the task of initialization of the model. Still this time is a bit unreliable as it is influenced by other tasks that the CPU has to take care of simultaneously, in particular if the initialization time is in range of milliseconds.

`Number of result points`: Shows how many points are stored in the simulation result file. For how to explain the value, see `Number of grid points` below.

`Number of grid points`: Shows how many points in the simulation result file that are defined by the solver settings (either by the interval or by the absolute value). The number from the robot demo above is 466. That number is explained by the solver settings of 500 points in 2 seconds, and the simulation terminated after 1.856 seconds. This gives (1.856s/2s)*500 = 464 intervals. However, there is one interval more, the one between 1.856 seconds and the exact end point. This gives 465 intervals, which means 466 grid points. The reason why the `Number of result points` is higher, 557 points, are;

- Every event adds two points to the result file for better plotting. (This can be deactivated in the solver settings since it can make post-processing cumbersome, by deactivating the setting **Store variables at events**, see "Output tab" starting at page 809) Looking at the number of events that are present (for explaining the events, see that entry below); we have 4 model time events and 42 state events. This gives the number of result points as 466 + 4*2+42*2=558 points.
- The last event equals the end of the simulation, therefore we must subtract one point from this file; this gives us the total number of 557 points in the simulation result file.

`Number of accepted steps`: That many accepted steps were carried out by the integration algorithm. The (variable-stepsize) integrators choose their own step sizes and therefore this number typically doesn´t match the number of result points.

`Number of f-evaluations (dynamics)`: Counts how many times the model has been evaluated for computing only the outputs and the state derivatives. A bit more precisely, it describes how often the "Output Section" and "Dynamics Section" in `dsmodel.mof` has been evaluated. This number will multiply for higher order single-loop solvers depending on the number of predictor-corrector evaluations.

Another way of explaining this number is that it is the number of function evaluations that affect the dynamics, i.e. function evaluations used by the solver to advance time, normally represented by derivative computations in an ODE.

Finally, you can see this number as the number of iterations needed to compute the next state in the continuous simulation.

Note that the number of f-evaluations does not contain the function evaluations done to compute Jacobians. For that number, see below.

`Number of crossing function evaluations`: Indicates how often the model was evaluated either to determine if a state-based event (zero crossing) has happened, or to localize

such an event. This is used to find the exact time at which state events occur. A high value indicates many state events. (As an example, consider

```
when x >= 0.5 then
   flag = true;
end when;
```

When the solver tries to find the exact time when the variable x passes 0.5 a number of crossing function evaluations are performed.)

`Number of Jacobian-evaluations`: Counts how often the integrator Jacobian has been computed. If a numeric Jacobian is computed (default), several f-evaluations are required for each Jacobian evaluation. These f-evaluations are not counted under the `Number of f-evaluations (dynamics)` above.

`Number of model time events`: Represents the number of time-based events in the model, for example, something like "if (time < 2)…" changing its value. Note that only "time < x" or "time >= x" generate time events. Using "time > x" or "time <= x" will generate state events, according to the Modelica specification.

`Number of input time events`: Counts events that are generated from the model inputs, for example, in `dsu.txt`.

`Number of state events`: Represents the number of state-based events in the model, for example, when something like "x > 2" change its value. See the note above for `Number of model time events`.

`Number of step events`: A step event is generated when a simulation with dynamic state selection switches currently selected set of states.

`Minimum integration stepsize`: This variable shows the minimum step size chosen by the solver throughout the simulation.

`Maximum integration stepsize`: Similar to the above, but reporting the maximum step size.

`Maximum integration order`: This number shows the maximum integration order used during the simulation.

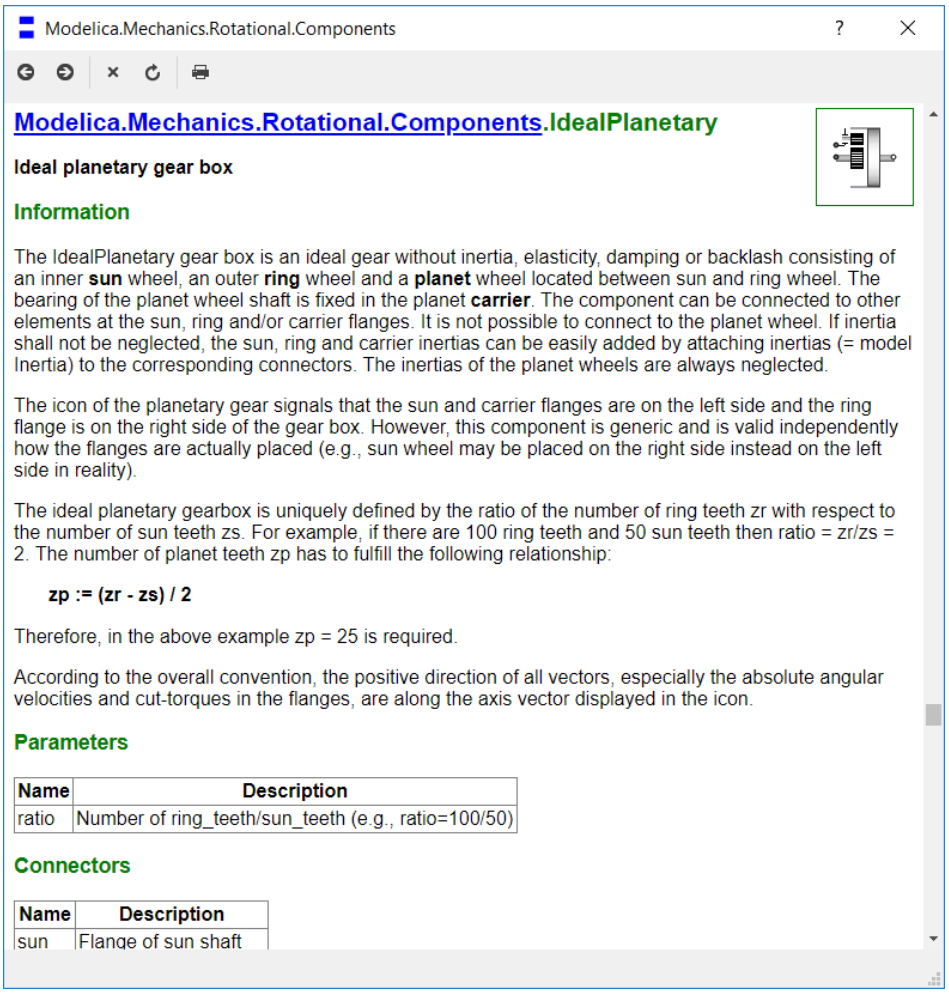
Version tab

The version tab shows information related to version management with Dymola's built-in support for CVS and SVN source control systems accessible with the ModelManagement option. Please see the chapter "Model Management" for more information about this.

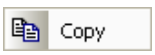
5.1.10 Information browser

An information browser is used to display model documentation, e.g. when using any **Info** button or **Info** command in any context menu.

Information browser.



The toolbar contains the usual **Go back**, **Go forward**, **Stop**, **Reload, Copy** and **Print** facilities.



Right-clicking on a selected text will pop a menu that enables copying of that text. **Ctrl+C** can of course also be used, as can the corresponding icon in the toolbar.



Right-clicking on an empty area will give a context menu that allows going back or reloading present view.

These possibilities are also available in the toolbar.



Right-clicking on an image will give a context menu that allows copying the image to the clipboard.

For more information what is displayed in the browser, see the index entry "*Info : button*" in the index in the end of this manual.

5.2 Model simulation

5.2.1 Basic steps

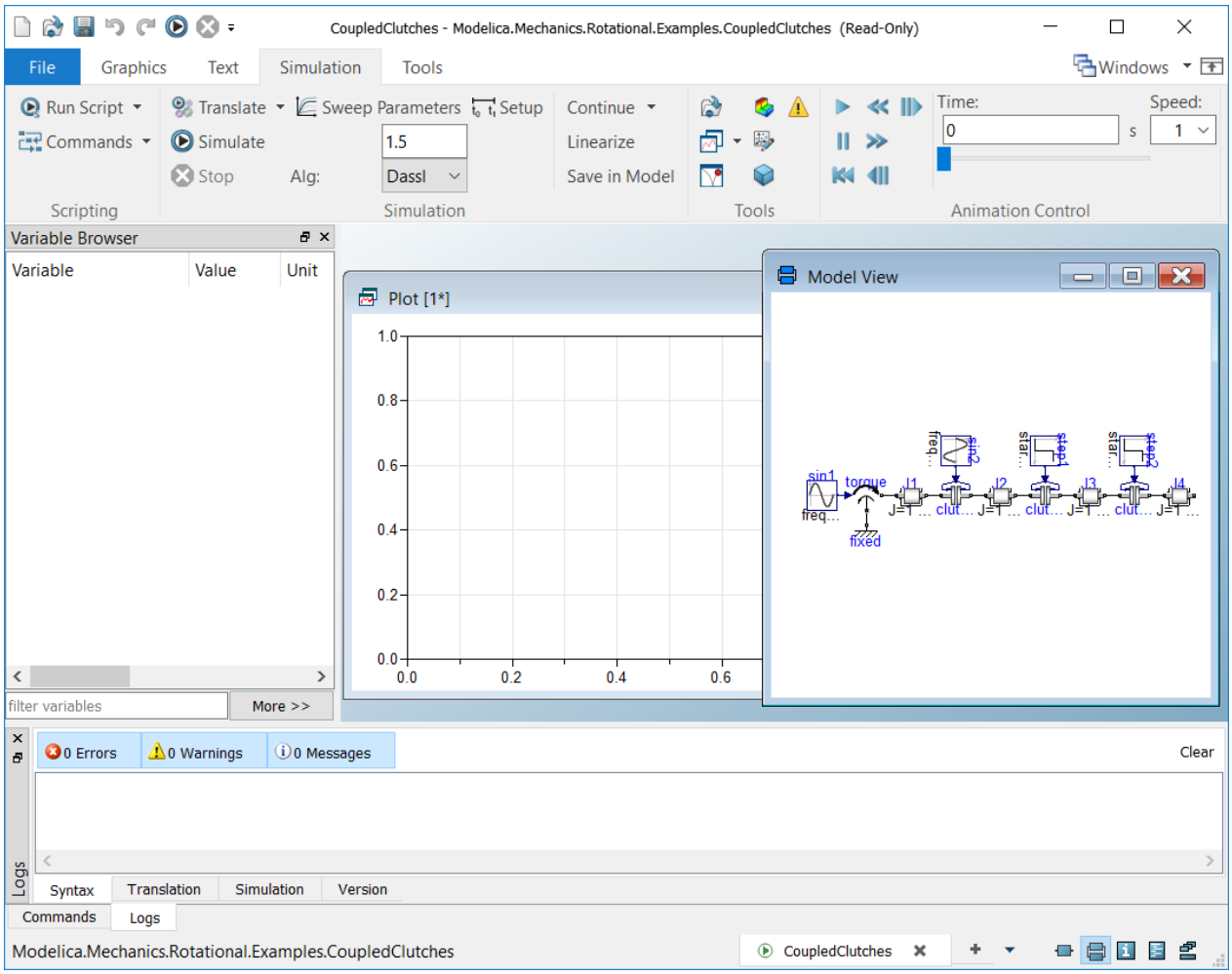
The basic steps when simulating a model will be outlined below. The focus will be on the use of Dymola's graphical user interface. The script facility is described in the section "Scripting" starting on page 904.

Selecting model

The model to be used in a simulation experiment, the active simulation model, is when working in the **Graphics** tab or **Text** tab. For more about these tabs and developing a model, see previous chapter.

By default, the active simulaton model is the current model in the currently selected model tab. However, you can yourself select a certain model that should always be the active simulation model. For more information about this, see the previous chapter, section "Basic model editing", subsection "Selection of active simulation model".

To work with simulating a model, click the **Simulation** tab. This gives you a number of commands for handling the simulation, an example:



(In most cases, it is better to work with full screen, but the tab will be too small if such an image is made here. For a better view of how the **Simulation** tab looks when working with full screen, see "Main window: Script Editor tab" starting on page 832. Here also all commands of the tab are presented.)

Translation

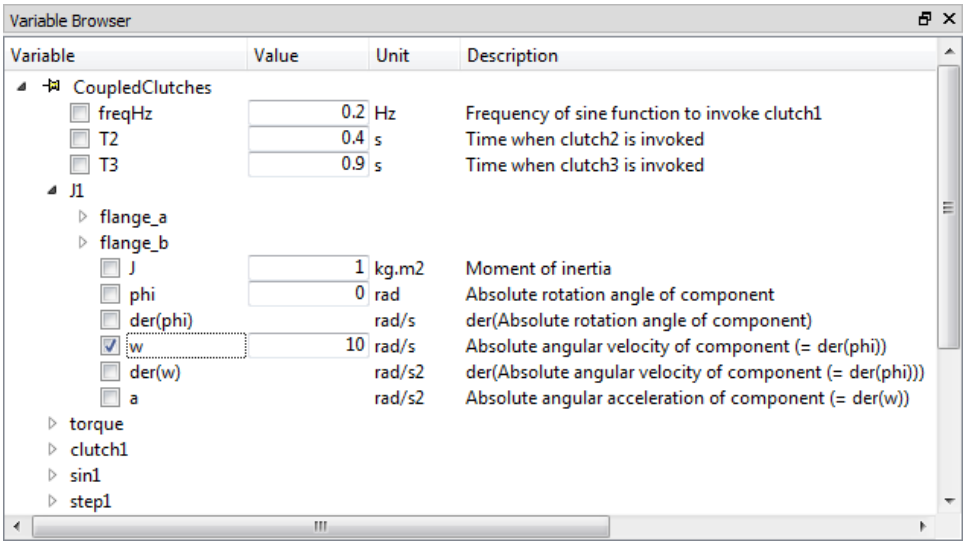


To prepare a model for simulation, it needs to be translated. The translation is initiated by pressing the **Translate** button in the **Simulation** tab. If the translation fails, a message is displayed in the log window. For more information on this, please see "Errors and warnings when translating" on page 753.

Setting parameters and initial conditions

After translation, new values for parameters and initial values can be entered using the variable browser, which displays a tree of all variables.

The variable browser for setting parameters and initial values.



Please note the possibility to use the diagram layer to displaying the wanted variables.

For parameters, such as J1.J, there is an input field in the value column. New values are set by clicking in the corresponding value cell and entering the new value. The description strings are extracted from the model classes.

For time varying variables having active start values i.e., fixed=**true** and the start value being a literal, there are also input fields to change their start values. Above J1.phi and J1.w are examples of such variables. Setting parameters may of course also influence an active start value bound to a parameter expression.

Modified initial conditions and parameter values entered in the variable browser when simulating the model can be saved to the model.

For more information about the variable browser, see “Variable browser interaction” starting on page 619.

Specify simulation run



To set up the experiment, click on the **Setup** button to get a menu for setting simulation interval, output interval and specifying integration algorithm etc. (A number of settings are available; please also see the section “Simulation settings” on page 782.) Notes:

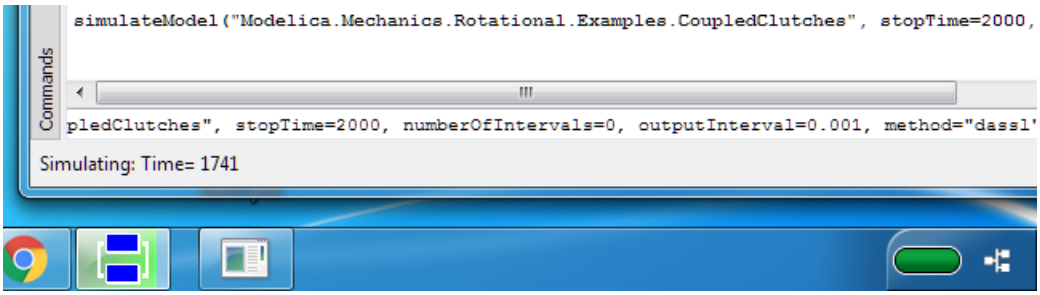
- You can directly select the stop time and the algorithm (beside the setup button).
- In the setup, you can also see the active simulation model name, and if it has been selected by the user.

Perform simulation



To run the simulation, click on the **Simulate** button.

In Windows, when you simulate, you get at progress bar in the taskbar icon for Dymola. During initialization you get a green blob that moves back and forth. During simulation you get a progress bar based on the current simulation time. In the example below the max time was set to 2000 s. Note that this progess bar is currently only available on Windows.



Plot results

What simulation result files should be displayed can be selected by the user, and what result files that should be kept between simulations can be selected by the user or automatically. For the handling of simulation result files in the variable browser, please see section “Handling of simulation result files in the variable browser” starting on page 674.

Dymola supports plotting of any variable. Multiple plot windows may be created. Each plot window may contain several diagrams. Multiple curves in each diagram are allowed. Multiple diagrams in a plot window allow the user to lay out the curves nicely with aligned time axis and different heights and y-scales of the diagrams. Text object can be inserted in plots; time line can be displayed as well as signals operators, expressions can be plotted, etc.

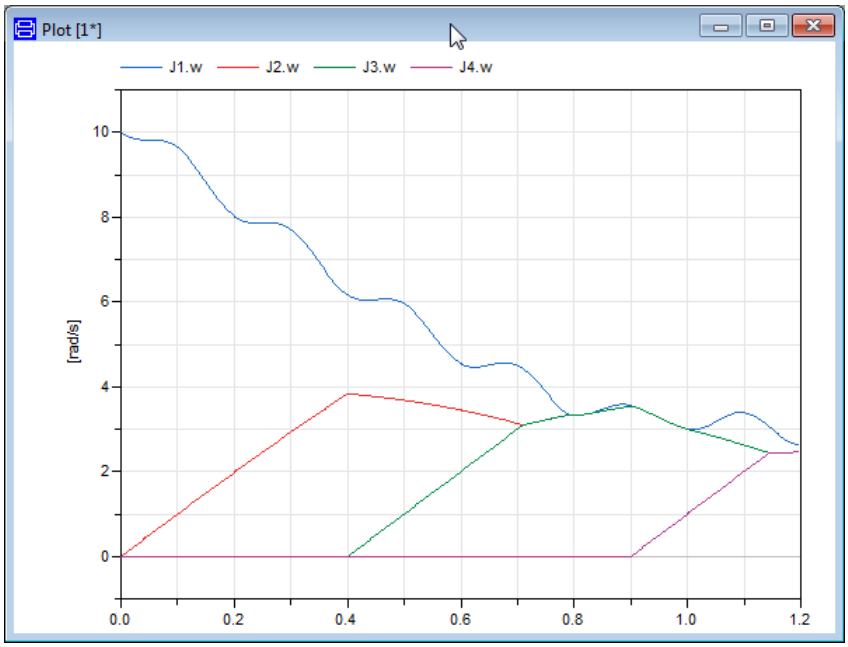
Variables to be plotted are selected by clicking in the variable browser, or by dragging the variable to a plot/table window. When a variable is selected by clicking or has been dragged to a plot/table window, the square in front of the name is ticked. The variable browser above has J1.w and J2.w selected for plotting.

The user can interact in the plot window using zooming, tooltips etc. Please see section “Plot window interaction” starting on page 692 for more information.

For an overview of where to find information in the manual concerning plot functionality, please see “Plotting” starting on page 690.

Please note that the diagram layer can always be displayed in the model layer window when simulating. That diagram can be used to decide what variables should be shown in the variable browser. Please see section “Using the diagram layer in the Model View window to select what to show in the variable browser” on page 620 for more information.

A plot window.



Exporting results

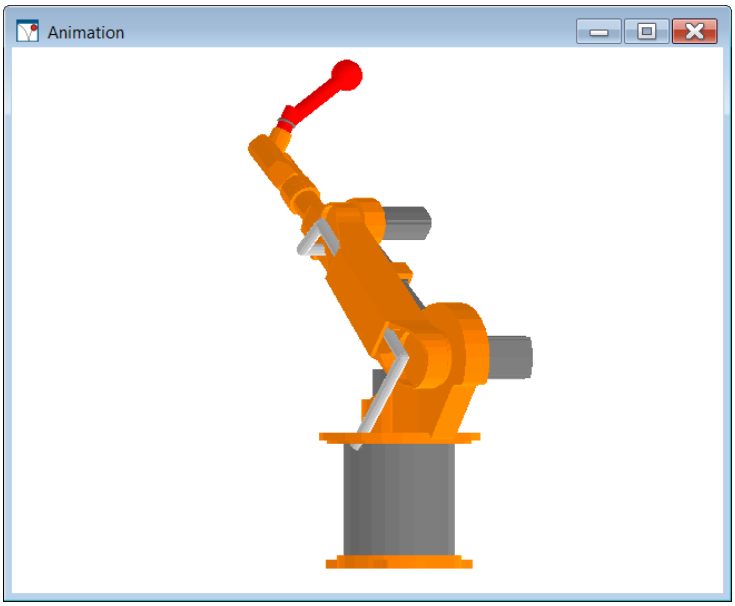
It is possible to export plotted variables in files of different formats (CSV, Matlab, Scientific Data Format (SDF), and text format). Please see section “Context menu: Variable browser – nodes” on page 869 for more information.

It is also possible to copy curve values, tooltip values, signal operator values and values from tables, to the clipboard, by right-clicking them in plot/table windows, and using the copy commands in the context menu.

Animation

If the model has been prepared for it, animation can be displayed in an animation window. For more details on animation, please see section “Animation” starting on page 757 and section “Animation window interaction” starting on page 760.

An animation window



Documenting the simulation

The simulation can be documented using the command window. A documentation editor is optional in the command log pane of the command window. The content of plot and animation windows can be inserted in the documentation by activating flags. Links and images can be added, math rendering of formulas and equations (including Greek letters and indexing) can be displayed. It is also possible to copy content to e.g. Microsoft Word for further work. Please see section "Working with documentation in the command window" starting on page 771 for more information about documentation of simulations.

Storing the simulation as a script

It is possible storing the simulation as a script file, or inserting the commands of the simulation into e.g. a function. For more information about scripting, please see section "Scripting" starting on page 904.

5.2.2 Variable browser interaction

Displaying plot variables

To be able to select plot variables, the first thing to do is to display relevant variables. This can be done in two ways; either by navigation in the variable browser directly or using the diagram layer in the Model View window to select the interesting component(s) and then work with the corresponding variables displayed in the variable browser.

Note that the variables available in the variable browser can be controlled by the programmer using annotations for variable selections. Please refer to previous chapter, section "Advanced

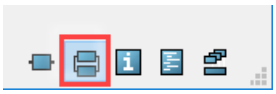
model editing", sub-section "Matching and variable selections". However, all variables can still be available if the setting **All variables; ignore selections in model** has been set in the **Output** tab reached by the **Simulation > Setup** command. Please see section "Output tab" on page 809 for more information about this.

Note also that the setting **Store variables after failed initialization** can be used to make the result of a failed initilalization available in the variable browser for debugging. For details, see "Plotting the results of the failed initialization using the variable browser" starting on page 1027.

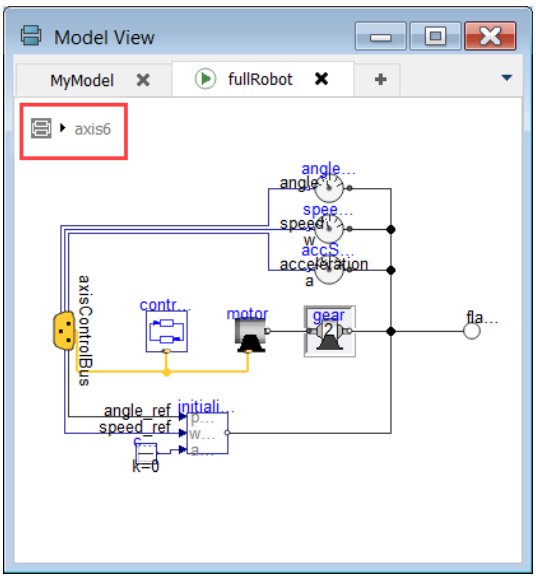
Using the diagram layer in the Model View window to select what to show in the variable browser

The diagram layer representing the model can always be displayed in the Model View window when working in the **Simulation** tab, to enable you to follow a simulation by displaying variables and to control it by setting parameters. You can descend into any level of the model in order to plot or display variables.

If the Model View window is not displayed, or if it displays another model layer (for example, the Modelica text layer), click on the button for diagram layer in the toolbar in the right lower corner of the Dymola window:

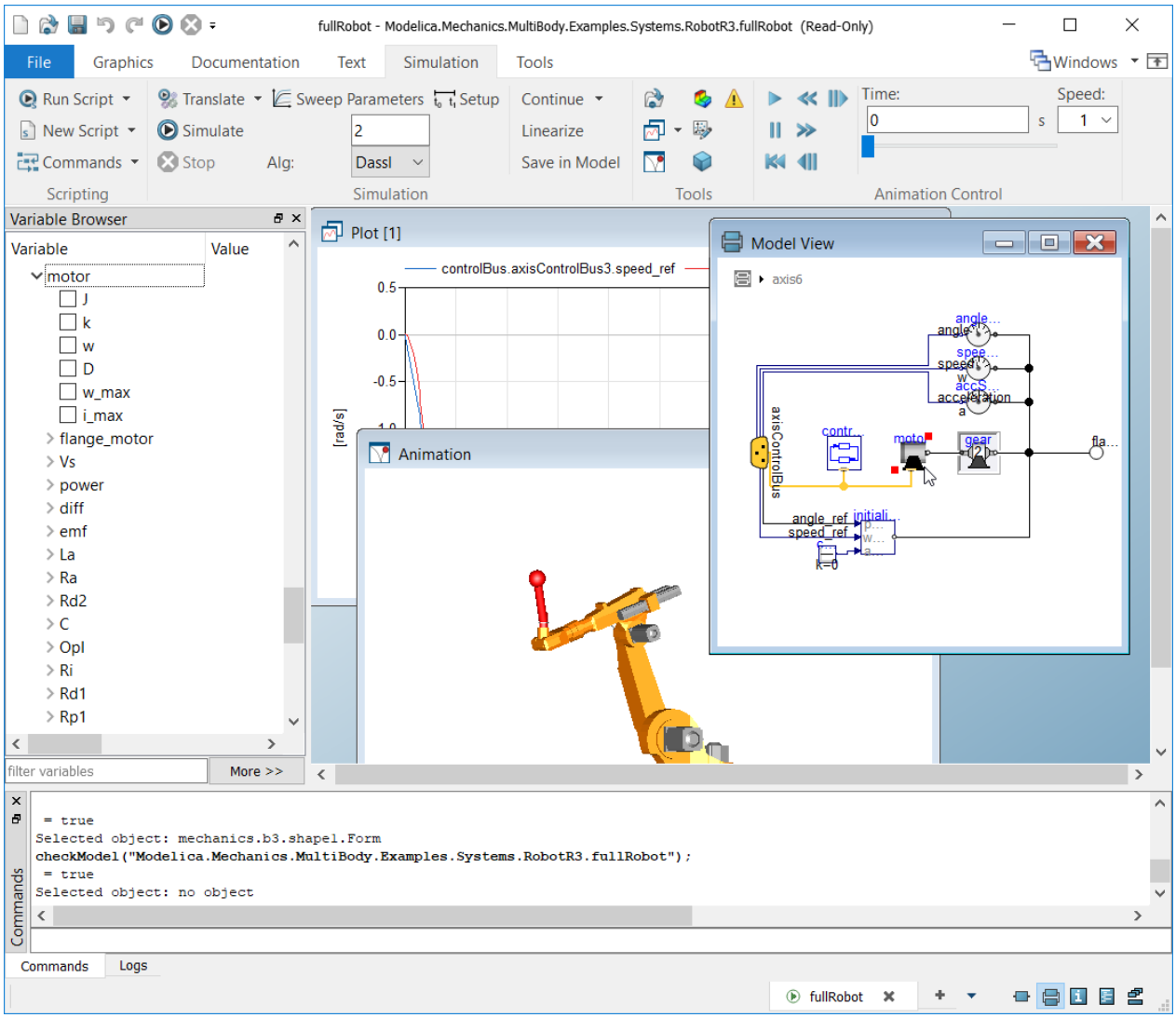


To display the content of a components in the model, select it, right-click to display the context menu and select **Show Component**. You can perform this several times to go deper into the hierarchy. To go back to any previous hearchy level, click any part of the model hierarchy path displayed in the upper left corner of the Model View window.



In the example above, if you would click on the Dymola icon that is the first item in the path, you would return to the top level of the displayed model; in this example the Robot demo. (The window shows that right now you have performed **Show Component** on the component **axis6** in the Robot demo.)

The context menu for a component contains the choice **Show Variables**, which will highlight and open the selected component instance in the variable browser – below the variables for the motor have been displayed this way.



Please note selection of another component in the variable browser will also change the selection in the Model View window.

It is possible to filer components in the diagram layer, to dim components not matching domains/component names.

More information on how to use the diagram layer when simulating is presented in section "Working with the diagram layer in the Model View window" starting on page 680.

Working with the variable browser

To select plot variables two things must be done. The first is to expand the tree to see the relevant plot variable(s) (if not already visible); the second is to select them.

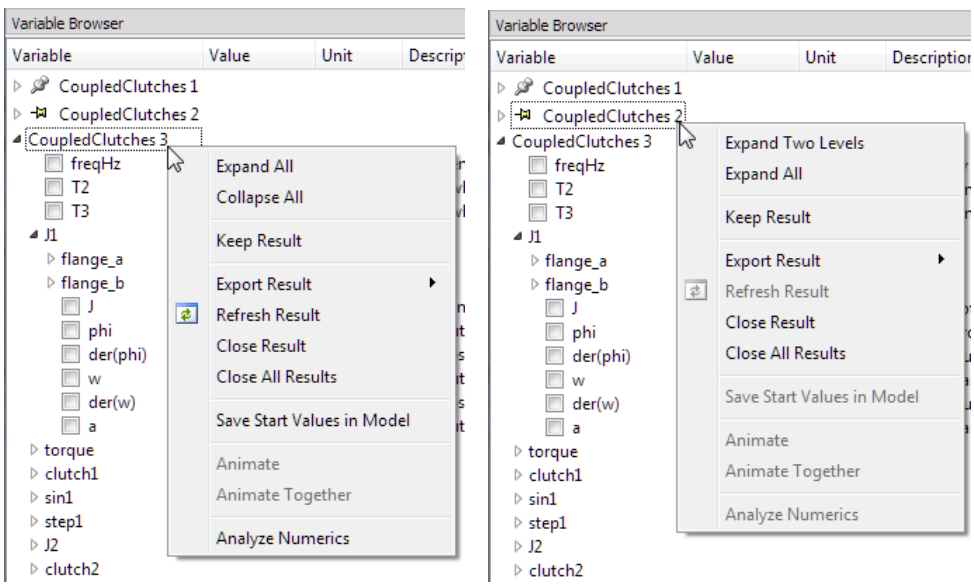
Please note that it is possible to use the diagram layer in the Model View window to decide what variables should be shown in the variable browser. Please see section "Using the diagram layer in the Model View window to select what to show in the variable browser" on page 620 for more information.

Several results of separate simulations (result files) can be shown in the variable browser. Result files to keep when doing a new simulation can be selected. The status of the result files is indicated by icons and tooltips. Please see section "Handling of simulation result files in the variable browser" starting on page 674.

Nodes representing vectors/arrays are indicated by  .

An important tool when working with the variable browser tree is the context menu, reached by right-clicking on any node of interest in the tree. The content of the context menu is different depending on the node; whether it is expanded or collapsed, whether it is a top node or not etc.

Context menus for a top node (expanded and collapsed) in the variable browser



For more information about the context menu, please see section "Context menu: Variable browser – nodes" on page 869. Please note:

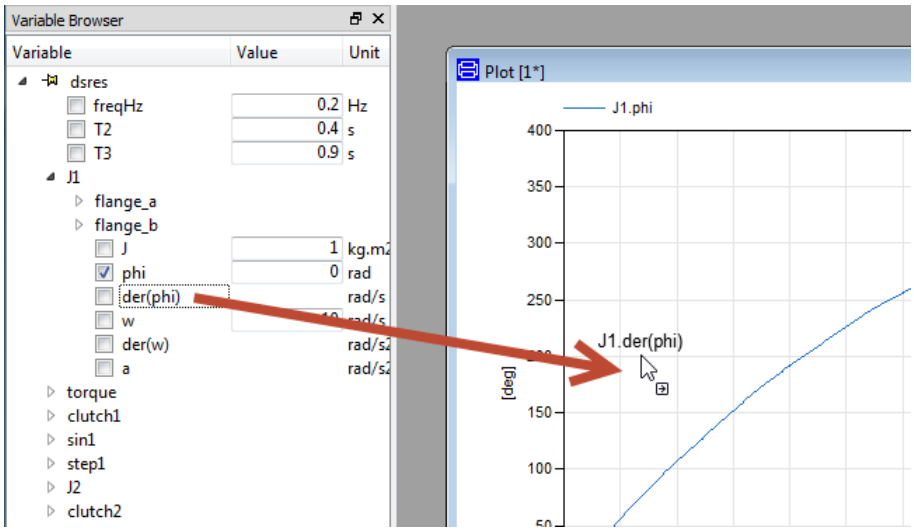
- If the cursor is beside/resting on a *variable*, another context menu is shown; please see section "Context menu: Variable browser – signals and arrays" starting on page 873.
- If the cursor is beside/resting on a vector/array node, another context menu is shown, please see section "Context menu: Variable browser – signals and arrays" starting on page 873.

Opening/Closing of the tree structure

- Clicking on the expand icon ▷ in front of a closed node opens one more level of the tree and displays variables and nested sub-models.
- Double-click on the node name. If not already open, opens the node and displays variables and nested nodes. Otherwise the node is closed.
- If you are located on a node, clicking right arrow will open the node. (You move up and down by arrow up/down.)
- The **Expand Two Levels** operation in the context menu (reached by right-clicking on the model/sub-model) opens two levels, which makes the model structure clearer without creating a huge browse tree.
- (The **Expand All** operation in the context menu opens every sub-node in the browse tree. Large sub-trees may become very large and hard to navigate.)
- Clicking on the collapse icon **–** in front of an opened node will close it.
- If you are located on a node or variable, clicking left arrow will close the node. (You move up and down by arrow up/down.)
- The **Collapse All** operation in the context menu will collapse all nodes in the browse tree. The difference is that the next time you open the node, all nodes will be closed.

Selecting the variable(s)

- **Click on a variable**. Plots the variables if it was previously unselected; the diagram is normally rescaled. If the variable was already selected, it is removed from the diagram.
- If you are located on a variable, the space key will select it or deselect it (toggle). (You move up and down by arrow up/down.)
- **Click on a variable while pressing the SHIFT key.** Plots multiple variables. All variables from the previous non-shifted click to the last click are plotted. Note that multiple variables are only plotted if the range is limited to a single sub-model instance.
- **Drag a variable.** You can drag a variable from the variable browser to a plot window or a table window to create a curve/table. If you drop a variable on the background, a new plot window is opened, where the curve is added. If you drop in a text window, the signal path is inserted. An example of dragging to a plot window:

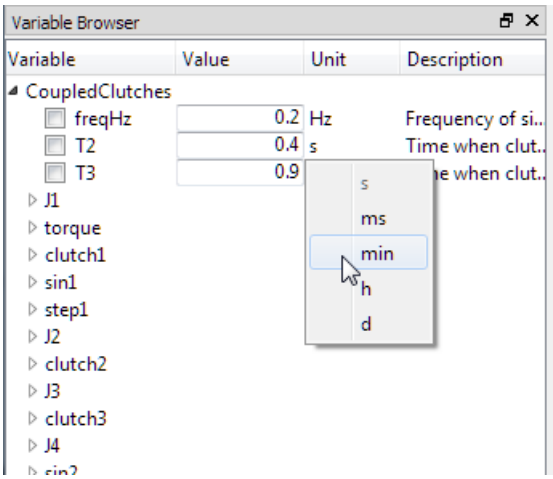


Changing parameter values

Parameter values can be changed from the variable browser by entering values. Parameter values can also be changed from the parameter dialog of components in the diagram layer when simulating. For conditions and interactive parameter tuning, see section “Changing parameter values using the diagram layer” on page 685.

Selecting display units

You can select display units for signals in the variable browser, if display units are defined for the signals, and if the signals are editable. Click on the unit to get a selection like in the parameter dialog:

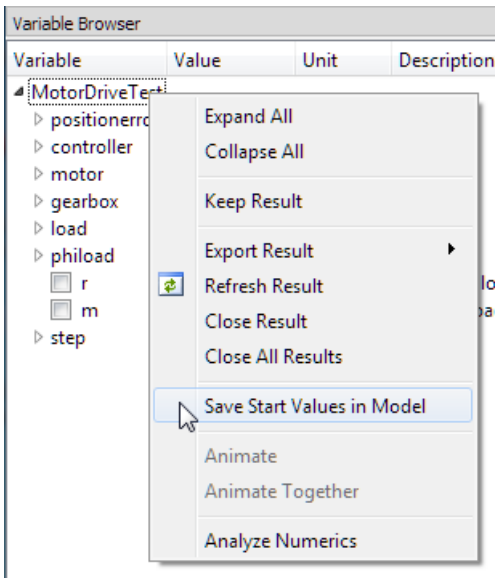


When you have selected the new display unit the value is converted.

Save start values in model

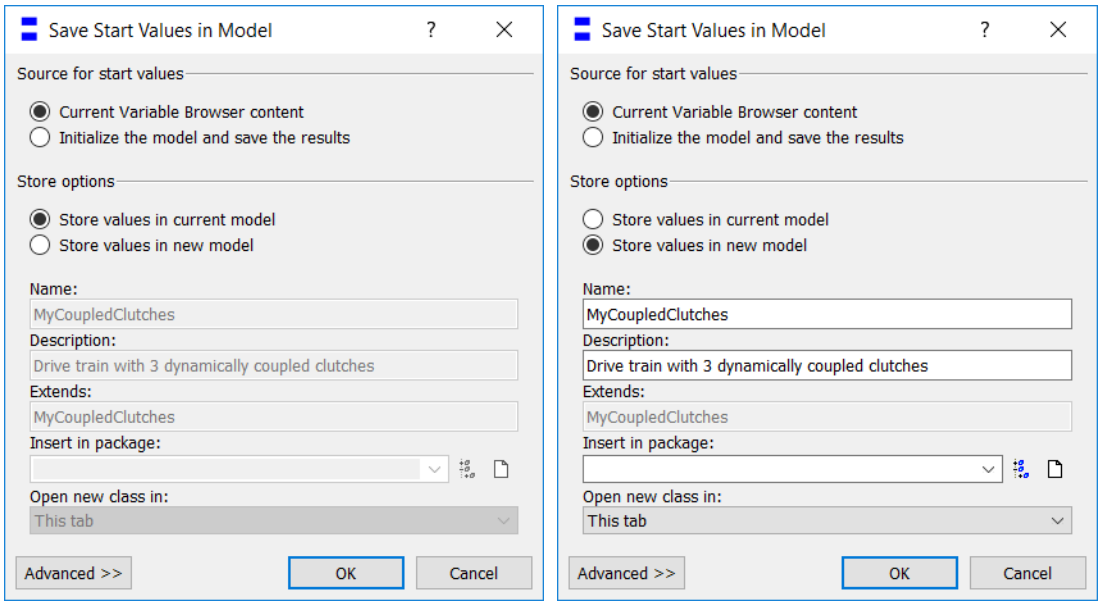
Basic usage

Modified parameter values, initial conditions, and start guesses entered in the variable browser when working with simulation can be saved to the model, typically after being validated by a simulation, by right-clicking the result file in the variable browser and selecting **Save Start Values in Model** from the context menu.



(You can also use the command **Simulation > Save Start Values in Model** as an alternative to the context command.)

It is possible to save the start values to the current model, as well as exend the model before saving. (For more information about extending in general, see index entry "*extend*" in this manual.) This selection is made in the dialog that appears when selecting **Save Start Values in Model**. The default selection in the Store options group is **Store values in current model**, extending to a new model is done by selecting **Store values in new model**. Examples of the dialog without and with extending to a new model:



When the option **Current Variable Browser content** is selected, **Save Start Values in Model** stores, in the model, current changes made in the variable browser to parameters, initial values and start guesses. This option is available after the model has been simulated and if any changes have been made in the variable browser. It is selected by default when available.

Alternatively, Dymola can initialize the model and read the results to the variable browser before storing. To do this, select the option **Initialize the model and save the results**. This feature may, for example, be used to store the results of an initialization as start guesses for iteration variables. When reusing such a model the initialization process will be faster and more robust.

When clicking **OK** with the **Initialize the model and save the results** selected, the model will be initialized at the simulation start time (default time is 0). If the initialization is successful, the results are saved in the model (or in a new extended model depending on the store option) for those modified variables that have modifiable start values. Note that changes to parameters and initial values in the variable browser before opening the **Save Start Values to Model** dialog will be used for the initialization and therefore these changes will also be saved in the model. The option **Initialize the model and save the results** is available directly after the model is translated.

When you save start values in model, start values having a prefix final and start values of protected variables may be among the start values to save. If final start values are modified, Dymola may issue an error when translating the resulting model as the modifications may not be conforming to the Modelica specification. To turn this error message off, you can set the flag:

```
Advanced.Translation.Log.FinalStartModifierError = false
```

(The flag is by default true.) Changing the start values of protected variables only gives a warning by default.

The following model can be used to test this:

```
model ProtectedAndFinalVariables
  Real x(final start=1);
protected
  Real y;
initial equation
  x + exp(x) = 2;
  y + exp(y) = 2;
equation
  der(x) = 1;
  der (y) = 1;
end ProtectedAndFinalVariables;
```

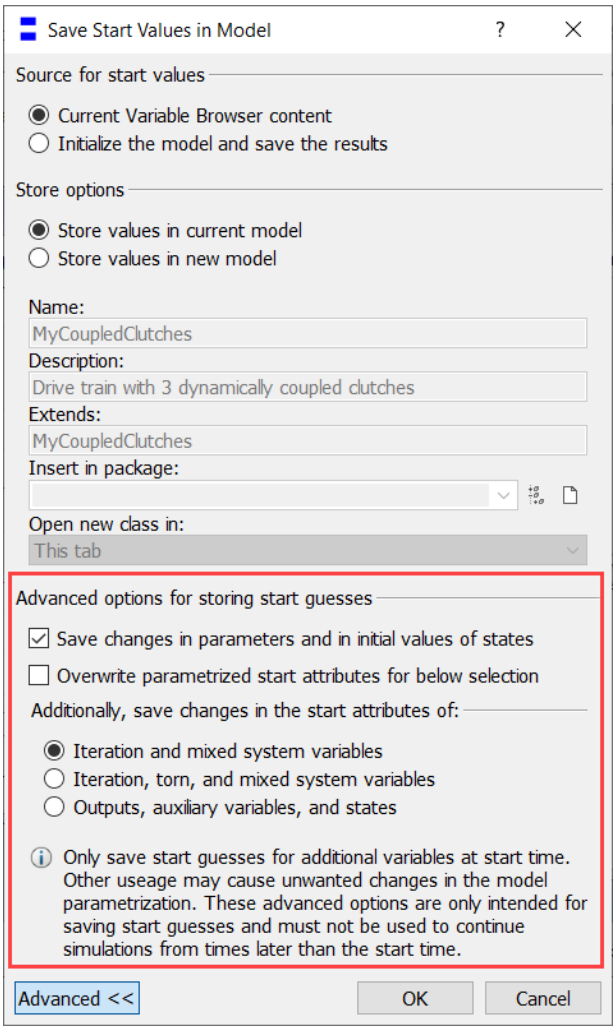
There are some limitations for what parameters, initial values, and start geusses that can be saved in the model. The following cannot be saved with the **Save Start Values in Model** command:

- Evaluated parameters
- Final parameters
- Concealed variables
- Protected variables may cause problems, see above paragraph
- Arrays of variables in arrays of components
- Start guesses for derivatives, e.g. der(x)

See also the **Simulation > Continue > Import initial…** command, in section “Simulation > Continue > Import Initial…” on page 827.

Advanced settings

By clicking the button **Advanced >>** in the dialog, a number of advanced settings are displayed:



By default, changes in parameters, initial values, and start guesses are saved in the model. Start guesses are only stored for variables that are currently iteration variables of nonlinear systems of equations. By using the advanced options, it is possible to store modifications to start attributes of additional variables. The reason is that these variables may potentially become iteration variables when the model is reused in another configuration. This includes, e.g., changes to the model or using an instance of it as a component in a larger model. Upgrading to a newer version of Dymola may also cause changes in which variables are used as iteration variables. The purpose of saving start guesses is to enable easier reuse of the model. This is done by initializing the model at a chosen operating point and then storing results of the initialization as start guesses for reusage in other configurations.

The setting **Save changes in parameters and in initial values of states** can be disabled to only store start guesses as specified by the following options. Note that if there are parameters

that are part of nonlinear equation systems their start attributes will still be modified if other options of the dialog call for it.

If **Overwrite parametrized start attributes for below selection** is enabled, start attributes that are defined by parameters will be overwritten by the value given by the result of the initialization. This applies to the start attributes for the variables as specified by the following selection.

The **Additionally, save changes in the start attributes of**-group specifies which variables that start guesses should be stored for.

- By default, start guesses for current interation variables are stored; the setting **Iteration and mixed system variables** is by default enabled. Start guesses for variables that describe the discrete state of mixed systems of equations are also stored. In particular, note the possibility of having this setting enabled but disabling both the above settings (**Save changes in parameters and initial values of states** and **Overwrite parametrized start attributes for below selection**) – the result will be that we only save the start guesses for iteration variables of nonlinear systems and start guesses for the discrete states of mixed systems. This guarantees that the parametrization of the model is not changed – the only change is improved start value guesses that simplifies the numerical solving of the initialization problem. All other changes that the user has done in the variable browser are discarded.
- By selecting **Iteration, torn, and mixed system variables** the start attributes are stored for all variables that are currently part of a nonlinear system of equation. This includes iteration variables as well as torn variables. Also start guesses for variables that describe the discrete state of mixed systems of equations are stored
- Alternatively, **Outputs, auxiliary variables, and states** may be selected to store start attributes of all variables that may potentially become part of nonlinear or mixed systems in another configuration. This triggers overwriting of all parametrized start attributes.

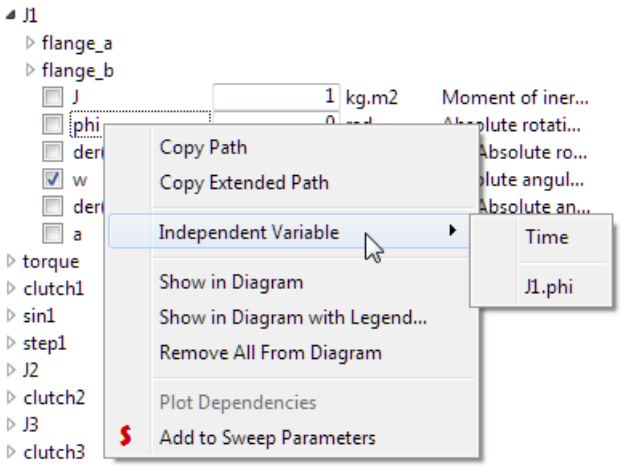
All options under **Save changes in the start attributes of**-group are subject to the normal limitation of the Save Start Values in Model functionality, see above section.

Note that these advanced options are not intended to be used to continue a simulation from a time later than the start time. For that functionality please refer to the command **Script Editor > Generate Script**, selecting **Variables** for storing in the script. See section “Script Editor > Generate Script” on page 833.

Selecting independent variable

The default behavior for plotting is trend curves. It means that time is the independent variable (x-axis variable). To select another independent variable, one alternative is to go to the variable browser, put the cursor on the variable and right-click. A context menu pops up where the third alternative is a selection of independent variable.

Selecting independent variable.



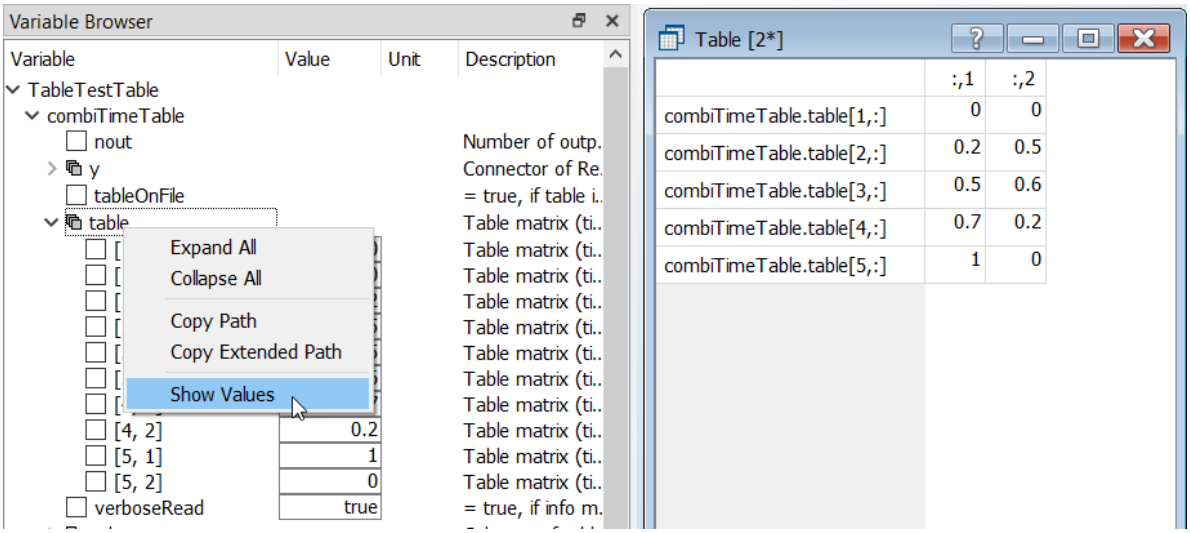
In the above figure, `J1.w` is already plotted, and by right-clicking `J1.phi` and pausing over **Independent Variable**, we see that we can now select `J1.phi` as the independent variable for the plot of `J1.w`.

Time is always an alternative to allow going back to plotting trend curves.

Other alternatives to select independent variable is to use any of the commands **Plot: Options > Independent Variable** or use the context menu of, for example, a curve and select **Independent Variable**. (The menu entry **Independent Variable** is available in a number of different context menus for plot features.)

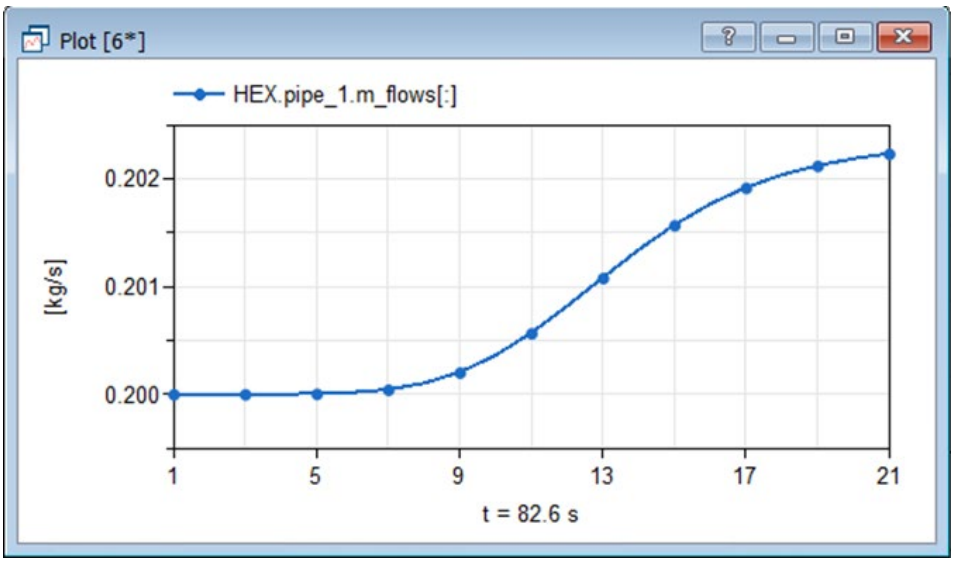
Displaying constant arrays as matrices

The context command **Show Values** is available for constant arrays in the variable browser. The command creates a table window with the values in the array. An example:



Plotting a 1D discretization

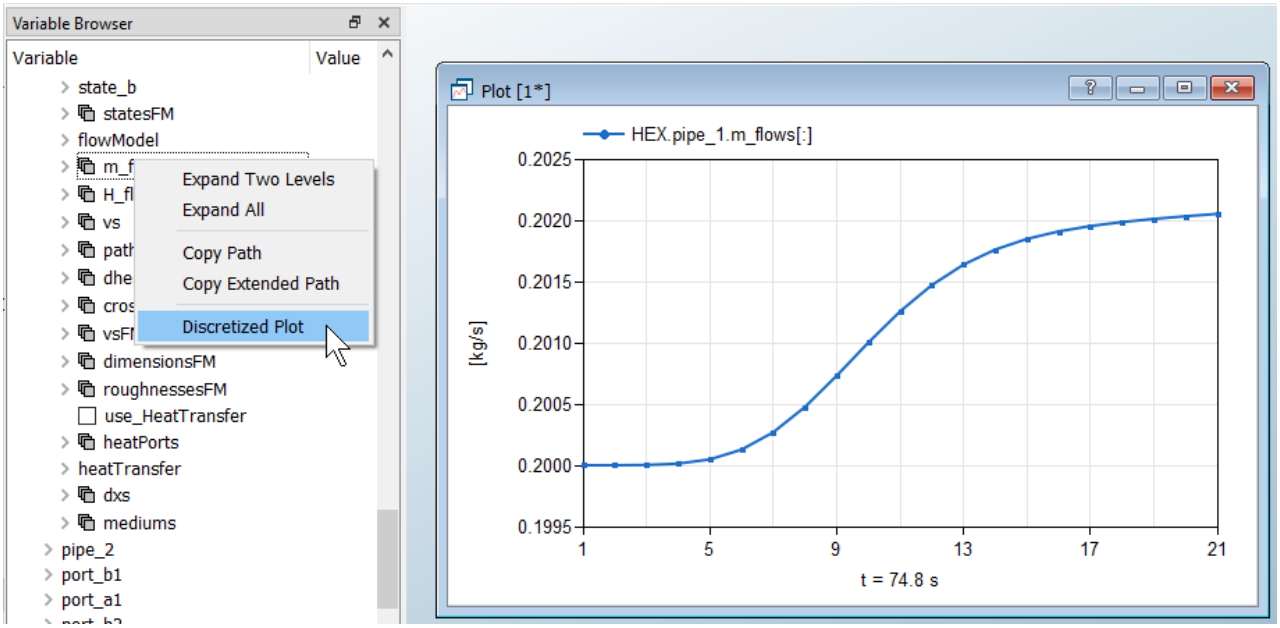
Physical objects are sometimes discretized along one dimension, for example, to model the temperature of flow in a pipe discretized into several sections. Several such Modelica components can be combined into a single discretized plot, which also allows different types of components to be used for the first, last, and middle sections.



The discretized plot shows the values of each variable at a particular time instant, which is controlled by the animation time slider. This means that the shown values correspond to the value at the measurement cursor in normal plots. The time is shown below the horizontal axis of the discretized plot.

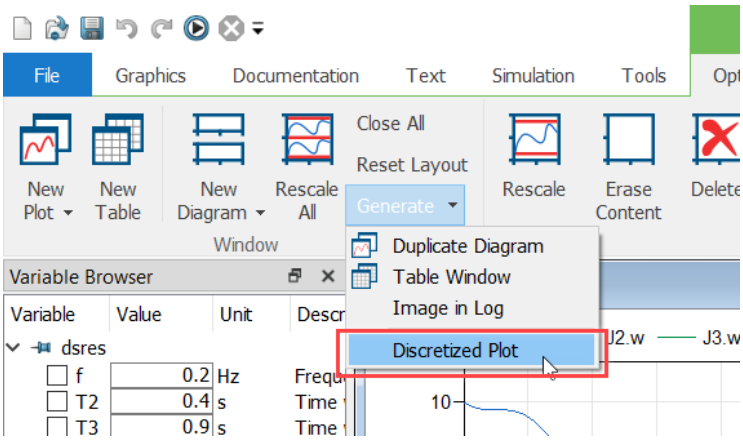
(The model used for the example is Modelica.Fluid.Examples.HeatExchanger.HeatExchangerSimulation.)

You can create a discretized plot directly from an array in the variable browser:



If you apply the operation on an array of record variables, each variable in the record will be added to the discretized plot.

You can create a discretized plot from variables in an ordinary plot using a command found in the **Simulation > Plot: Options** tab, the command **Generate > Discretize Plot**:



Plotting discretization is also supported by scripting, by the built-in function `plotDiscretized`. For more information about this built-in function, see “plotDiscretized”

on page 963 . Note that scripting also allows you to plot several variables in the same discretized plot window.

Plot dependencies

It is possible to display the dependencies of a selected variable by plotting them. This facilitates browsing of results to understand the behavior for normal simulations as well as debugging it. (Note that it is also possible to apply plot dependencies after failed initialization; see "Using plot dependencies after failed initialization" on page 1027 for details.)

Introduction

Dymola converts the differential-algebraic system of equations symbolically to differential equations in state-space form, i.e. solves for the derivatives. The steps include index reduction and state selection.

Efficient graph-theoretical algorithms are used to determine which variables to solve for in each equation and to find minimal systems of equations to be solved simultaneously (algebraic loops). Thus the problem of calculating the derivatives and all other variables but the states at a time instance is a sequence of problems that can be solved in turn. (See further the chapter "Advanced Modelica Support", section "Symbolic Processing of Modelica Models".)

It means that a computational causality has been assigned to the variables and the equations. Such a subproblem may be a scalar problem consisting of one scalar variable and one equation. It may be several variables assigned by a record equation, an array equation, a multi-returning function or an algorithm. In the general case, it is mixed system of equations including Boolean, integer and real variables. All unknown variables to be solved for from the equations depend in the general case on all other time-varying variables appearing in the equations of the subproblem. Thus, these variables are considered to depend of each unknown of the subproblem. The dependet variables are all calculated from subproblems appearing earlier in the sorted sequence of subproblems.

By plotting the dependencies of a variable and recursively plotting their dependencies, it would hopefully be possible to understand or identify the reasons or origin of observed behaviors.

Below an example is given to illustrate this.

As described below it is simple to have the behavior of the dependency variables of a time dependent variable, v, plotted.

When plot of dependencies are activated, the simulation will also store the behavior of protected variables.

The legend includes the equation from which the variable is calculated, however, in case the variable is solved from an algebraic loop, no explicit equation are given, but it is indicated whether it is a linear, nonlinear or mixed system of equations.

Simple equations v1=v2 are frequent in Modelica models due to connections. Such equations are exploited by Dymola to eliminate the variable from the calculation and turn it into an alias variable to allow it to be plotted. Plotting of its dependencies is supported. If v1 = v2 is used

to eliminate v1, then if v2 is solved from a simple equation v2 = e, then the legend will be v1 = v2 = e; and the incidences of v2 are plotted. In other more complicated cases, the legend will be v1 = v2 and v2 will be displayed to allow further traversing of the dependencies.

It is also possible to plot the dependencies of a continuous-time state, x. It gives the dependencies of its derivative, der(x) and the equation displayed in the legend is "x=Integral(der(x))".

Applying plot dependencies

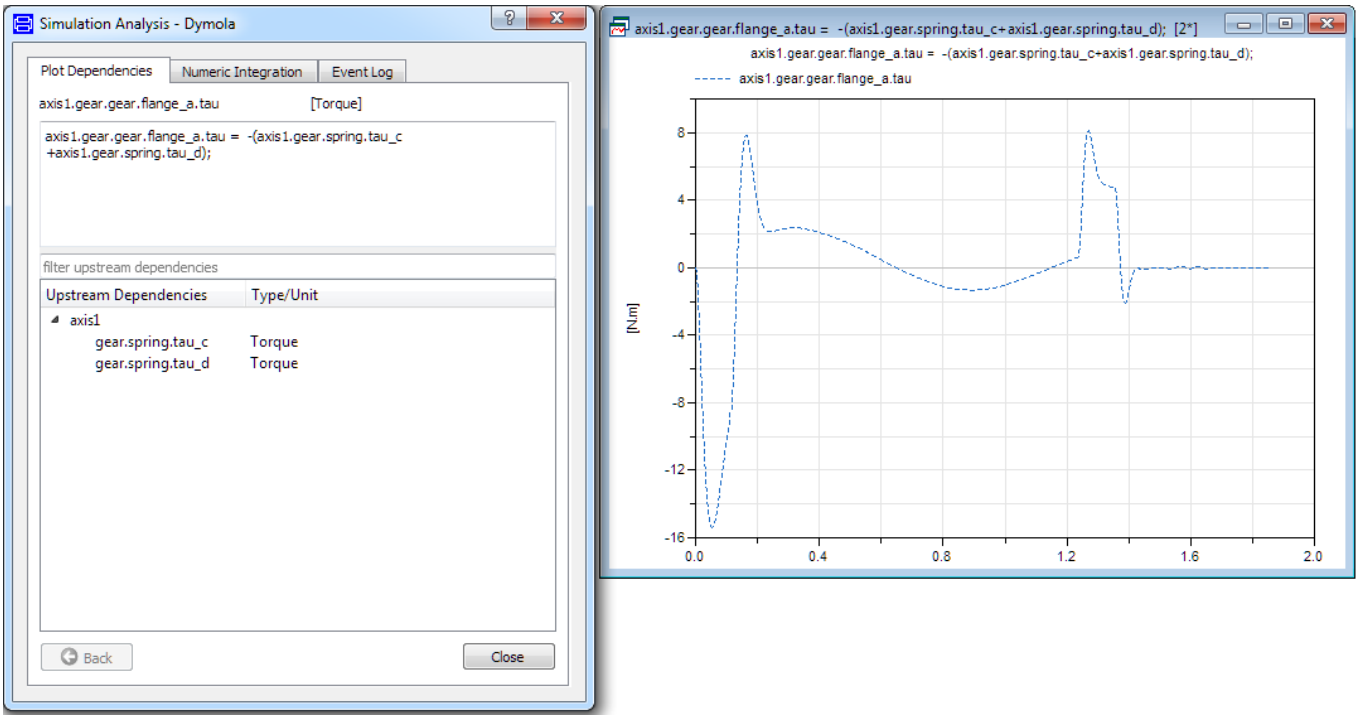
To enable plotting of dependencies, the setting **Provide variable dependencies and equation incidence** must be ticked in the simulation setup. For information about this setting, please see it in the section "Debug tab" starting on page 812.

To display the dependencies for a variable, simulate the model, then right-click the variable in the variable browser, and select **Plot Dependencies** from the context menu. You can also right-click the corresponding curve in the plot and select **Plot Dependencies**.

When giving this command, two items are displayed:

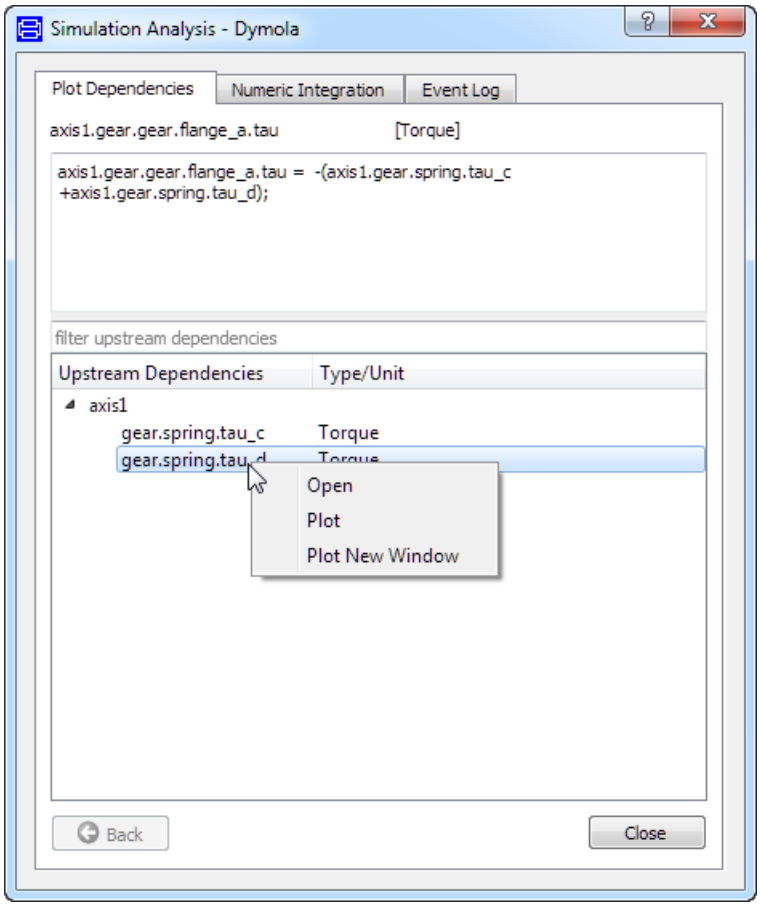
- A browser is displayed, showing the dependency expressions, and all associated values are shown. The variables are grouped by top-level compents.
- A new plot window is displayed; showing the variable (dashed) and the equation of the variable, both as window header and as plot header.

A simple example, for the variable `axis1.gear.gear.flange_a.tau` in the Robot demo (a longer example will follow):



For the Type/Unit column, common long names are for cosmetic reasons abbreviated by removing any occurrence of `Modelica.Units.SI`, so, for example, `Modelica.Units.SI.Torque` becomes just `Torque`.

Right-clicking a variable in the dependency, for example gear.spring.tau_d, gives:



The commands available in this context menu are:

- **Open** displays the plot dependencies for the selected variable. Pressing **Back** button returns to the previous view.
- **Plot** plots the selected variable in the currently active plot window (in the default case the window shown in the figure above).
- **Plot New Window** creates a new plot window and plots the variable in this window. The dependency browser now displays the dependencies for that variable.

The variable for which the dependenies are plotted is dashed in the plot.

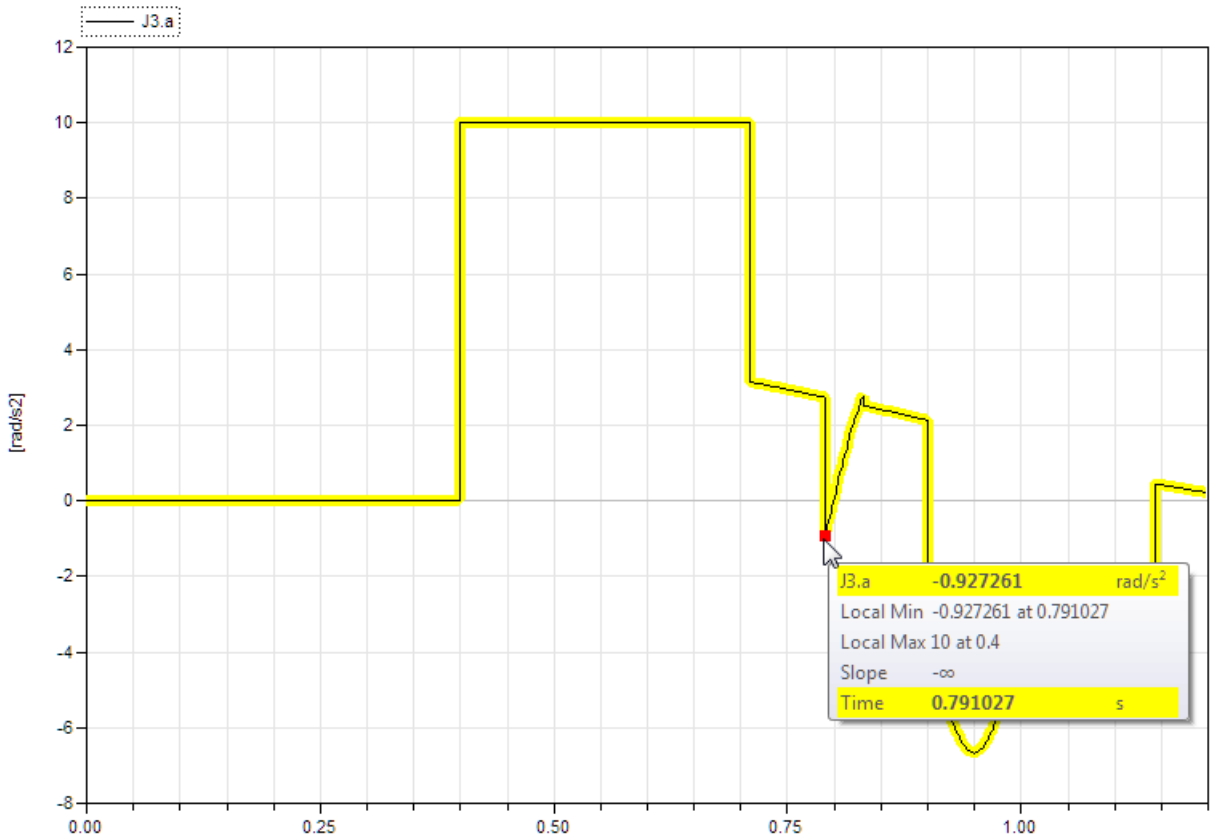
Notes:

- The feature of dragging curves between diagrams can be useful if working with several plot windows; see section “Drag and drop of curves between plots/diagrams” on page 718.
- The might be a number of plot windows after a while. You can close them all by the command **Windows > Close All**.

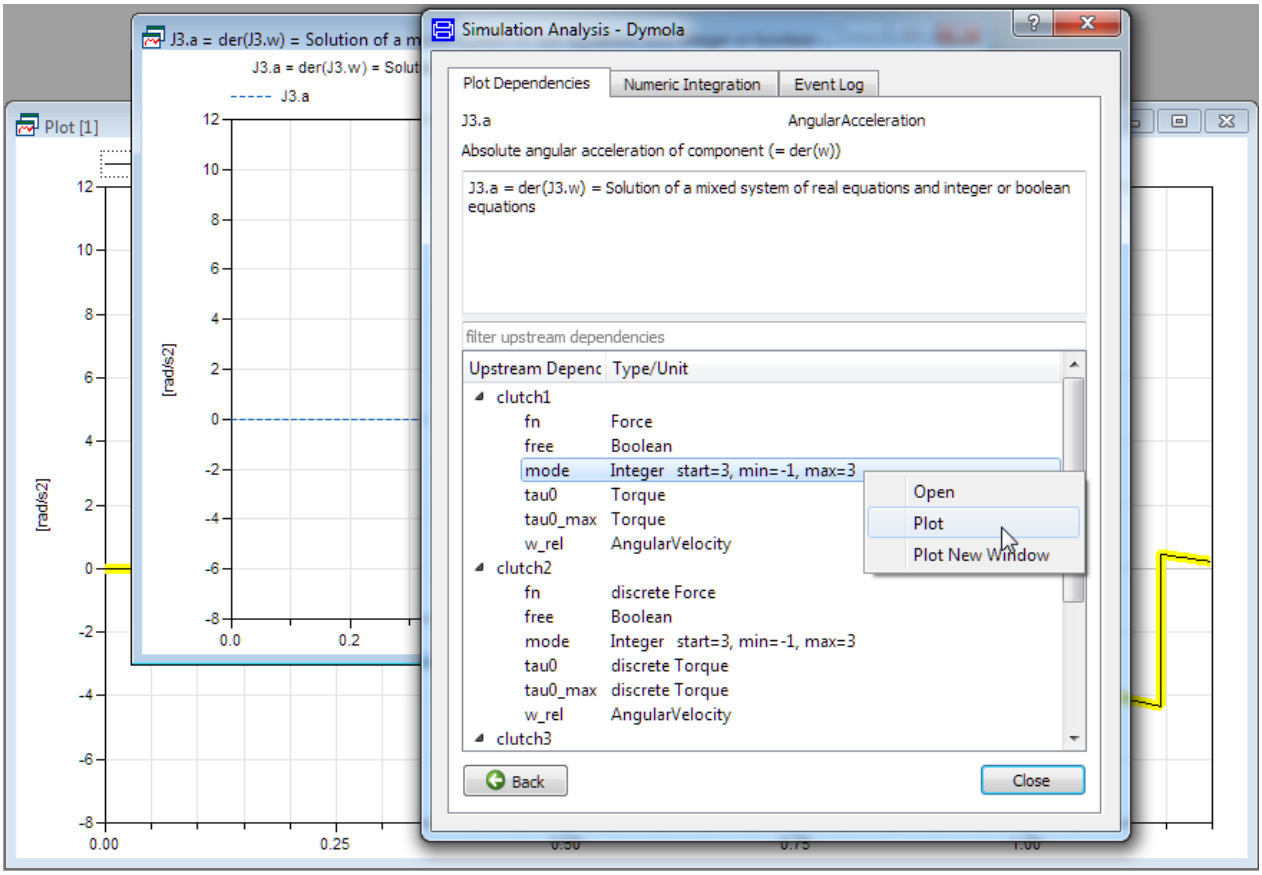
- There is a filter “filter upstream dependencies” that makes it possible for you to easily look only at upstream dependencies matching your regular expression. The filter works like other similar filters in Dymola; the search is dynamically updated while typing, and regular expressions can be used for the search.
- You can specify how to present alias in equations, see section “Handling of alias names of variables in error messages and other outputs” on page 757.

Example

Running the demo Couple clutches (**File > Demos > Coupled clutches**) we notice that the acceleration of the mass J3 jumps at certain times:

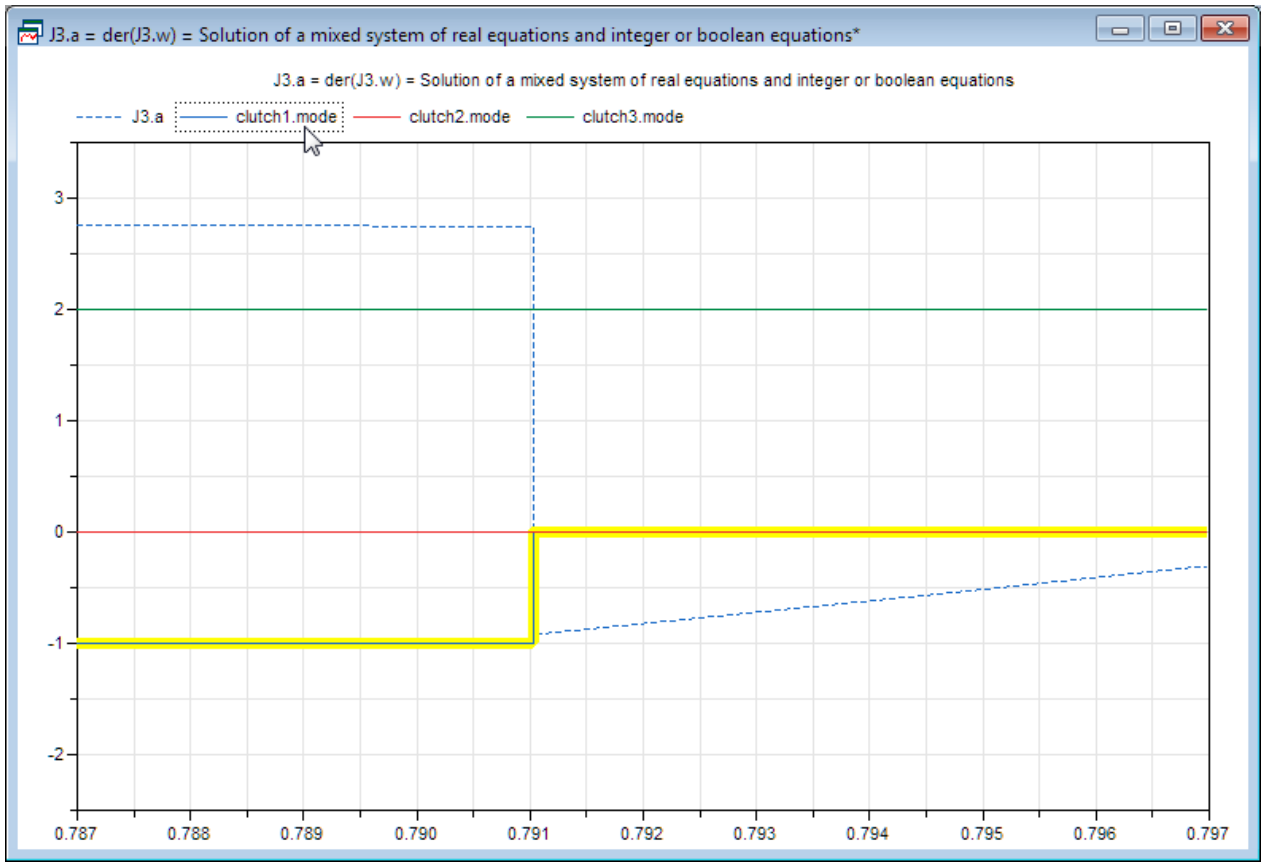


Let us investigate the change of acceleration at t=0.79. First we display the dependencies for J3.a (easiest done by right-clicking the curve and select **Plot Dependencies**). From this set of signal, we plot the mode variables by right-clicing them and selecting Plot, one by one. An example:

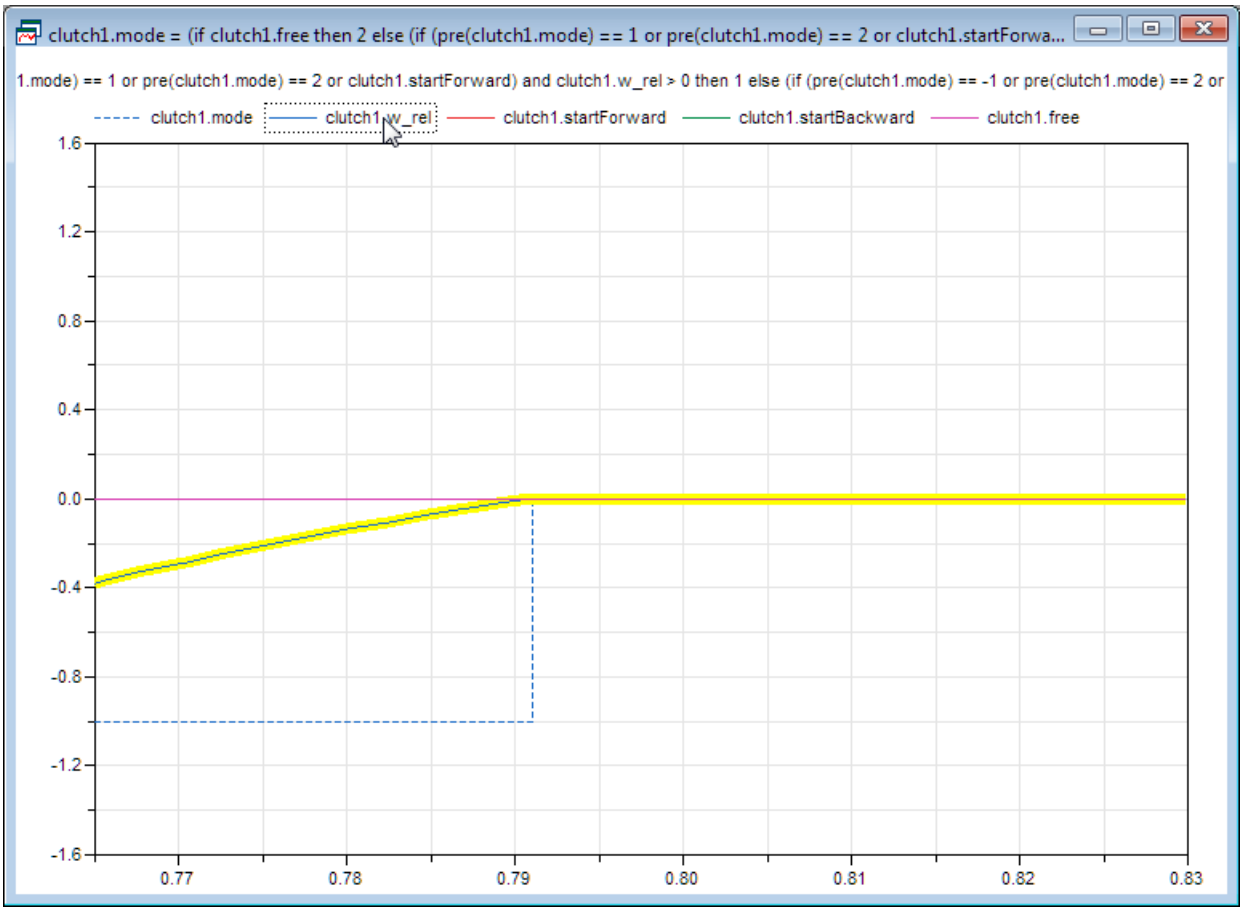


When having plotted them, we can center the result on t=0.79 by zooming on that time (by **Shift**+drag, both vertically and horizontally).

Having the result, we pause over the curve legends, one legend at the time, to see each curve highlighted in the plot, until we find something interesting:



Plotting all dependencies for clutch1.mode (and zooming), we get:



We realize that the change was due to clutch1.w_rel reaching zero and the clutch switching to stuck mode.

(Note that double-clicking the equation in the header displays it in a more readable way.)

Equation incidence graphs

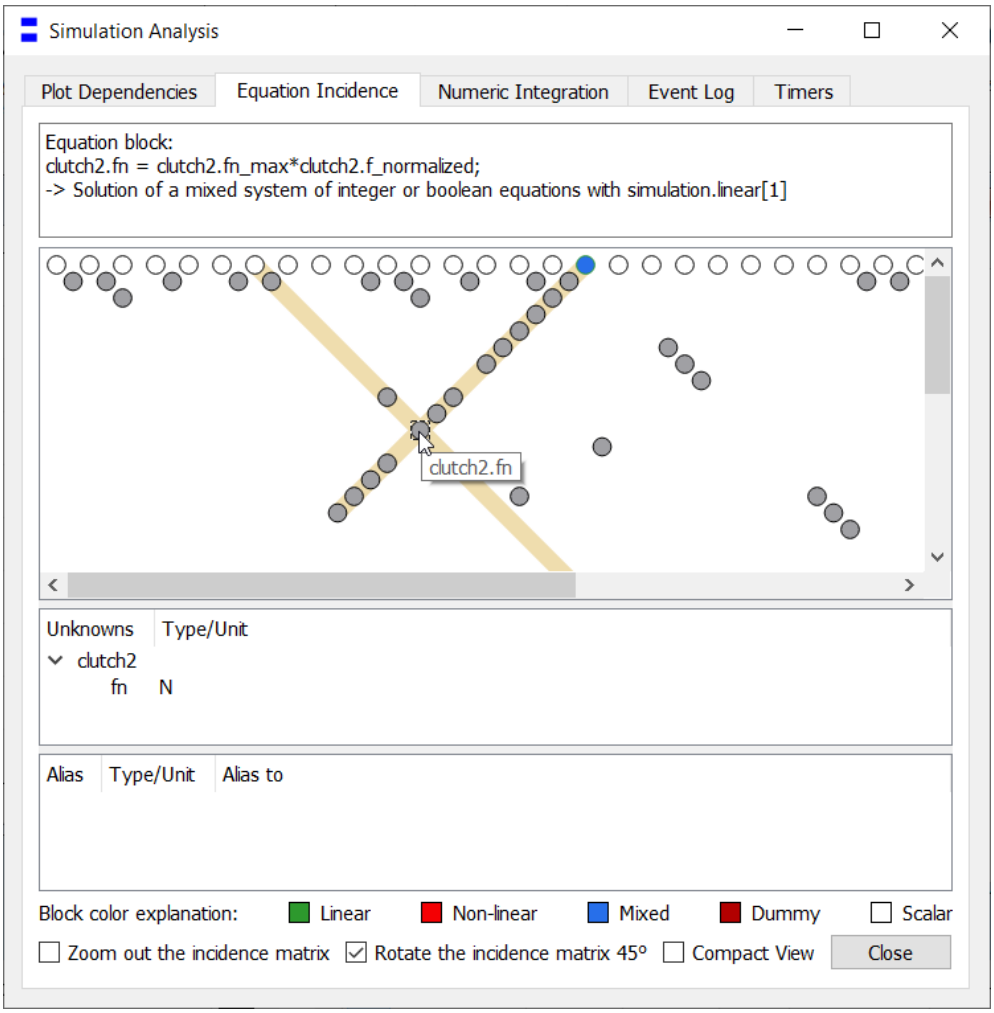
General

You can display the equation incidence graphs for the systems of equations in the simulated model.

Note: To use this function, you must enable the setting **Provide variable dependencies and equation incidence** in the simulation setup, reached by the command **Simulation > Setup**, the **Debug** tab. See section “Debug tab” starting on page 812 for more information about this setting.

The indicence graph is displayed by right-clicking the simulation result (top-level node) in the variable browser and selecting **Simulaton Analysis** from the context menu (or, as an alternative, by the command **Simulation > Simulation Analysis**).

In the window that opens, select the **Equation Incidence** tab. An example from the demo Coupled Clutches:



The **Equation Incidence** tab shows the equation blocks. This is the sorted Block Lower Triangular (BLT) form of equation blocks, turned 45 degrees counter-clockwise to make it easy to scroll vertically to follow the diagonal.

Selecting a “diagonal” block shows the equations blocks, and the variables solved from that block (“Unknowns”). Selecting a “non-diagonal” block shows the two blocks and the “relevant” variables solved from the first block, as in the example above. (Concerning

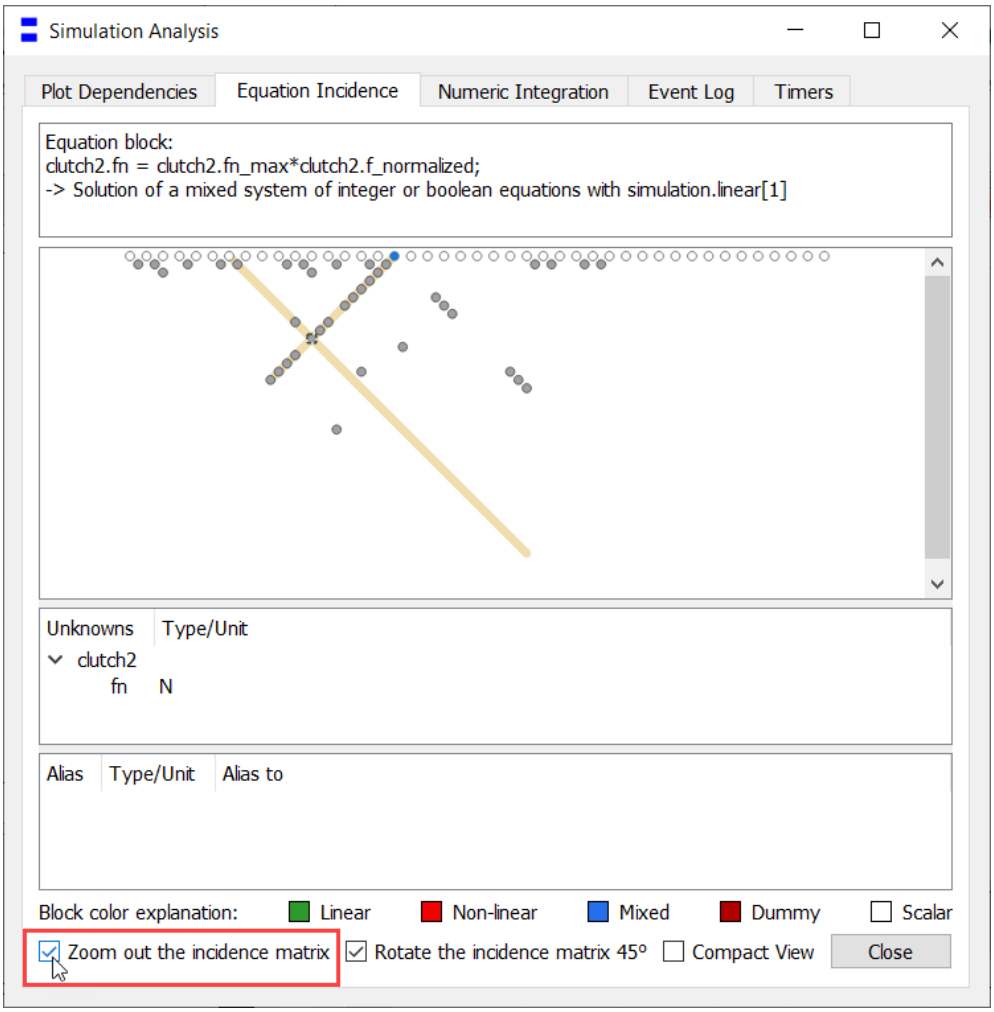
"diagonal", note that by default the diagonal is displayed rotated, that is, it is the upper horizontal lines in the figure above.)

The fourth pane displays the aliases for the selected block (if any). Note that you can specify how to present alias variables in the top pane and tooltip by a flag. See section "Handling of alias names of variables in error messages and other outputs" on page 757.

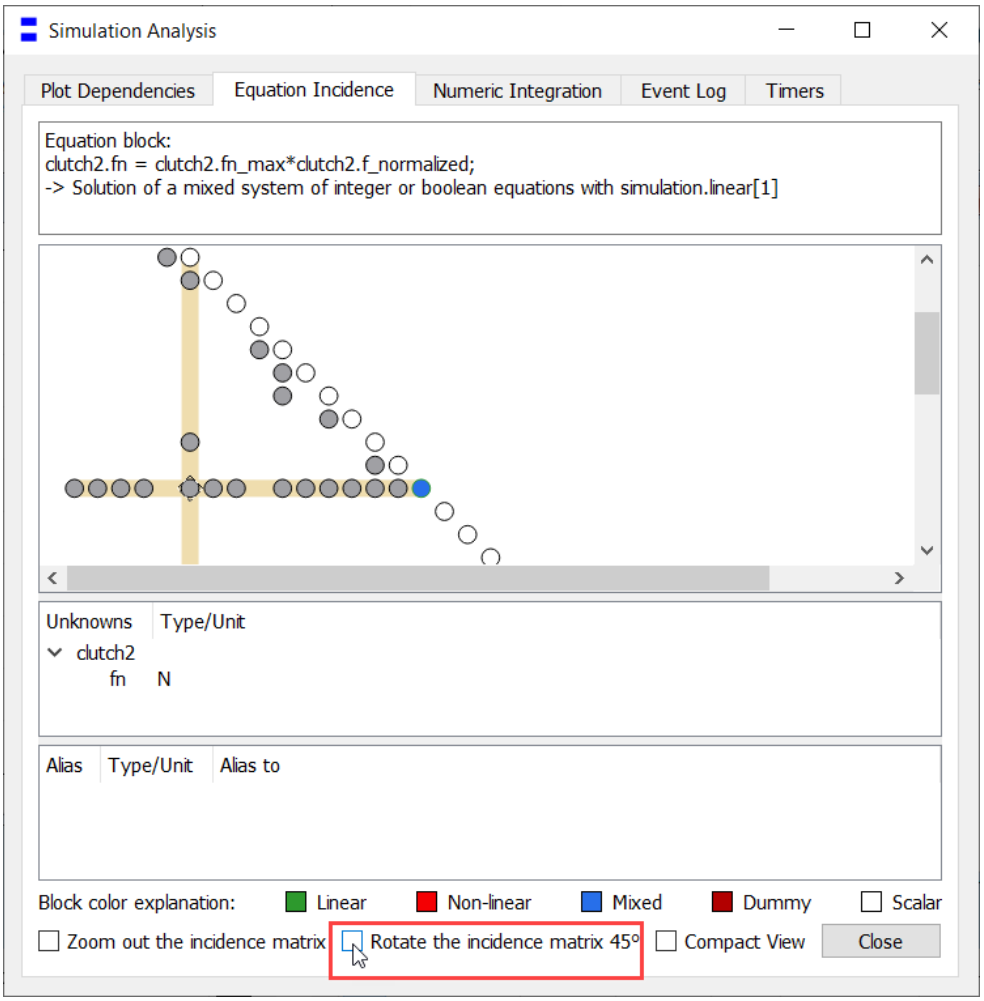
The block color has the following meaning:

- Green (**Linear**): Linear system of equations
- Red (**Non-linear**): Non-linear system of equations
- Blue (**Mixed**): Mixed system of equations, border define if linear or non-linear
- Dark red (**Dummy**): Dummy derivative system of equations for dynamic state selection
- White (**Scalar**): Simple scalar equations, including alias equations

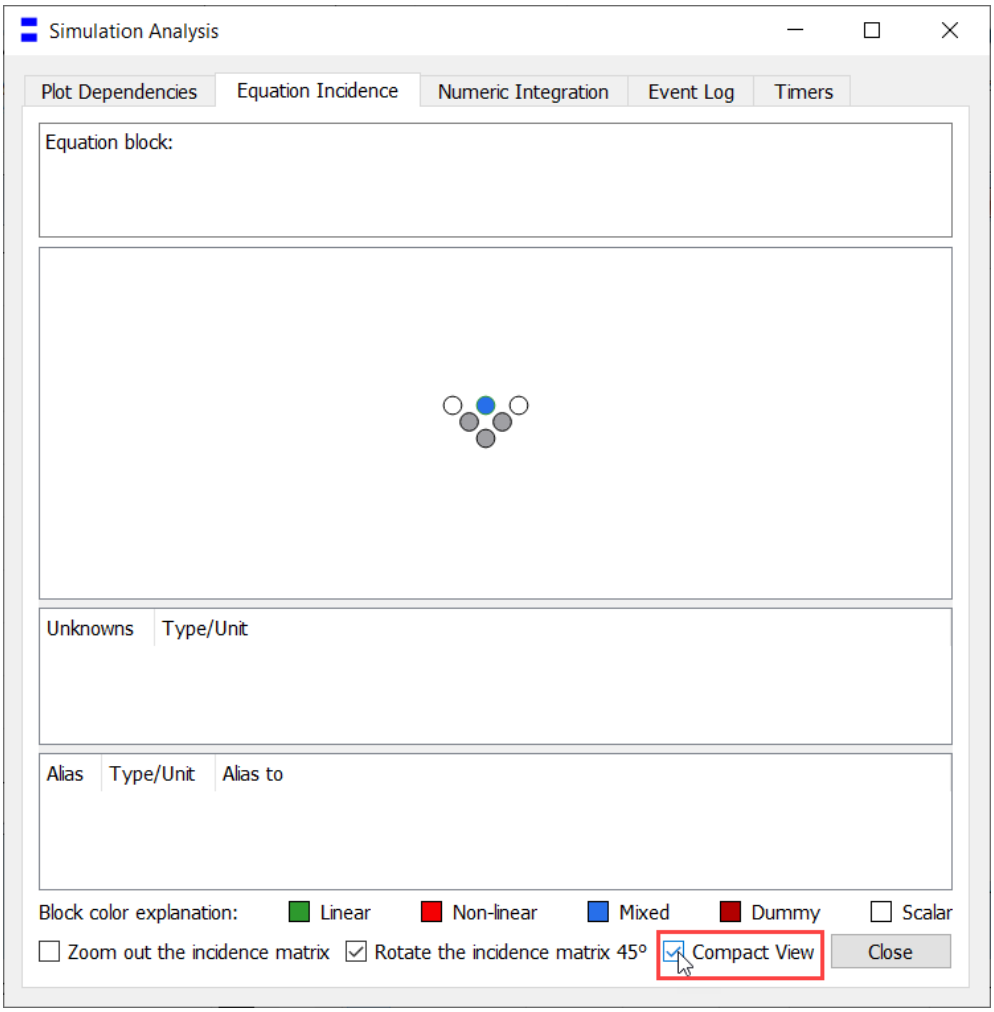
You can zoom out the incidence matrix by activating the option **Zoom out the incidence matrix** at the bottom of the window. The result for the above example is:



By default, the incidence matrix is rotated 45 degrees. You can reset this by deactivating the setting **Rotate the indicence matrix 45**. The result is:



Finally, you can display the incidence matrix in a compact view, that is, displaying the incidence matrix with all simple systems merged, by activating the option **Compact View**. Doing this for the current example gives:



Note that when you have selected a node, the content of the Plot Dependencies tab is adapted to your selection, if applicable.

For more information about equation incidence, see the paper Exploring Model Structure with Equation Incidence.

Example: Exploring the translated model structure by using equation incidence

Do the following:

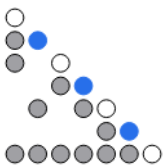
- Open the library **Modelica_StateGraph2**.
- Open model `Examples.Applications.ControlledTanks.ControlledTanks1`.
- In the simulation setup, the **Debug** tab, activate the setting **Provide variable dependencies and equation incidence**.

- Simulate the model.

The example consists of two tanks, three valves, and a state machine controlling the valves. During simulation, valve1 opens to fill up tank1 with water from the reservoir. When tank1 is full, its content is drainded into tank2, making a pause when the former tank is almost empty. Finally, tank2 is emptied and the cycle repeats. (You can use the Animation Control, the **Play** button, to run the demo, using speed 10 is a good idea.)

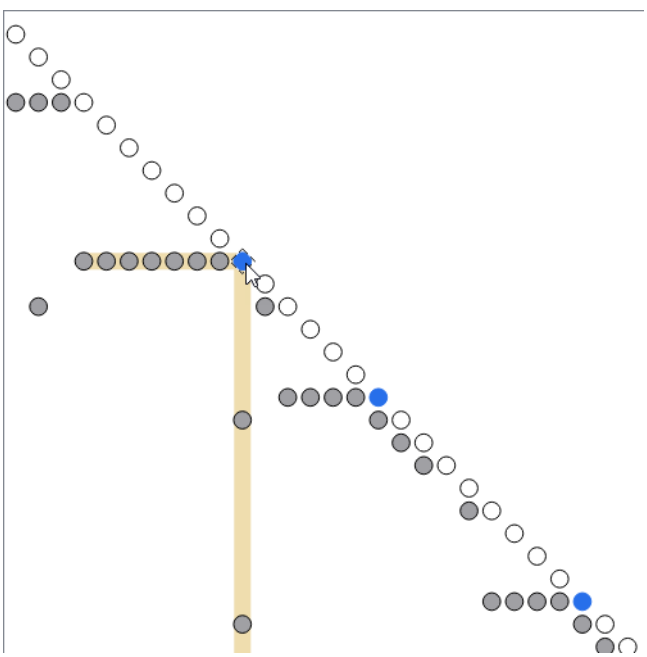
Now, display the Simulation Analysis, the **Equation Incidence** tab. Deactivate **Rotate the incidence matrix 45**, if activated (making it easier to understand the below).

Start by activating **Compact View**. This gives:



From this image, we can see that we have three mixed systems in this model.

Deactivating **Compact View**, we can click the first blue node to select the first mixed system, as below:



In the current model, this example, the upper pane (not displayed in the figure), you get information of what kind of equation block you have selected. In the lower pane (not displayed in the figure), you get a list of all the variables the equation block solves for. This includes iteration variables, torn variables, as well as discrete variables. For this example, you can see from the list that this sytem solves for the flow and pressures around valve2, between the two tanks.

The gray off-diagonal nodes represent dependencies. If you click a gray node, you see the corresponding dependency equation in the upper pane.

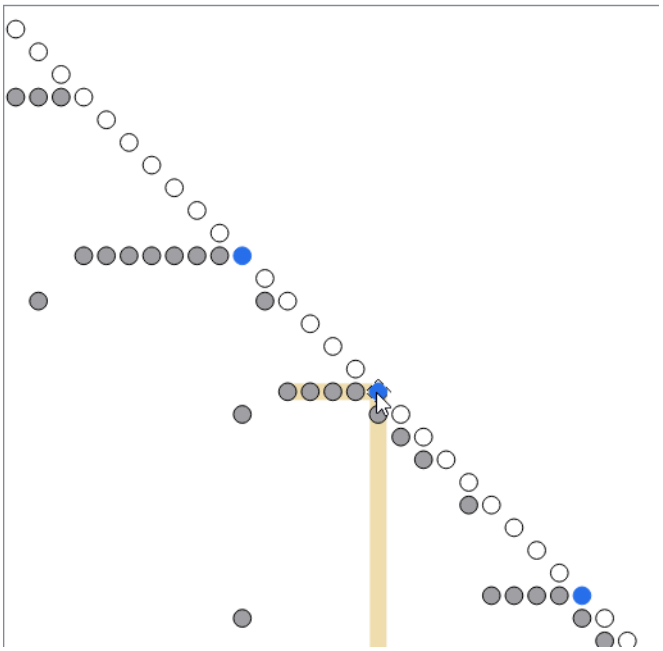
The gray nodes to the left of the blue node, on the same row, represent variables that need to be computed before the first system can be solved. That is, they represent upstream dependencies for that system. The gray nodes below the blue node, in the same column, show where variables from the first system are used in subsequent computations. That is, they represent downstream dependencies for that system.

Note: For upstream dependencies, the difference compared to the plot dependency feature is that an equation incidence can represent several variables as part of the same upstream dependency.

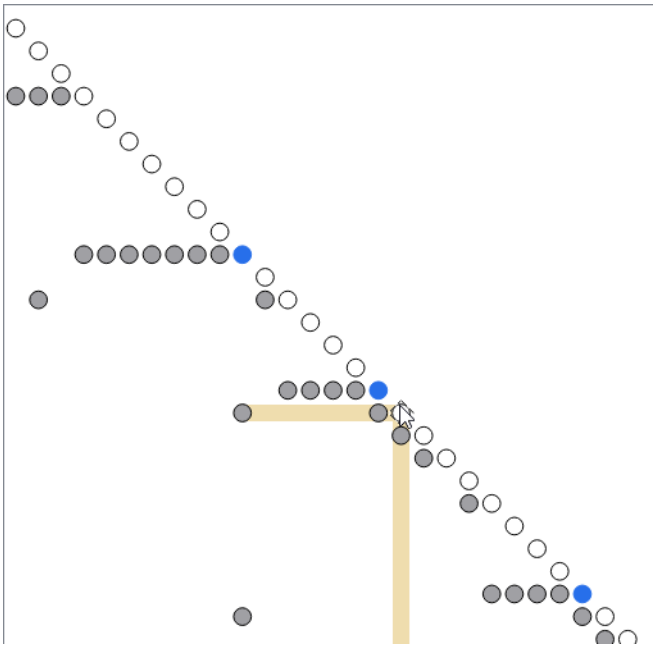
An important use of the equation incidence graph is to investigate if the systems are independent of each other. The first system calculated is by definition indpencent.

To investigate if the second system is independent of the first system, look at the figure above. You can see that the first gray node in the column of the blue node appears after the line of the second system (second blue node), meaning that dependence is not relevant until after the second system has been computed. That is, the second system is independent of the first system.

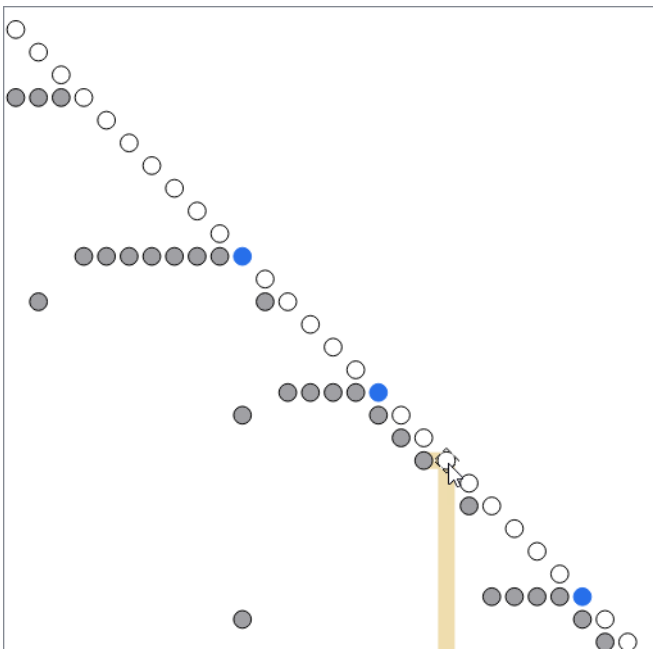
To investigate if the third system is independent of the second system, select the second system as below:



You can see that the column contains a dependency node just below the system line. To investigate that dependency, select the corresponding node in that row in the diagonal:



You can see that the corresponding column also contains a node directly under the selected node. If you repeat the action by clicking the corresponding node in the diagonal, you still see a node directly under the selected node. However, when you repeat the action again, you see the following:



This means that the third system is independent of the second system.

In a similar way, you can prove that the third system is independent of the first system.

Note: To be able to easier slide along the diagonal, you can activate **Rotate the incidence matrix 45**. This is the default, but it is easier to explain the incidence graph when this option is not activated.

Analyzing numeric integration

Notes:

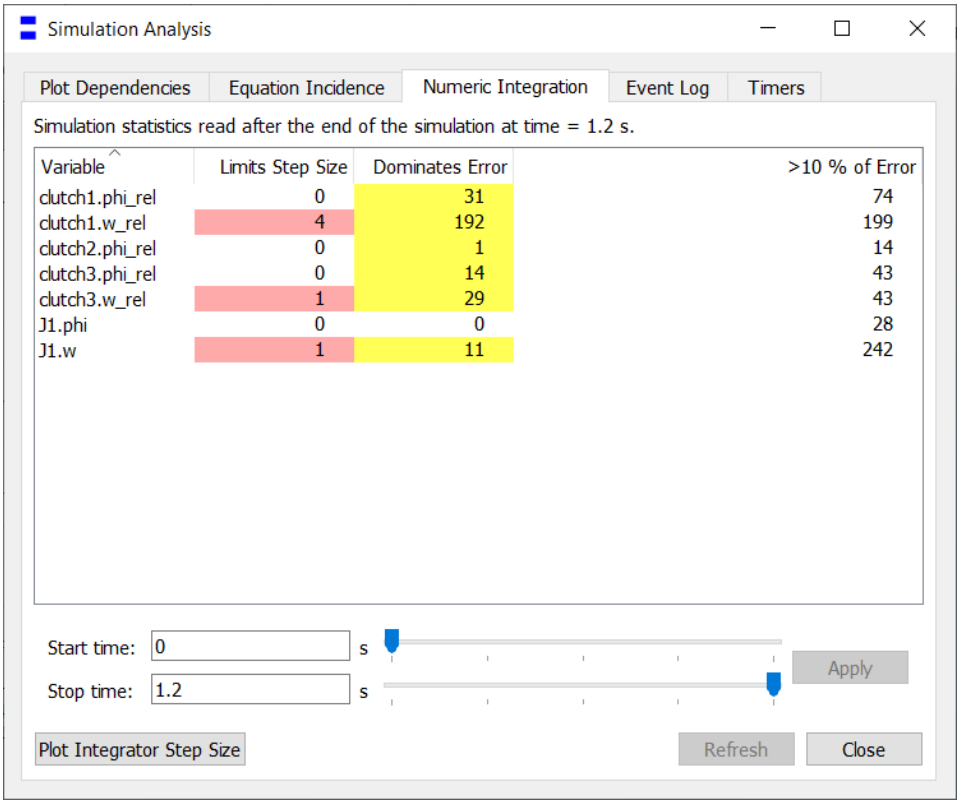
- To use this function, you must enable the setting **Which states that dominate error** in the simulation setup, reached by the command **Simulation > Setup**, the **Debug** tab. See section "Debug tab" starting on page 812 for more information about this setting.
- The solvers Euler, Rkfix*, and Cvode do not support numerical integration statistics and therefore produce no diagnostic files. Other solvers are supported.

Statistics from the numerical integration is available, facilitating inspection of state variables that most likely slow down the simulation of large models. It is available by right-clicking the simulation result (top-level node) in the variable browser and selecting **Simulaton Analysis** from the context menu (or, as an alternative, by the command **Simulation > Simulation Analysis**).

In the window that opens, select the **Numeric Integration** tab. The tab shows the following information (it can be sorted in ascending or descending order by clicking on any of the column headings).

- The state variable name.
- The number of times the variable limits the step size of the numerical integrator. Counts greater than zero are highlighted.
- The number of times the variable dominates the error, i. e. has an integration error larger than any other state variable. Counts greater than zero are highlighted.
- The number of times the variable contributes to more than 10 % of the integrator error.

As example, open the Coupled clutches demo (**File > Demos > Couple clutches**), use the command **Commands > Simulate and Plot** to simulate it, then right-click the simulation result `dsres` in the variable browser and select **Simulation Analysis**, the **Numeric Integration** tab. The result is:

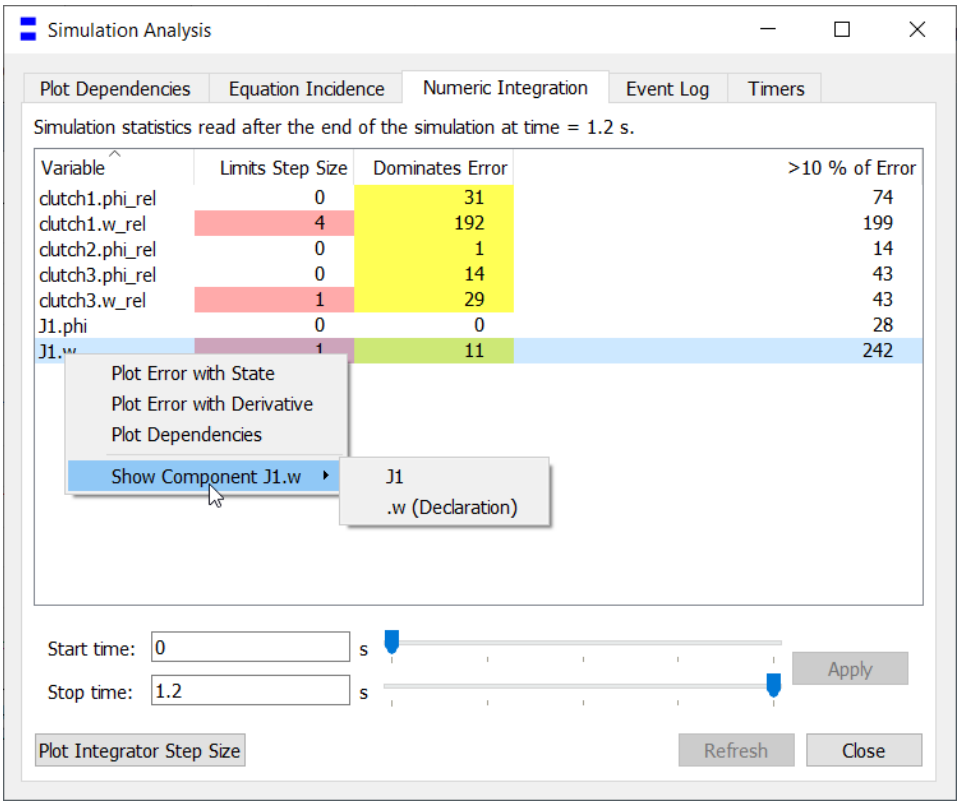


A time range filter is available in the lower part of the dialog; here you can specify the start time and stop time to limit the integrator diagnostics data, by entering values or dragging slides. Click **Apply** to use the selected range.

The values presented in the summary will be limited to the data gathered in the specified time range. Step size and error plots will also be limited to the specified range. Note that the limits are specified inside Dymola and doesn´t limit the output from the integrator.

The reason for using the time range filter is that there may be different states dominating the error during different times of the simulation. Limiting the time range of the integration diagnostics data makes it possible to see, in the summary, what is relevant for the chosen time interval. For the corresponding plots, the benefits are that it is faster to zoom in at the interesting range and that downsampling the plot data may be avoided. This reduces the risk of missing something important. (In the examples below time range filtering is not used, however.)

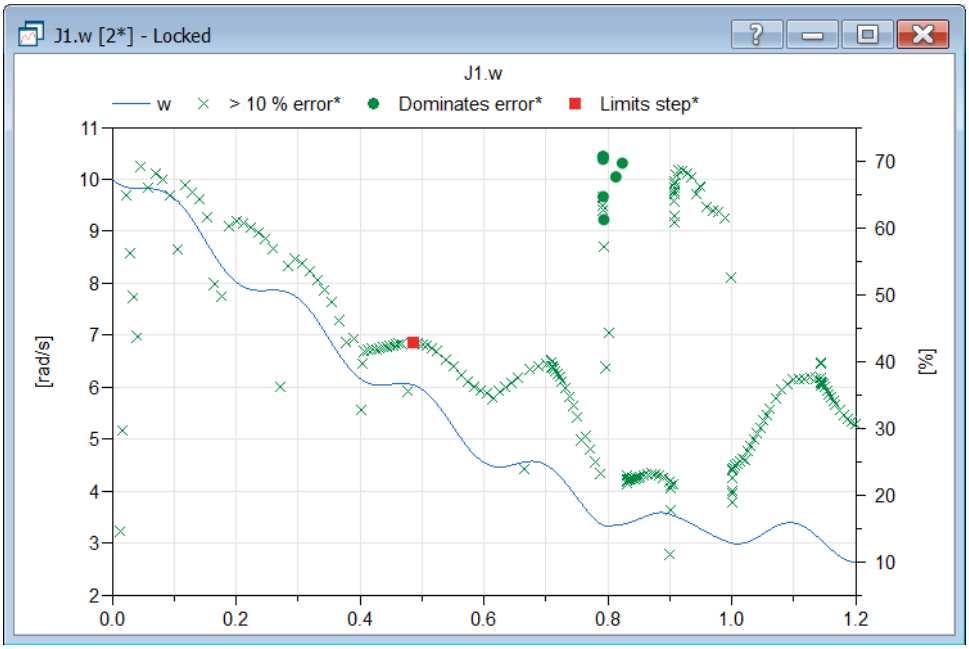
Right-clicking a variable, for example, `J1.w` displays a contex menu with four commands:



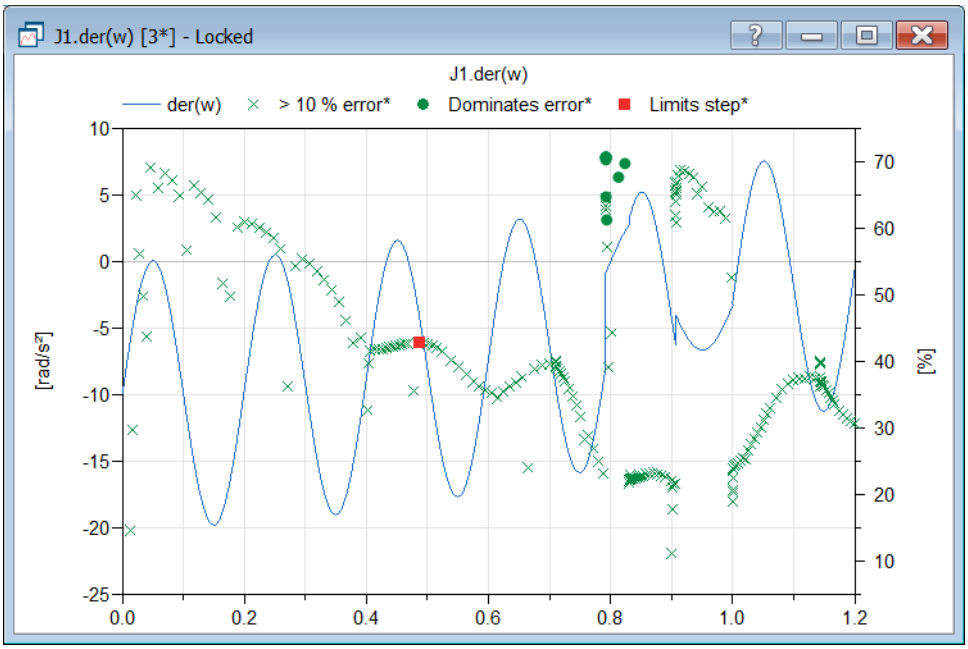
Plot Error with State creates a new plot window and displays the time-plot for the variable, as well as the information regarding limits step size, dominates error, and more than 10 % of error.

In the plot, a marker is plotted at each time point where such an event occurs, and the value on the right vertical axis is the contribution to the numeric error in percent. This plot facilitates a better understanding of the variable´s effect on the numeric integrator at various stages of the simulation.

In the following example (selecting **Plot Error** for `J1.w` as above), we can see that the variable `J1.w` dominates the error around t=0.8, but not at t=0.9-1.0 although the error is still > 60 %. The variable `J1.w` limits the step size near t=0.65 and t=1.15.



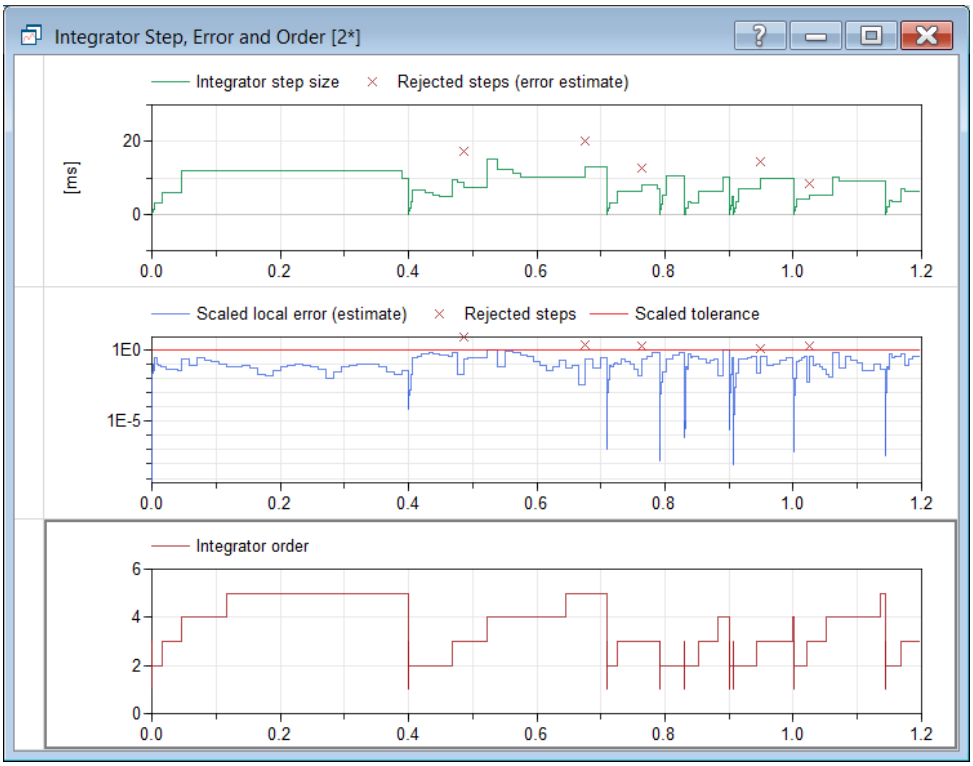
Plot Error with Derivative creates a new plot window and displays the time-plot for the derivative of the variable, as well as the information regarding limits step size, dominates error, and more than 10 % of error, like the above. An example:



Plot Dependency displays the dependencies of the variables in a new window, as described in previous section. The menu choice is disabled if dependencies have not been used.

The **Show Component** context menu entry can be used to show either the corresponding component (in the above case **J1**) highlighted in the diagram layer, or the corresponding variable (in the above case **.w**) highlighted in the text layer.

In addition, the behavior of the numeric integrator can be studied by pressing the **Plot Integrator Step Size** button, which opens a new plot window showing tree diagrams:



The first diagram shows the internal integrator step size. Note that rejected steps (and subsequent restart with a smaller step) are indicated.

In the second diagram, the blue “Scaled local error (estimate)” plots the error estimate from all successful steps. The red crosses “Rejected steps” marks the error estimates from rejected steps. The red line “Scaled tolerance” is always constant 1.0 representing the limit from a step to be rejected. The integrator computes local error estimates during simulation, these are rescaled using the tolerance provided by the user. Thus all local error estimates larger than 1.0 results in a failed step, wheras all smaller estimates result in a successful step. The error estimate can for example be used to see if the integration step size is being limited by the error estimate or something else, like frequent restarts due to events.

The third diagram shows the integration order.

Event logging user interface

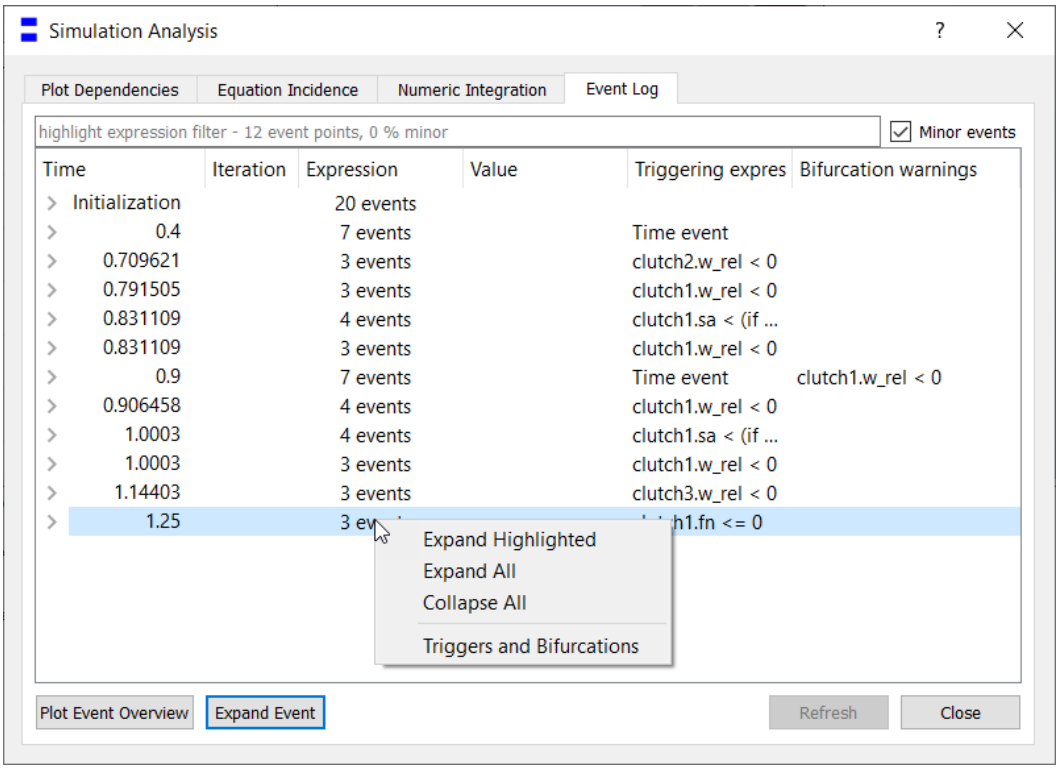
Event logging

Note: To use this feature, you must enable the setting **Events during initialization and simulation** in the simulation setup, reached by the command **Simulation > Setup**, the **Debug** tab. See section “Debug tab” starting on page 812 for more information about these settings. (Note that you here you can also find information about flags for fine-tuning what events to log.)

Note that you can list possible event-triggering expressions in the translation log, see section “Event logging” on page 985.

To follow the scenario below and have the same results, open the Coupled Clutches demo by **File > Demos > Coupled Clutches**.

Simulate (using the command **Simulation > Simulate**) and then perform the command **Simulation > Simulation Analysis**. (You can also right the simulation result file in the variable browser and select **Simulation Analysis**.) The Simulation Analysis window is opened, with the tab **Event Log** tab active.



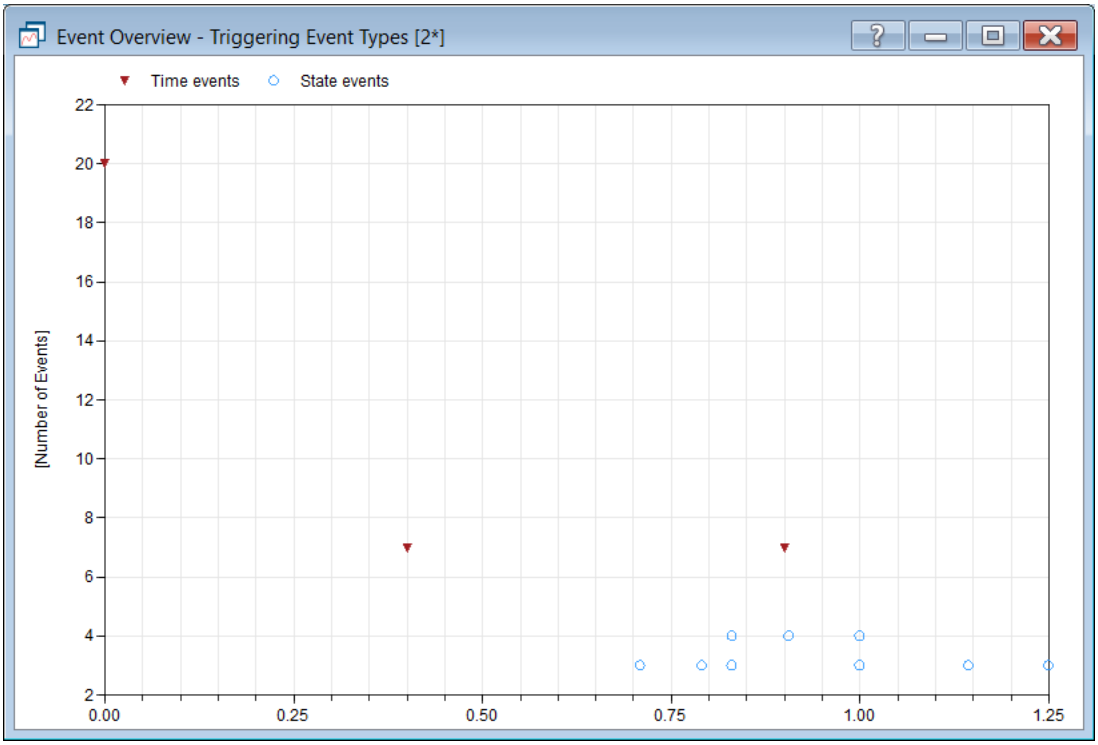
The tab displays a list of all event points the happened during the simulation, sorted by time. For each event point, the number of events is shown. The triggering expressions are listed, as well as any bifurcation warning. For the latter, see the section about bifurcation warnings below.

The percentage of minor events is displayed in the upper line, and you can also, if you have minor events, filter them by clicking the **Minor events** checkbox to the right of the upper line. A minor event also has the text “Minor event” in the Value column.

A context menu is available for a line, as shown above. For **Expand Highlighted**, see below. For **Triggers and Bifurcations**, see the section about bifurcation warnins below.

The event overview

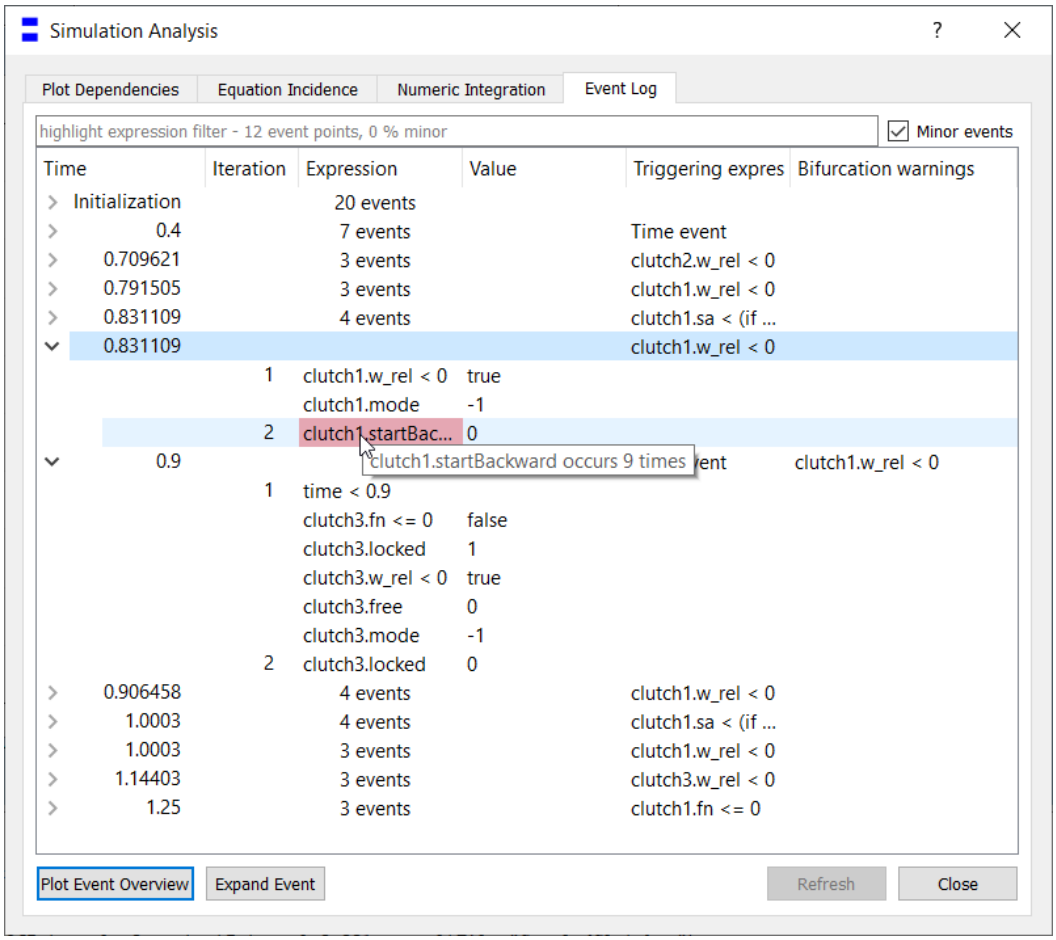
Clicking the **Plot Event Overview** button displays a window where the event iterations at each event point are plotted. The vertical axis displays the number of events in the iteration and the type of marker gives the type of event (time or state) that triggered the iteration.



Inspecting individual events

To see more details on which events that were involved in the selected event iteration, you can expand the corresponding row in the Event Log table. It shows the expression that triggered the event and value of the expression after the event.

To help analysis, the most frequently occurring events (the 10-percentile) are marked in red and the tooltips shows the total count in the simulation.

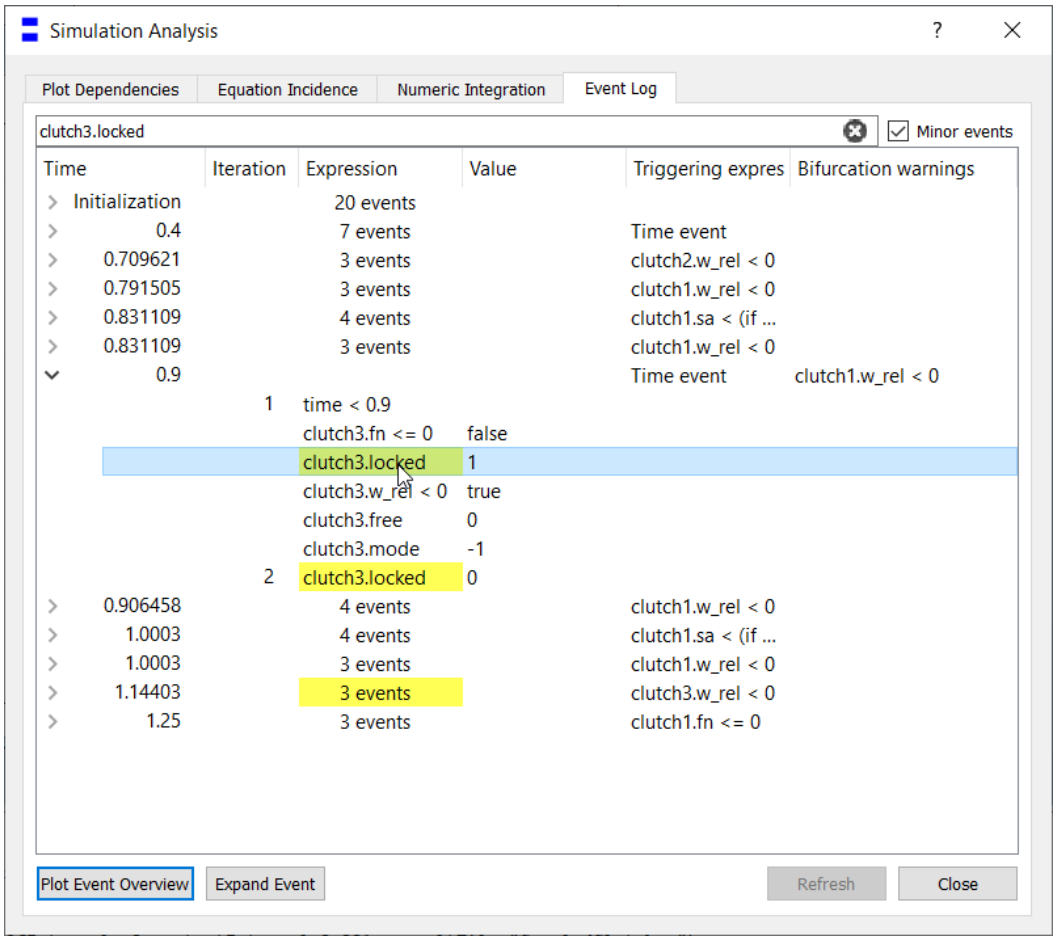


(Note that at t=0.9 no such events occureed, but expanding t=0.831109 (the second of that time, as in the figure) displays one such event.)

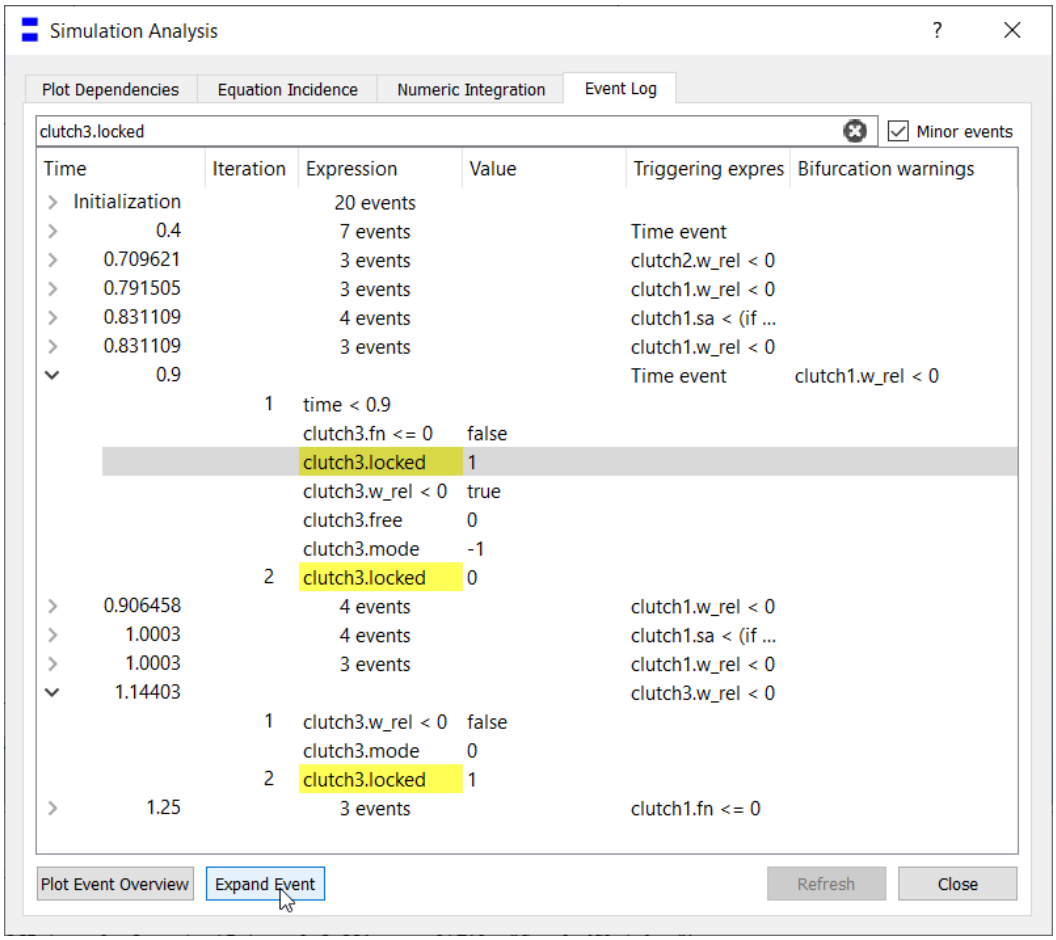
Double-clicking a row highlights all occurrences of this event expression in yellow, including collapsed event point rows. Highlighting can also be made by:

- Typing in the event expression in the search filter above the table.
- Right-click the event expression and select **Highlight Expression**.

Highlighting is removed by clicking on the clear button at the end of the search filter.



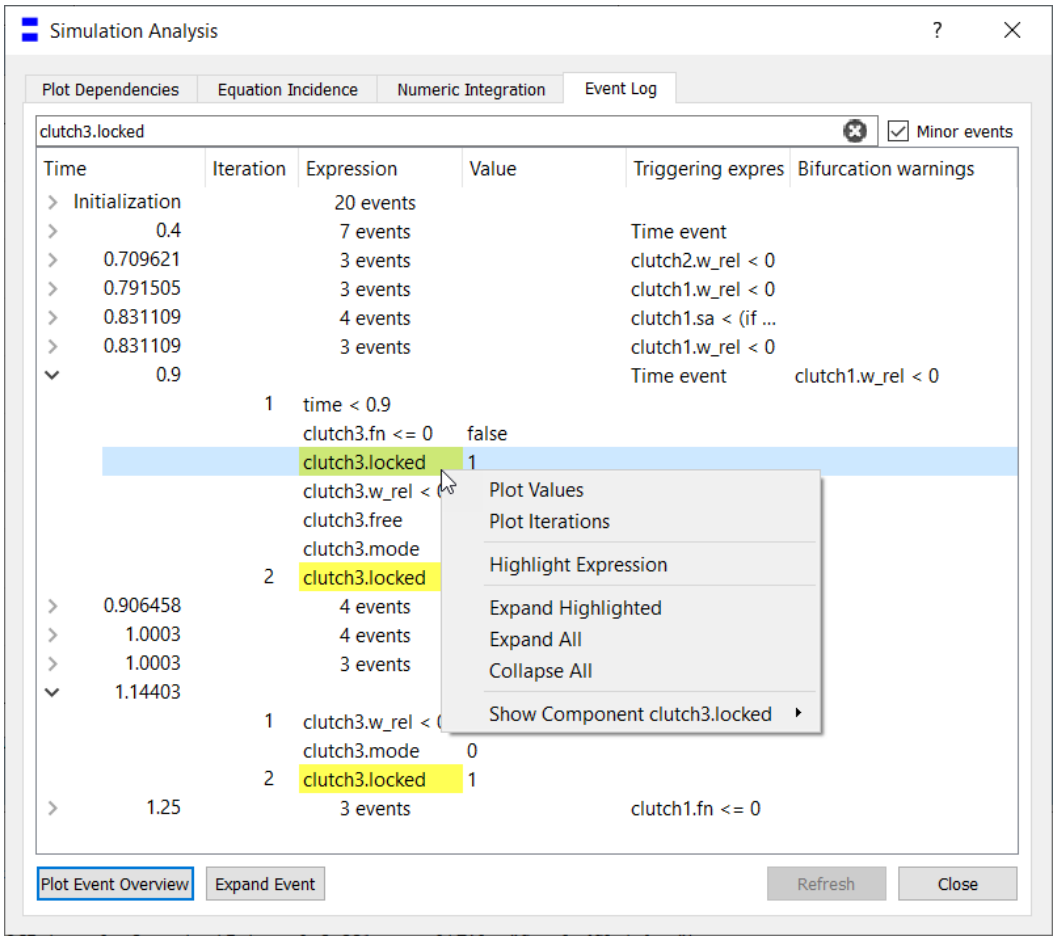
Clicking on the **Expand Event** button works like double-clicking the event, but also expands the events containing a highlight:



The search filter works like other search filters in Dymola:

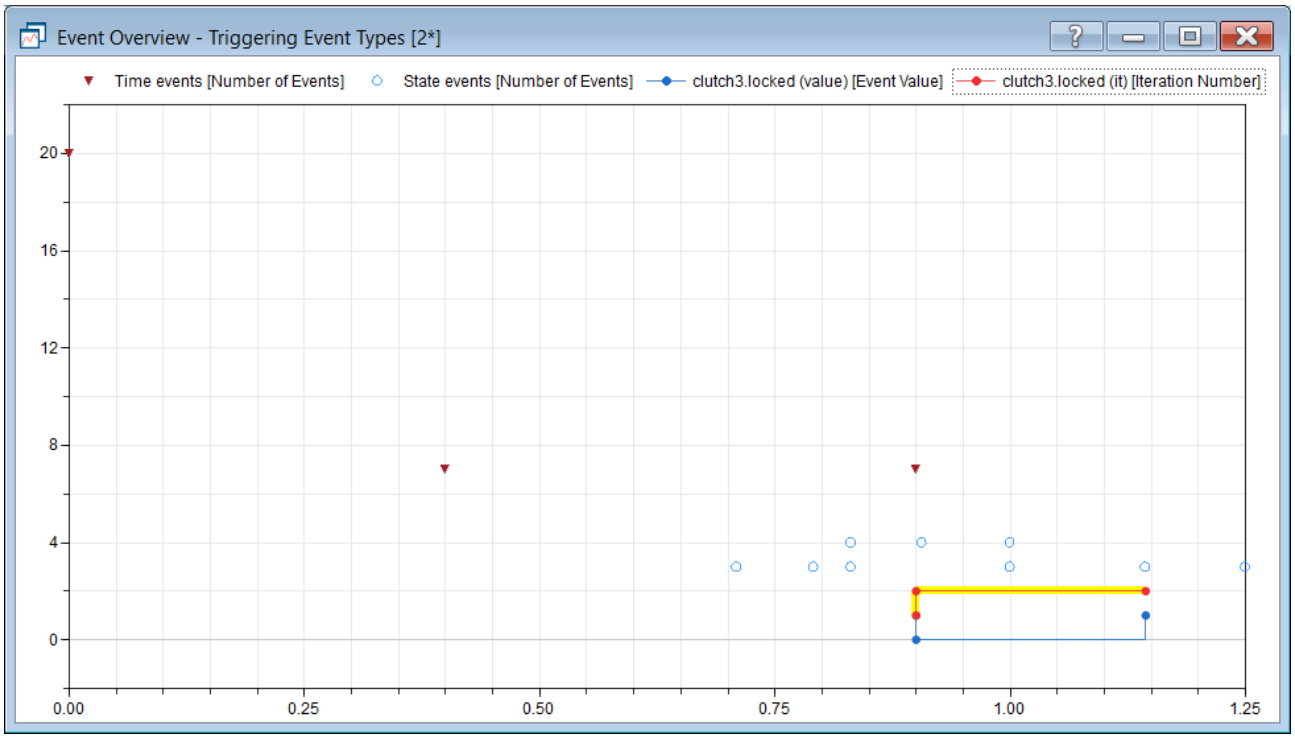
- The search is dynamically updated while typing.
- Regular expressions can be used for the search, with the same special symbols as in other search filters in Dymola. This makes it possible to, for example:
    - Type `start*ward` to highlight all events with `startForward` and `startBackward`.
    - Type `clutch{12}` to highlight all events in `clutch1` and `clutch2` but neither `clutch3`, nor any other components.

Further information about an event is available through right-clicking to display the context menu:



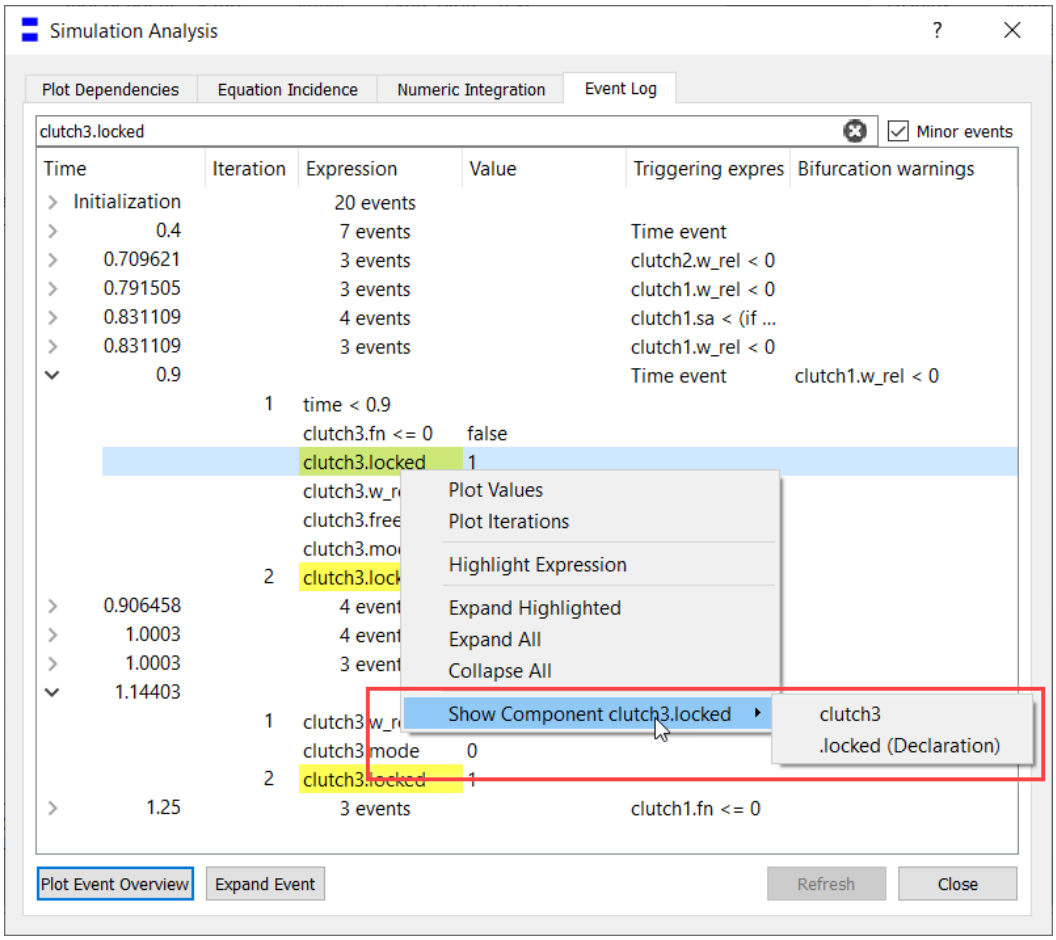
Plotting values and iterations

The context command **Plot Values** and **Plot Iterations** (see figure above) can be used to plot, in the currently active plot window, the values of the event expression, and the iteration number, each time the event expression is changed, respectively. In the figure below, these have been plotted in the event overview plot window.

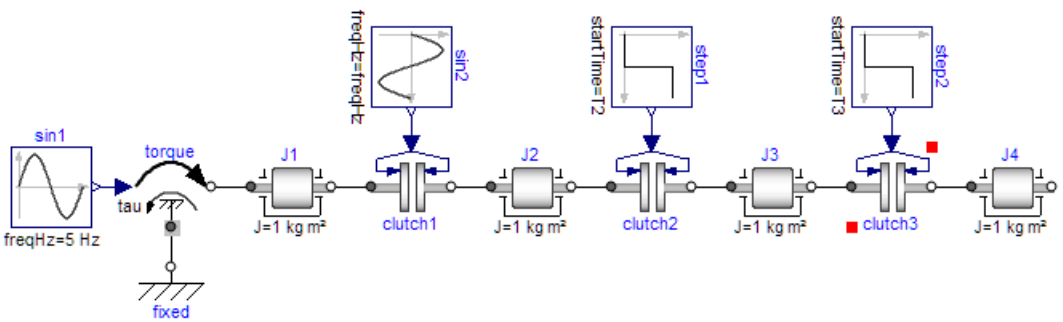


Showing component in the diagram and variable in the code for an event

You can use the context menu for an event to either show the corresponding component highlighted in the diagram, or show the corresponding variable highlighted (search hits) in the Modelica Text layer:



An example of the first subcommand for the event above is:



An example of the second subcommand is:

```
equation
  // Constant auxiliary variable
  mue0 = Modelica.Math.Vectors.interpolate(mue_pos[:,1], mue_pos[:,2], 0, 1);

  // Relative quantities
  w_relfric = w_rel;
  a_relfric = a_rel;

  // Normal force and friction torque for w_rel=0
  fn = fn_max*f_normalized;
  free = fn <= 0;
  tau0 = mue0*cgeo*fn;
  tau0_max = peak*tau0;

  // Friction torque
  tau = if locked then sa*unitTorque else if free then 0 else cgeo*fn*(
    if startForward then
      Modelica.Math.Vectors.interpolate(mue_pos[:,1], mue_pos[:,2], w_rel, 1)
    else if startBackward then
      -Modelica.Math.Vectors.interpolate(mue_pos[:,1], mue_pos[:,2], w_rel, 1)
    else if pre(mode) == Forward then
      Modelica.Math.Vectors.interpolate(mue_pos[:,1], mue_pos[:,2], w_rel, 1)
    else
      -Modelica.Math.Vectors.interpolate(mue_pos[:,1], mue_pos[:,2], -w_rel, 1));
  lossPower = tau*w_relfric;

end Clutch;
```

Notes:

- Any of these commands shift the mode from Simulation to Modeling.
- These commands are also available when right-clicking a component of an event in the simulation log.

Warning messages for possible bifurcations at events

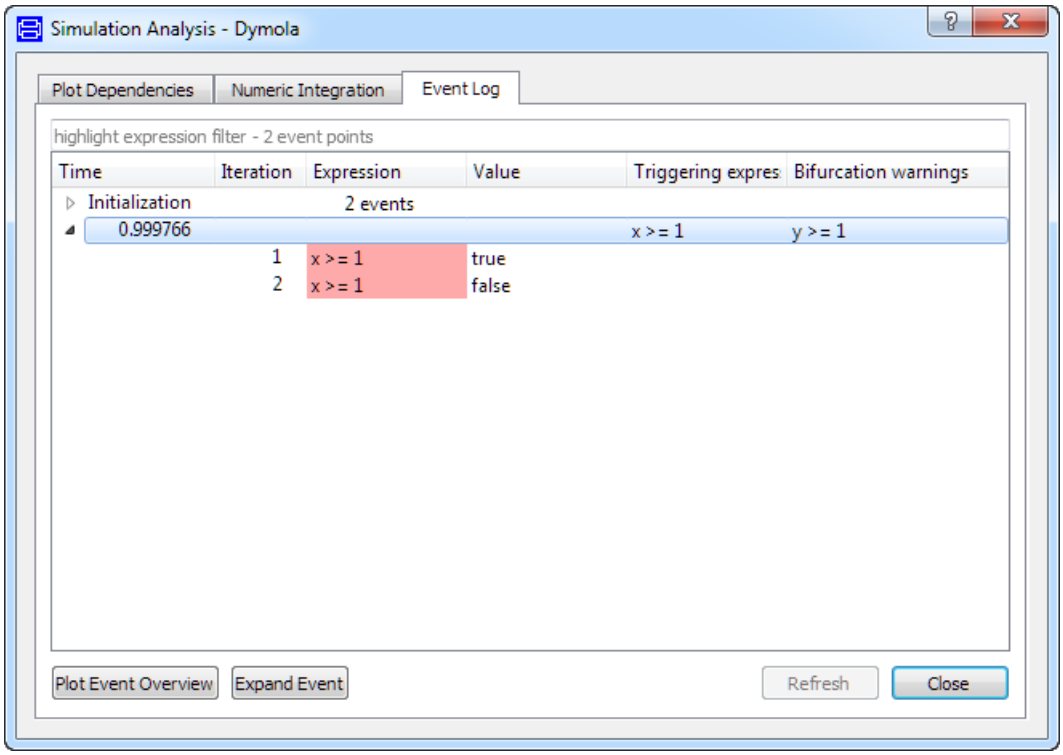
When event logging is used, and an event causes a bifurcation in a solution, a warning is given. Consider, for example, the model:

```
model BifurcationWarning
  Real x(start=0,fixed=true);
  Real y(start=0,fixed=true);
equation
  der(x) = 7*time^6;
  der(y) = 8*time^7;
  when x >= 1 then
    reinit(x, 0);
    reinit(y, 0);
  elsewhen y >= 1 then
    reinit(x, -1);
    reinit(y, -1);
  end when;
end BifurcationWarning;
```

At t = 1 s the numerical approximations of both x and y are close to 1. Depending on the solver the approximation of either of the variables x or y may reach 1 before the other. In that case, the variable that first reaches 1 decides how the model is reinitialized and how the simulation will continue. In such cases, where there are two possible events close to each other, the extended event logging issues a warning. These warnings can be viewed in the **Simulation Analysis** window.

As an example, consider a Dassl simulation of the above model `BifurcationWarning`, using the default simulation setup, with the exception of activating, in the **Debug** tab of the simulation setup, the setting **Events during initialization and simulation**, to be able to study the events.

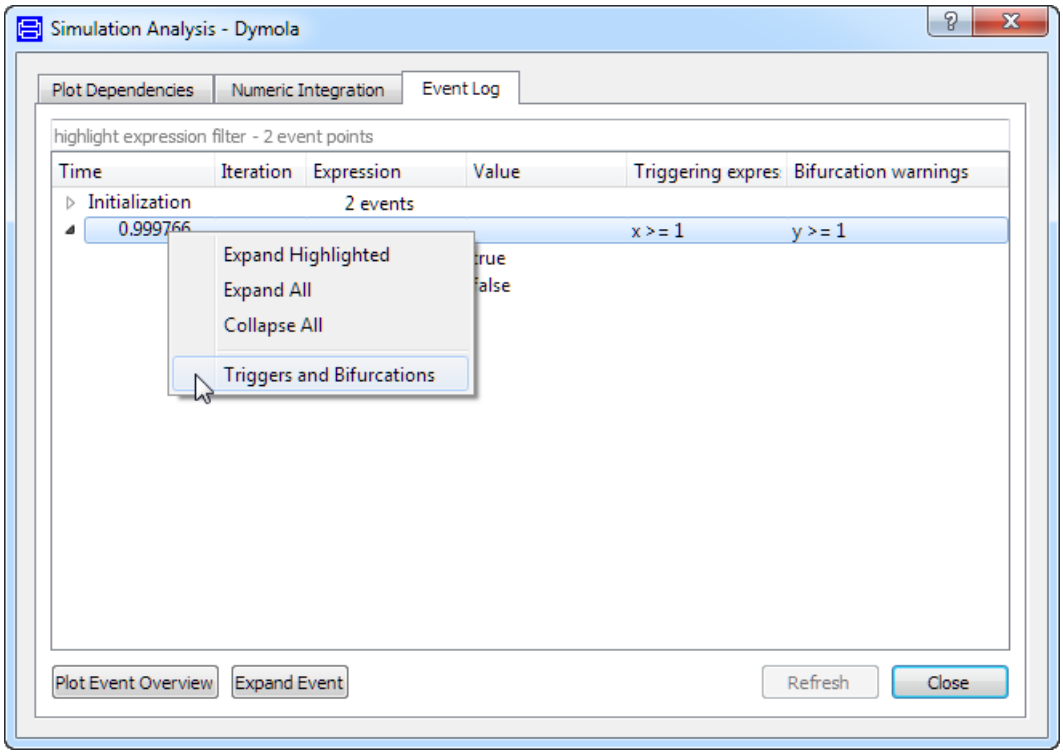
Now simulate the model and then perform the command **Simulation > Simulation Analysis**. This will display:



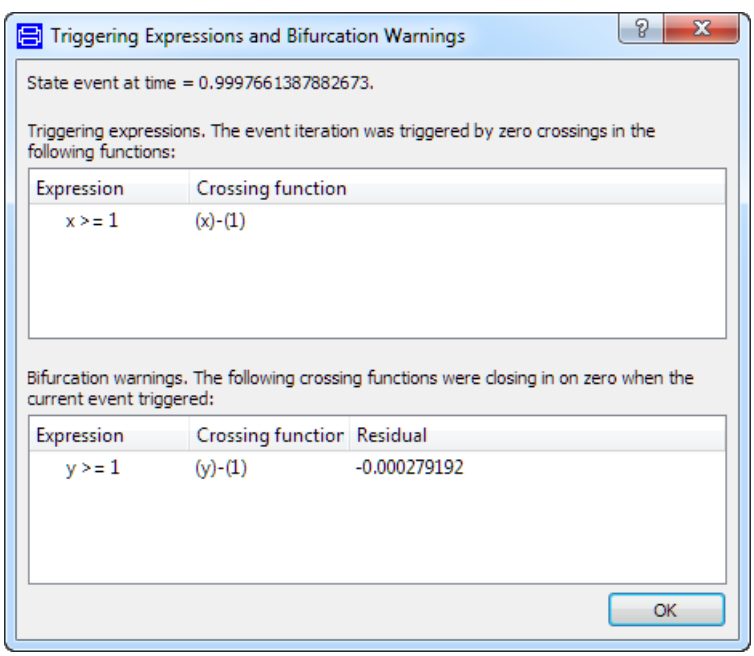
The approximation of x reaches 1 at time = 0.999766 causing an event iteration and a reinitialization of x and y to 0. If y had reached 1 first, then x and y would have been reinitialized to -1 and the result had been significantly different.

In the Simulation Analysis window there is now a bifurcation warning listed. We learn that `x>=1` triggered the event, but the expression `y>=1` was also close to triggering an event.

To view more details, right-click the discussed event and select **Triggers and Bifurcations**:

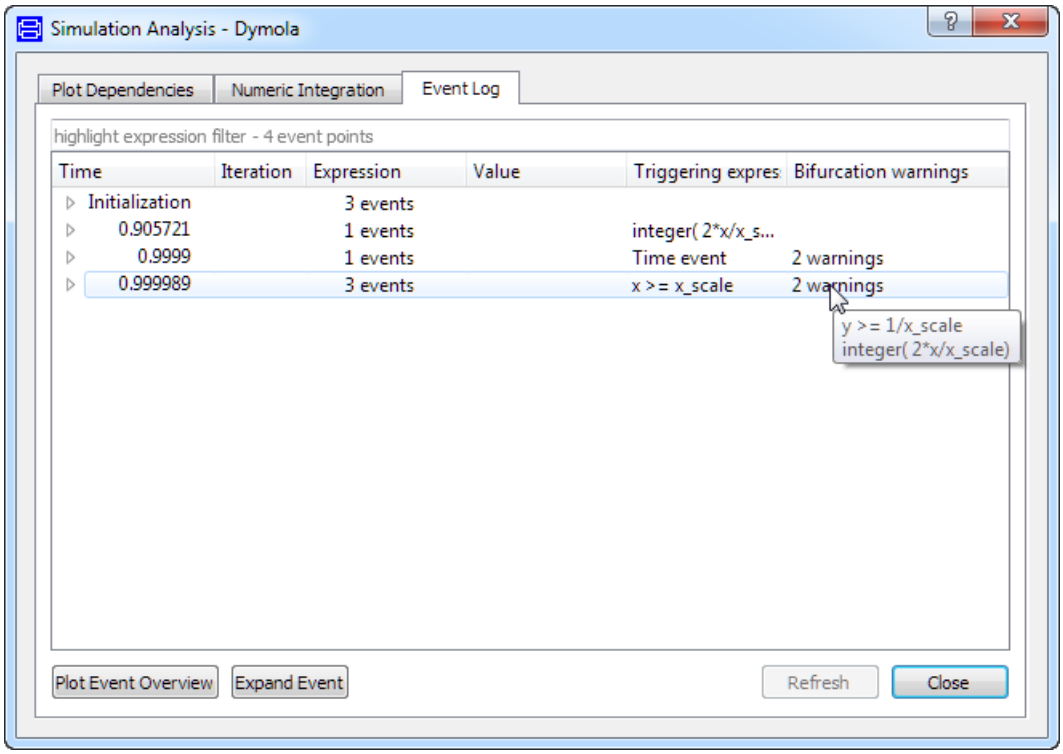


The result is:



Here we can see how close the crossing function (y)-(1) was to zero.

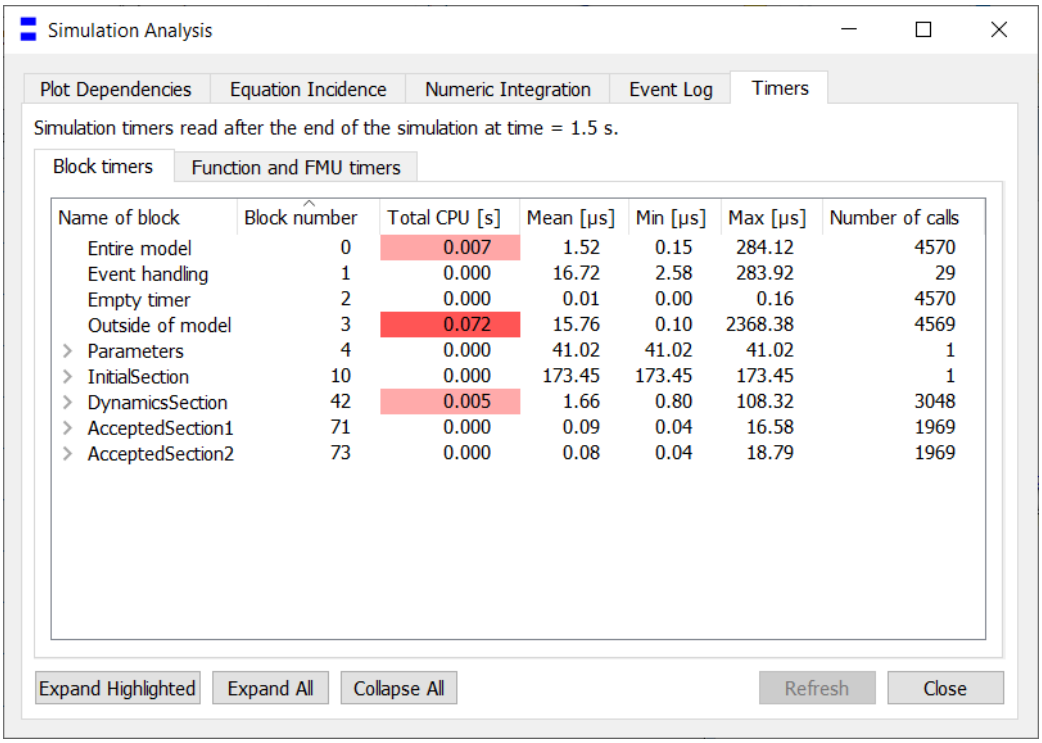
In the case of several warnings (or triggers) for a particular event, the warnings (or triggers) can be viewed by hovering over the line for that event, in the **Bifurcation warnings** column. Like above, all details can be found in the **Triggers and Bifurcations** dialog. An example from another model:



Analysing timers

Presentation of timers

If timers are active, and the flag `Advanced.Debug.WriteTimerResultsToFile` is set to `true`, the timer values are presented in the Simulation Analysis dialog, in the **Timers** tab. The tab has two subtabs, **Block timers**, and **Function and FMU timers**. If you open the demo Coupled Clutches and simulate it (having activated the mentioned flag), the dialog can look like:



The timer values are automatically read from the file timers.txt at the end of the simulation (regardless if simulation succeeded or failed). For long simulations, you can use the **Refresh** button to read the timer values during the simulation.

The most significant contributions to the total time are highlighted in red. The more intense the color, the more time measured by the timer.

By clicking **Expand Highlighted**, all highlighted timers are expanded; this makes it easier to identify the most critical blocks and functions. (You can also **Expand All** or **Collapse All** timers.)

By right-clicking a timer and selecting **Expand All Levels**, the selected item and all its sublevels are expanded.

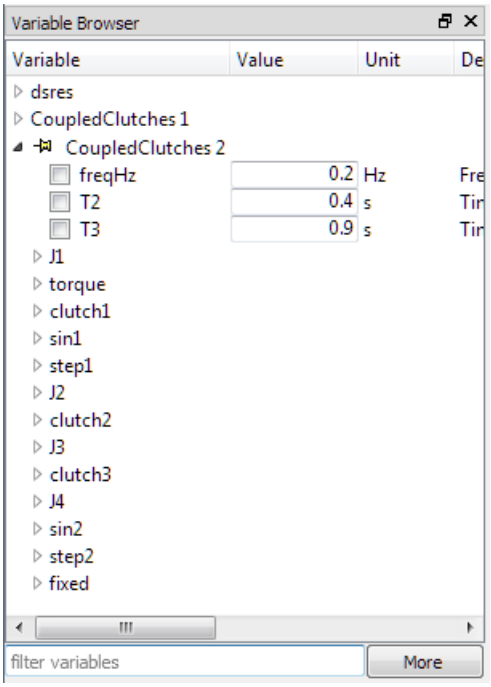
By clicking headers, the data can be sorted according to the values in the selected column.

Analysis of timers

For analyzing of timers, see section “Profiling” starting on page” 1017. Note that the examples in that section can be viewed by the above tab as well.

Filtering of variables

At the bottom of the variable browser an input line for filtering is available. (The text `filter variables` disappear when typing in the field.)



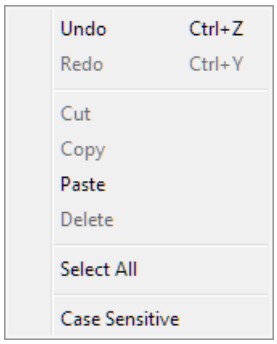
Entering a regular expression limits which variables are shown in the variable browser. Special symbols in the regular expression are

*	Match everything.
?	Match any single character.
{ab}	Match characters a or b.
{a-z}	Match characters a through z.
{^ab}	Match any characters except a and b.
E+	Match one or more occurrence of E.
(ab|cd)	Match ab or cd.
\d	Match any digit.
\w	Match any digit or letter.
^	Match from start.
$	Match at end.

The filtering is by default not case sensitive. To change it to case sensitive, right-click in the line and click **Case Sensitive** (the command works in toggle mode). Note that this change is not saved between sessions; the filtering is always not case sensitive when starting Dymola.

To reset the filter, click the cross in the filter input line (see figure below for example).

A context menu is available for the line:

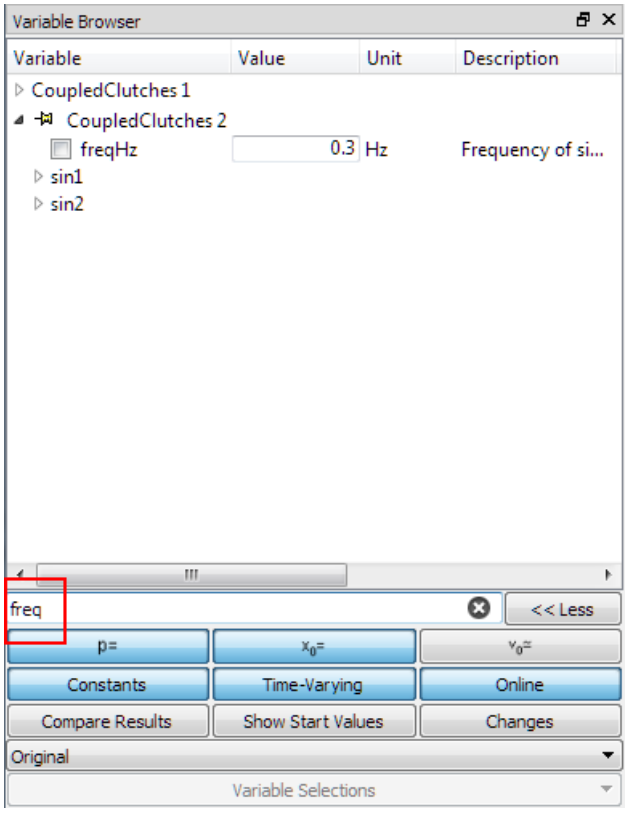


Advanced mode

Pressing the **More >>** button in the bottom of the variable browser (see the variable browser image above) displays buttons for selecting which categories of variables are displayed in the variable browser. It also allows plotting of the difference between signals and to compare the results of multiple simulations.

The below illustrates the advanced mode alternatives, and also an example of filtering.

Advanced variable browser mode.



p= Set parameters of translated model. You must first translate the model.

x_0= Set initial conditions of translated model. You must first translate the model.

v_0= Set initial guess values of translated model. You must first translate the model. These values are intended to help the initialization of non-linear system.

Constants Show constants in the variable browser. This includes all variables that do not vary during the simulation, including parameters, constants, and other variables that are bound to parameter or constant expressions.

Time varying Show all variables that vary during the simulation, excluding parameters, constants, and other variables that are bound to parameter or constant expressions.

Online Load preliminary result during simulation. This enables plotting and animation while a simulation is running. Disabling this gives slightly better performance.

Compare Results Allows you to select variables from the last results file. When selecting a plot variable, it also plots the corresponding variables from the other results. Note that the curves will have different colors, depending from which result file they come from. For more information about this feature, see section "Assign different colors to curves from different simulation result files" on page 721.

Show Start Values Activate synchronization between the animation time and the values displayed in the variable browser. See "Synchronizing values in the variable browser with the animation time" on page 670.

Changes Display changed values between simulations. See "Displaying changed variable values in the variable browser" on pag 672.

Original Clicking this button displays a dropdown with the following alternatives:

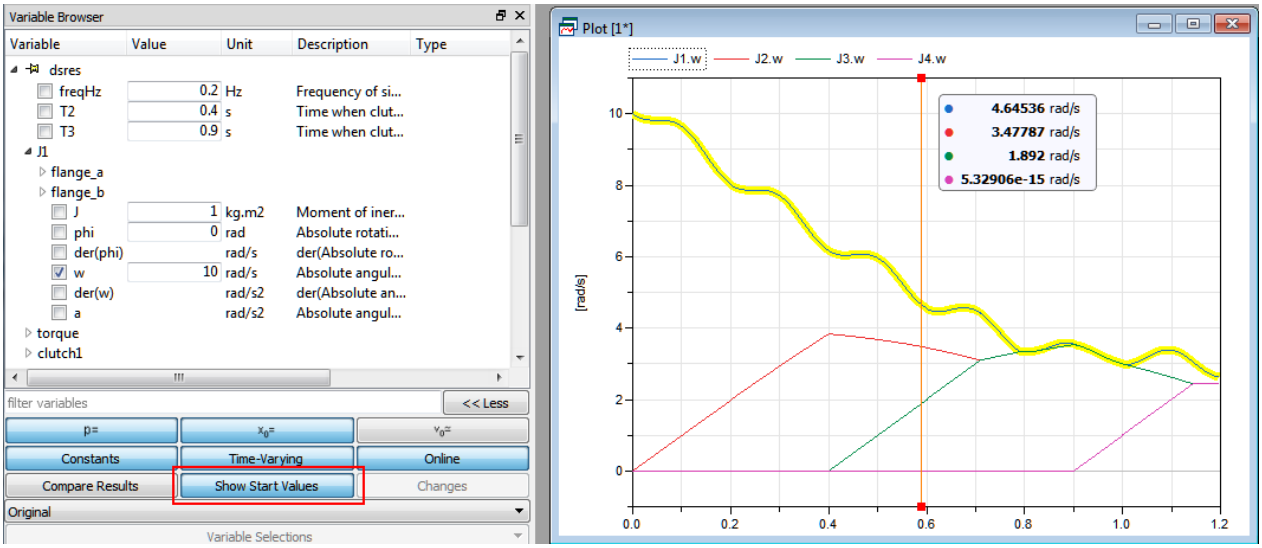
- **Original** Plot the selected signals.
- **Difference** Plot the difference between selected signals. When plotting the difference between two Boolan signals, exclusive OR (XOR) is used to display when the signals differ.
- **Original and Difference** Plot both alternatives above at the same time.

Variable Selections If the model contains variable selections annotation, the user can select what selections should be visible in the variable browser. In this example, the selections **Angles** and **Torques** are present, and part of the currently active simulation results in the variable browser. **MyFilter** is not part of the currently active selection results, which is indicated by **MyFilter** being displayed in italics. The selection **Parameters and States** is predefined and corresponds to the interactive fields for setting parameters and initial conditions of states. For more information about variable selections, see previous chapter, section "Advanced model editing", sub-section "Matching and variable selections".

Synchronizing values in the variable browser with the animation time

The advanced mode command button **Show Start Values** can be used to activate synchronization between the animation time and the values displayed in the variable browser. By default, this option is not activated. This means that the the command button **Show Start Values** is active (blue) and only start values are shown. As an example, running the Coupled

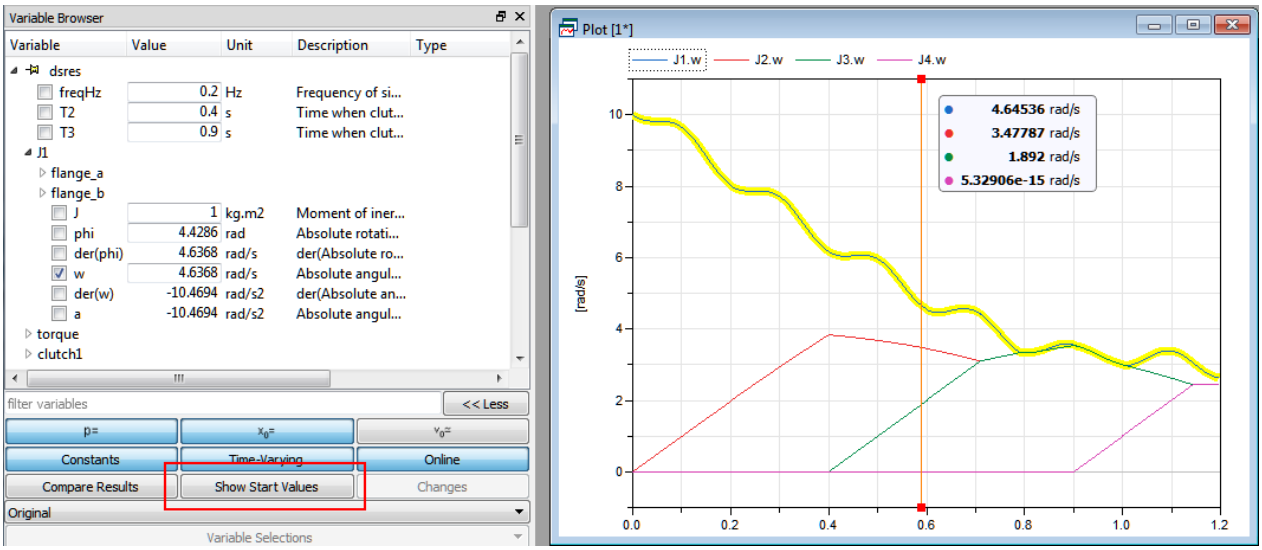
Clutches demo using the command **Commands > Simulate and Plot**, looking at a certain time (expand `J1` to see variables that vary during the simulation):



Clicking the **Show Start Values** button, the mode of synchronizing values is activated. The result is:

- The value of the animation time (set by typing in the **Time** box, moving the time slider, or moving the measurement cursor, if activated) is used to update the values in the value column of the variable browser.
- More values are displayed: all values that are changed by changing the animation time can now be seen in the variable browser.

The result (compare with the image above):



Displaying changed variable values in the variable browser

The advanced mode command button **Changes** displays changed variable values in the variable brower.

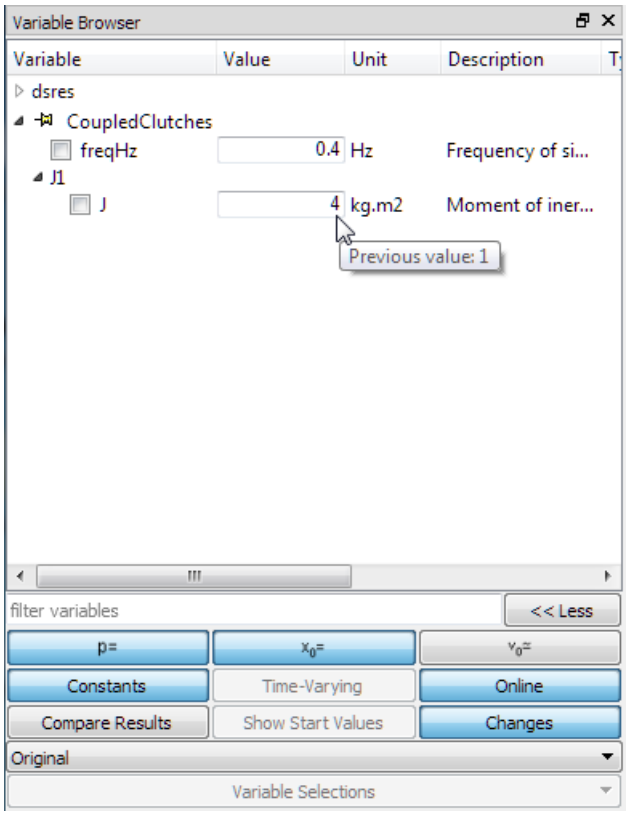
To be able to use this button, you must have at least two simulation results in the variable browser.

Clicking **Changes** will compare the two last result files corresponding to the two last results displayed in the variable browser. Any difference will be shown in the variable browser.

As an example, open the demo Coupled Clutches using the command **File > Demos > Coupled Clutches**, and simulate it with the command **Simulate and Plot**.

Now change `freqHz` from `0.2` to `0.4` and `J1.J` from `1` to `4` in the variable browser, and then simulate again using the command **Simulation > Simulate**. (You cannot use **Simulation > Commands > Simulate and Plot** now since this command does not take into account the changes you made.)

Now clicking **Changes** will display the variable browser:



The changed variables are displayed with present values. To see the previous value for a variable, use the tootltip.

Notes:

- The comparison is between two result files, not compared to the values in the model. For this reason, the result context command **Save Start Values in Model** may not save the same values as is displayed by the **Changes** button.
- Pressing the **Changes** button will have the following impact on other buttons in the advanced mode:
    - **Constants** is activated
    - **Time-Varying** is deactivated and disabled
    - **Show Start Values** is activated and disabled
- The name of the result files does not have to be the same. In the above example, the result file from the command **Simulation > Commands > Simulate and Plot** is `dsres.mat`, while using the **Simulation > Simulate** command on the model creates the result file `CoupledClutches.mat` (due to using different underlying built-in functions for the simulation).

Pressing **Changes** again restores the variable browser to the previous display of result files, and restores the buttons in the advanced mode mentioned.

Relative tic mark labels

When the range is too small to display absolute tic mark labels, diagrams use relative axes instead of absolute axes. This mode is highlighted by bold underlined text for the base number and "+" signs in front of the relative offsets.

Handling of simulation result files in the variable browser

Default handling

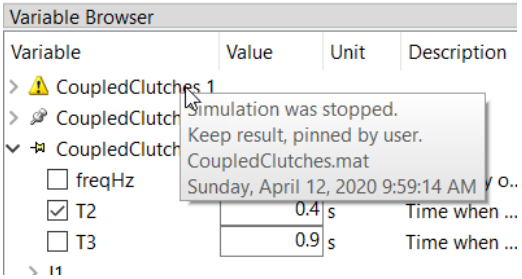
By default, two simulation result files are displayed in the variable browser, the one from the second latest simulation and the one from the latest simulation. This corresponds to the default setting (2) of **Number of results to keep** in the simulation settings, the **Output** tab reached by e.g. the command **Simulation > Setup**. Please see section "Output tab" on page 809 for more information about this tab.

They are numbered with a sequence number, by default absolute. (This is controlled by the setting **Number result files with absolute numbers** in the **Options** tab mentioned above.)

Performing a new simulation, any selection of variables to display in a plot window in the most resent simulation result file is moved to the new result file being shown. This includes the pinning of the result file as being automatically selected to be kept (see below).

The status of the result files (running, failed, and aborted) is indicated in the variable browser.

Failed and aborted are indicated by a warning icon in front of the result file name, while running is indicated by the text (Running) after the result file name. To see if a file has failed or been aborted by the user, the tooltip can be used for the result file. This tooltip also displays the name of the result file (including a timestamp when the file was last modified), and if the result file should be kept.



Opening additional simulation result files

Additional simulation result files can be opened (displayed in the variable browser) by the command **Simulation > Load Result** or by dragging or double-clicking such files. Note that using any of the two commands, several result files can be opened at the same time. Please see section "Simulation > Load Result" on page 829 for more information about this.

Such manually loaded files are not included in the setting **Number of results to keep** above.

Closing result files

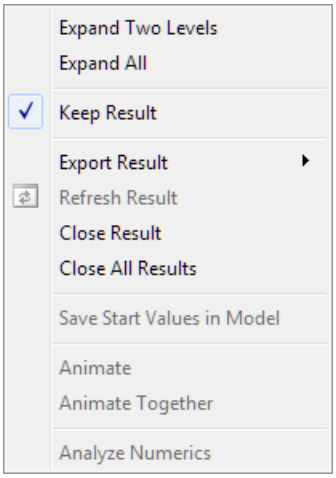
Result files can be closed using the context command **Close Result** in the context menu available for the result files.

Selecting which simulation result files to keep when performing a new simulation

On top of the above, there are two ways to select what displayed result files to keep when performing a new simulation; user-defined by context menu and automatically by keeping result files for which variables have been selected. Please note the difference in behavior for files selected to keep by the user and the ones automatically selected. See also example below.

GUI for selecting which simulation result files to keep (user-defined selection)

In the context menu of the result files in the variable browser the user can select **Keep Result**. If this is selected, the result file will *never* be automatically deleted when a new simulation is performed.





Such result files will be indicated by a specific pin symbol in front of the result file in the variable browser – see also the example below, and compare with the pin symbol for automatic selection of result files to keep.

Note. If the result file has failed or been aborted, only the warning icon is shown, not if the file is kept (which is usually not the case). To see the complete status, use the tooltip for the result file.

Keeping result files for which variables have been selected (automatic selection)

If a variable in a result file is selected for display in a plot window, that file will by default automatically be kept when a new simulation is performed, unless the variable is deselected before the simulation.



Such result files will be indicated by a specific pin symbol in front of the result file in the variable browser – see also the example below, and compare with the pin symbol for user-defined selection of result files to keep.

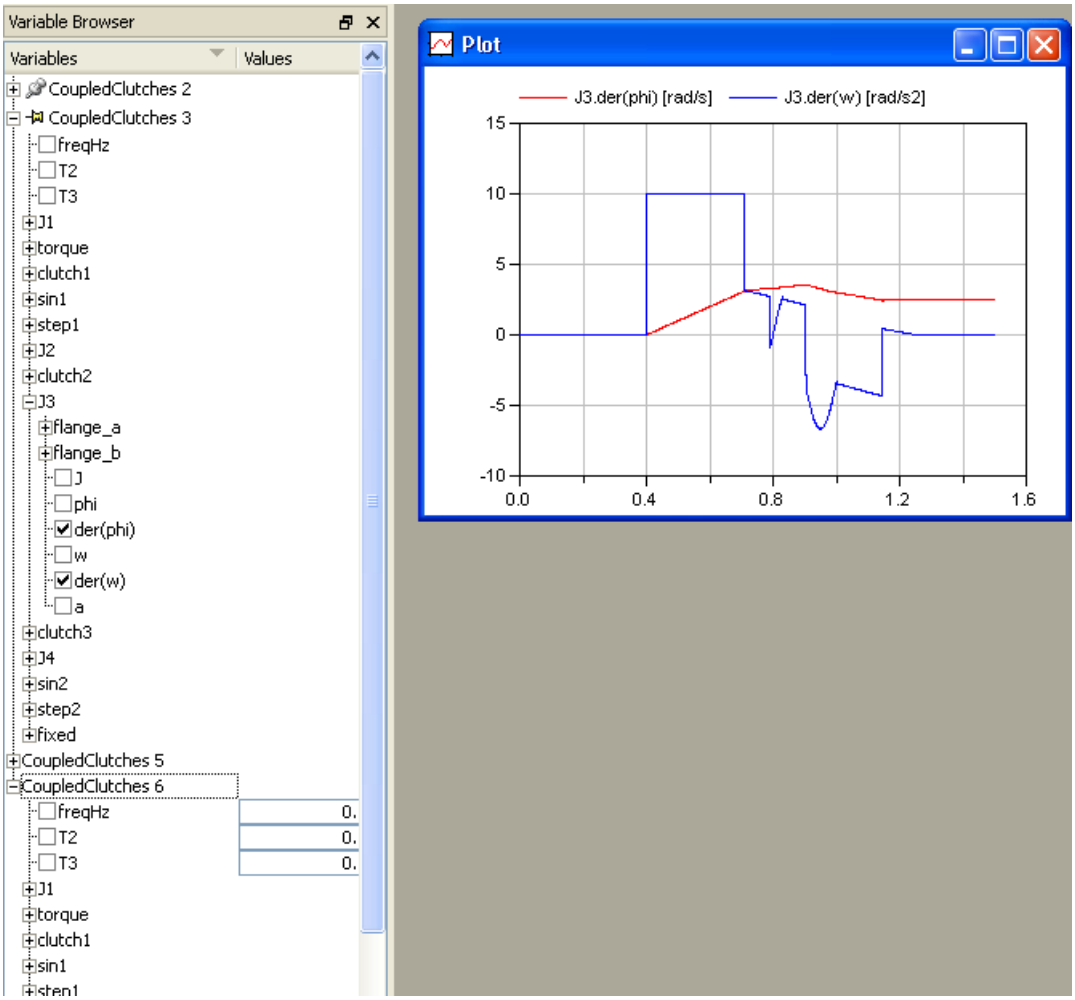
Notes:

- If the result file has failed or been aborted, only the warning icon is shown, not if the file is kept (which is usually not the case). To see the complete status, use the tooltip for the result file.
- In some cases, for example when you work with manual sweeping, you may not want each new simulation result file to be automatically kept. You can control this by the option **Keep result files when signal is plotted** in the plot setup. (This option is by default activated.) For more information about this option, see “Options tab” on page 854.

The relation user defined/automatic keeping of simulation result files

A result file can be both user-defined as to being kept and automatically selected to be kept. Such a file is displayed as user-defined to keep. If the user selects not to keep the file by unticking **Keep Result** in the context menu of that file, the file will be displayed as automatically kept.

Example



Six simulations of the demo Coupled Clutches have been performed.

The default value of number of result files to keep (2) has been left unchanged; the second last simulation (no. 5) and the last simulation (no. 6) are displayed in the variable browser. The default setting of using absolute sequence number on the result files has also been left unchanged.

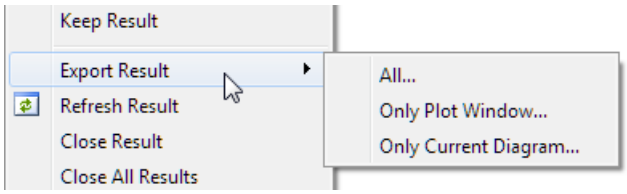
Before the forth simulation was performed, the second result (no. 2) was selected to be kept by the user using the **Keep Result** command in the context menu of that result file. Note the pin on that result file.

Before the fifth simulation was performed, the third result (no. 3) was automatically selected to be kept because of the using displaying variables from it in a plot window. Note the pin of that result file.

If a new simulation will be performed, a new result will be generated and displayed (no. 7); the fifth result (no. 5) will be automatically closed. No. 2, 3 and 6 will still be displayed (unless the user interacts by e.g. deselecting all plotted variables in no.3; then that one will be closed also).

Exporting simulation results

It is possible to export plotted variables in files of different formats (CSV/CSV with unit/in unit, SDF, Matlab and txt) using any of the **Export Result** commands in the context menu of a top node in the variable browser.



It is possible to export all variables in a result file, but also a selection (plotted variables in the whole currently active plot window or in the currently active plot diagram).

Exporting the result to a file `dsu.txt` using the command **Export Result > All...** and using the **Save as type:** selection **Dymosim input File (.txt)**, you can generate an input file for further simulation studies in Dymola.

Please see section “Context menu: Variable browser” on page 869 for more information.

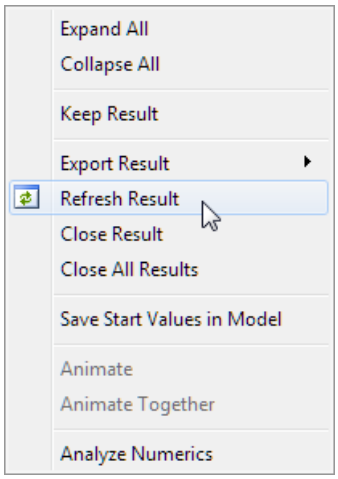
Note that you also can convert a result file to .sdf format (Scientific Data Format) by a pre-defined post-processing command. See “Example: Support for Scientific Data Format” on page 790.

Refreshing simulation result files

You can refresh simulation result files. This may be useful when you, for example, work with other applications than Dymola that may change the simulation result. To check the outcome of such interaction, you may want to open the changed simulation result file in Dymola without having to redo the plot setup.



To refresh a simulation result, right-click the corresponding top node and select **Refresh Result**.



The command reloads the latest corresponding simulation result file without any changes in plot setup etc.

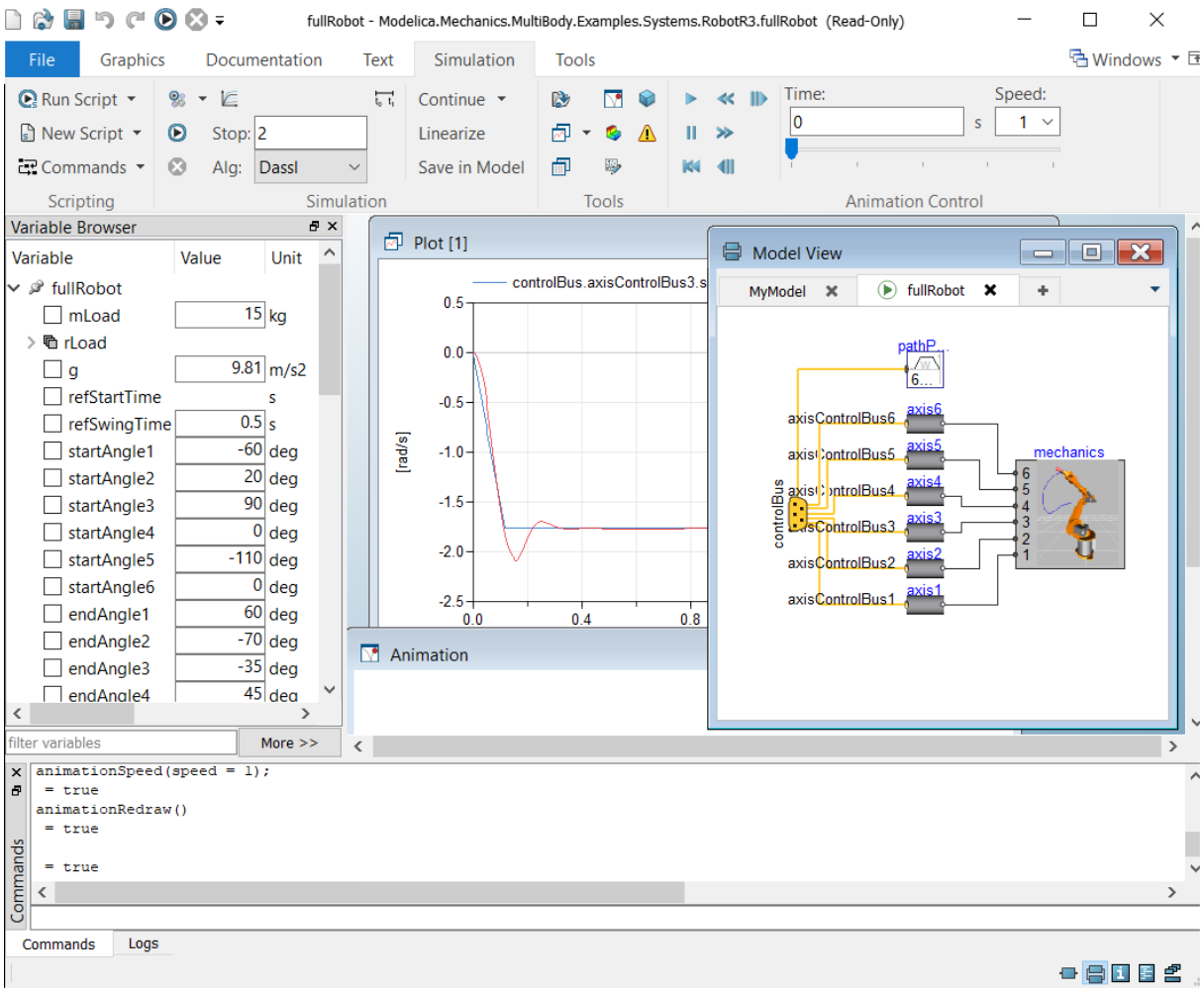
Notes:

- You can only refresh the latest result of simulating a certain model. For older simulation result files, the command is dimmed.
- Plots are automatically replotted based on the new signal data.

5.2.3 Using the Model View window

General

The Model View window is by default present in the **Simulation** tab, and does by default display the diagram layer of the model to be simulated.



Any model layer (icon, diagram, documentation, Modelica text, and used classes, can be displayed in the Model View window, by clicking the corresponding buttons in the toolbar in the lower right corner of the Dymola window (see above figure, where the diagram icon is highlighted since the diagram layer is displayed in the Model View window). The main use is working with the diagram layer.

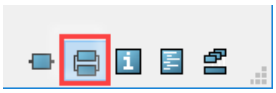
Note that model tabs are available in the Model View window, in the above example you can switch to MyModel by clicking that model tab.

Working with the diagram layer in the Model View window

The diagram layer representing the model enable the user to follow a simulation by displaying variables and to control it by setting parameters. The user can descend into any level of the model in order to plot or display variables. Filtering by dimming components in the diagram not matching domains/component names can also be applied.

Displaying the diagram layer

To display the diagram layer in the Model View window, if not displayed already, click the diagram button in the toolbar to the lower right of the Dymola window:



This also opens the Model View window, if it was closed before.

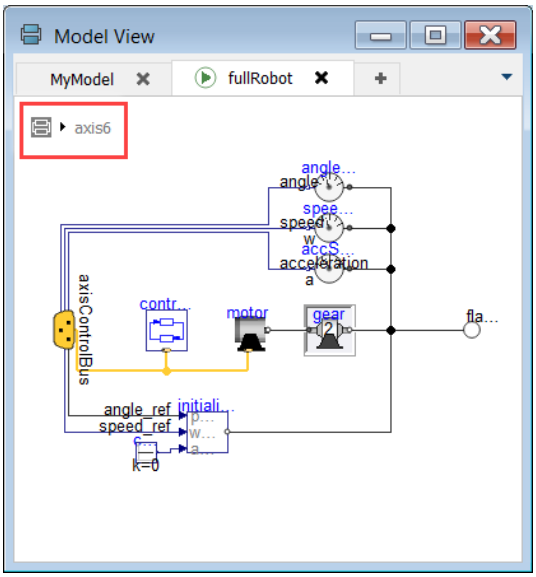
Navigating in the diagram layer

Navigating in component hierarchy.

You can navigate in the component hierarchy of the diagram.

To go deeper in the component hierarchy, use the context menu item **Show Component**; that is, select the component you want to investigate, right-click, and select **Show Component**. This will display the content of the component, and the name of the component will be added to the instance hierarchy path in the upper part of the Model View window.

As an example, having the Robot demo in the **Simulation** tab, right-clicking **axis6** and selecting **Show Component** will result in:



To go back, you can click any component in the instance hierarchy path (framed in the figure above) to go to that component. In the example above, clicking the Dymola icon will display the top level of the Robot demo.

Moving in the diagram.

To move the view, press, in empty space in the Model View window, hold the **Ctrl** key as well as the left mouse button, and then move the mouse. The diagram will be dragged along with your mouse cursor. To stop, simply let go of the left mouse button.

As an alternative, you can use the mouse wheel to move the diagram view in the y and x direction the following way:

- **Mouse wheel** moves the diagam view in the y direction only
- **Shift+Mouse wheel** moves the diagram view in the x direction only

Zooming in the diagram.

To zoom, there are four different options:

- Press and hold the **Alt** key and span a rectangle[9]. When the mouse button is released, the spanned area is zoomed to fit the window.
- Mouse wheel: press and hold the **Ctrl** key and scroll the mouse wheel.
- Mouse move: press and hold the **Ctrl** key and the right mouse button, then move the mouse forwards or backwards.
- Change the zoom factor by editing the zoom factor value in the toolbar.



A convenient way of displaying the whole model indepently if some objects are located ouside the coordinate system is to right-click an empty area, having nothing selected, and select **Fit to Window** from the context menu. (If components in the model are located outside the coordinate system, the resulting zoom factor will differ from 100 %.)

Show variables for a component or a connection in the variable browser from the diagram layer

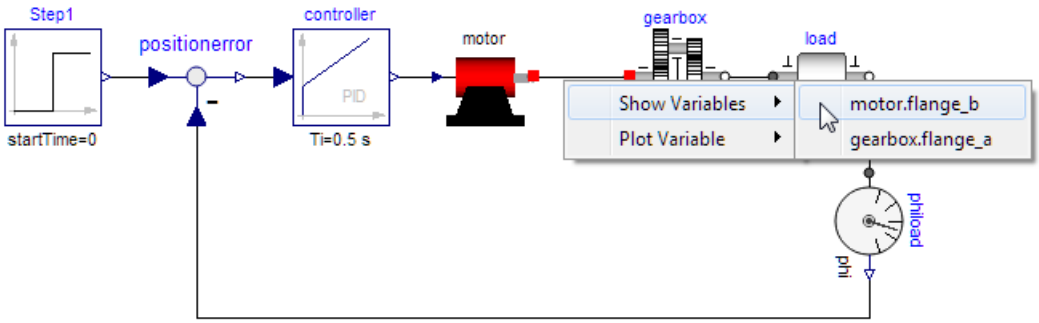
You can this in two ways, using a context menu command, or clicking directly on the component/connection.

Using a context menu command to show varibles.

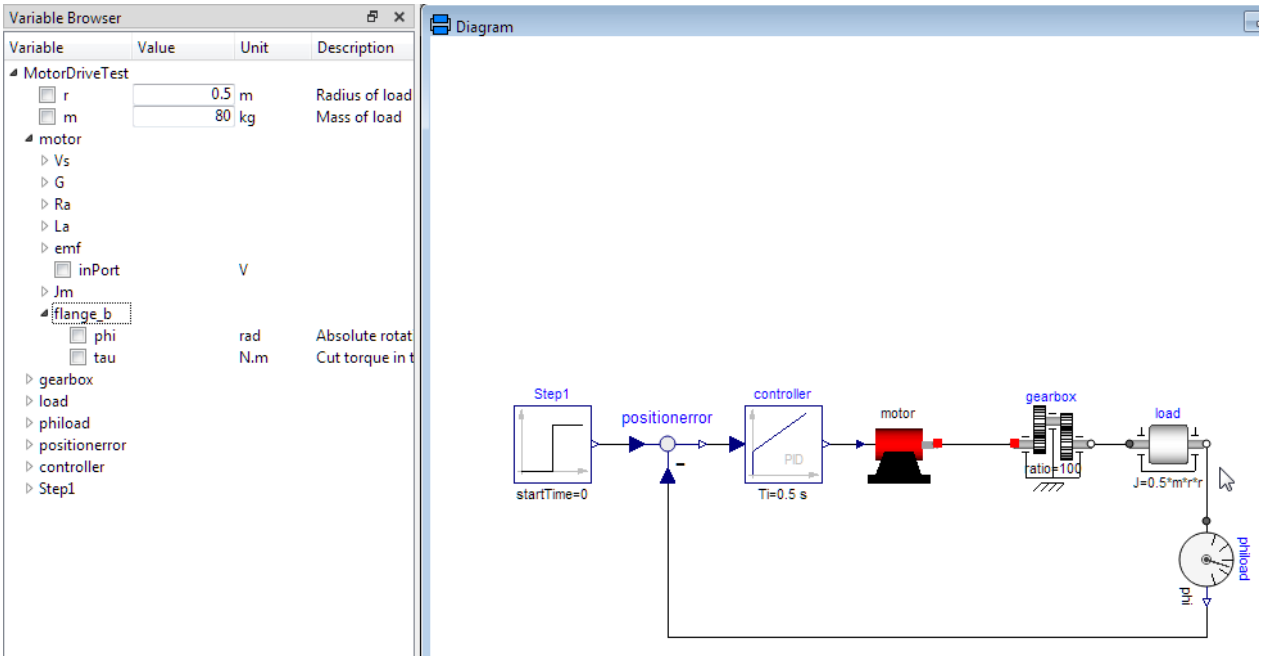
Right-clicking a component in the diagram layer and selecting **Show Variables** will highlight and open the selected component instance in the variable browser.

Right-clicking a connection line, you can select to show the variables of the first part of the connection or the second part of the connection in the variable browser. As an example, open the demo Motor Drive and simulate it. Display the diagram layer. Right-clicking the connection between the motor and the gearbox gives:

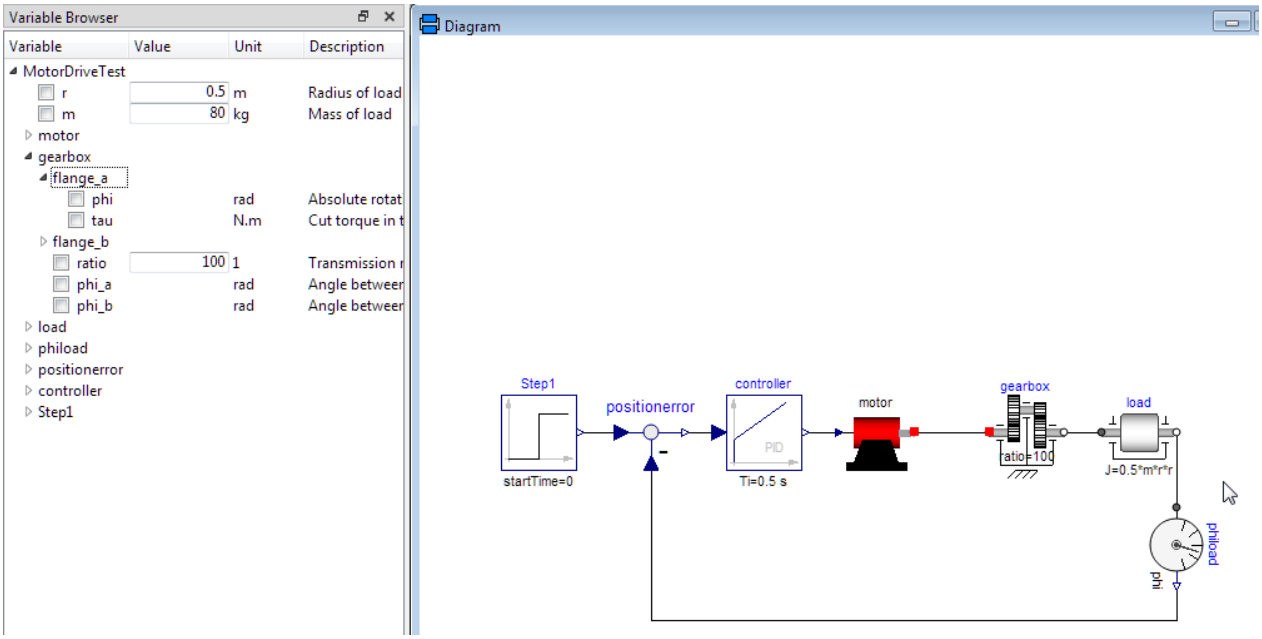
[9] On Linux, **Ctrl+Shift** have to be used instead of **Alt**.



Selecting the first part of the connection, as indicated by the cursor in the above image, will open the corresponding node in the variable browser:



If you instead would have selected to show the second part of the connection, the result would have been:



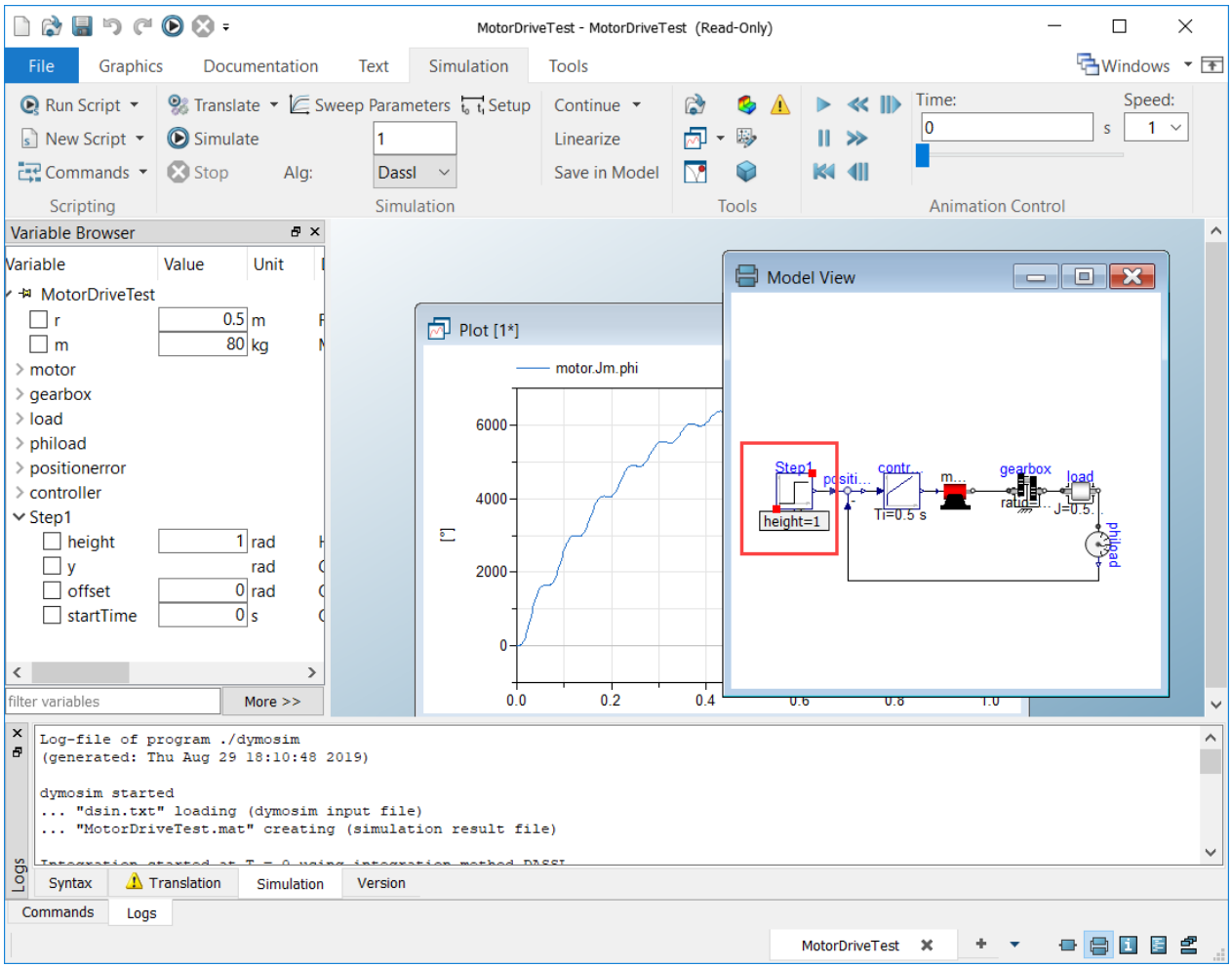
Please note that if the diagram layer window is active, selection of another component in the variable browser will also change the selection in the diagram layer window.

Click to show variables.

You do actually not need to right-click a component or connection and select **Show Variables** to navigate to and open the corresponding node in the package brower, as described in previous section. It is sufficient to just click on the component or connection. Please note however that for the connection, you can only navigate to the first part of the connection this way, to navigate to the second part, you have to use the context menu as described in above section.

Presenting values in the diagram layer

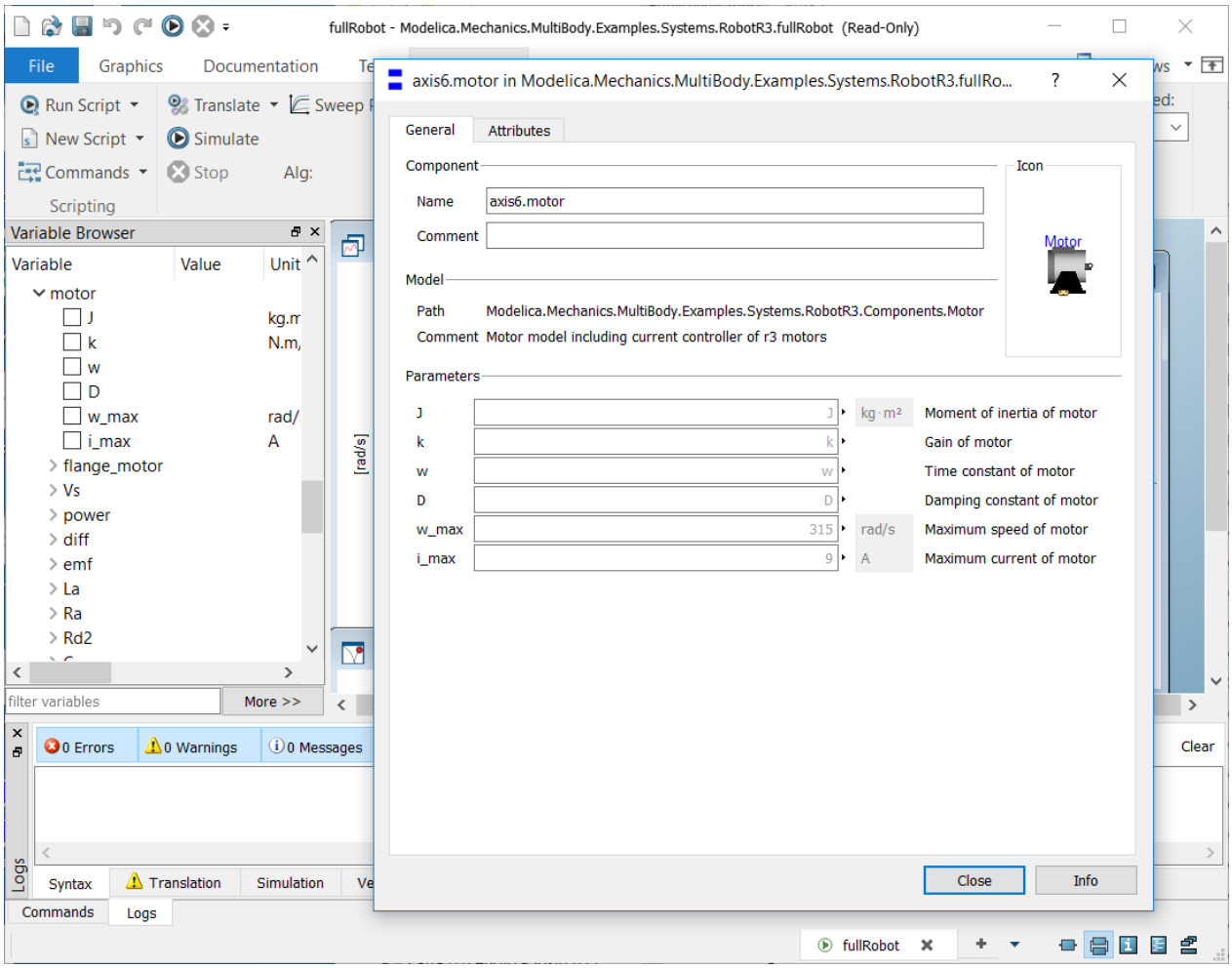
When the diagram layer has been used for selecting variables (please see previous section), it is very convenient to show values in the diagram layer also. The context menu in the variable browser for a variable has one entry **Show in Diagram** that will present the name and value of that signal below the relevant component in the diagram layer. In the example below the step height is presented (framed for clarity).



The context menu entry **Show in Diagram with Legend…** for a signal in the variable browser makes it possible to change the representation of the value display. For more information, please see section “Context menu: Variable browser – signals” on page 873.

Changing parameter values using the diagram layer

By double-clicking on a component or selecting **Parameters** in the context menu for the component in the diagram layer, the parameter dialog is displayed. (The action when double-clicking on a component can be configured, if double-clicking does not work, **Shift+double-clicking** will work.) While a simulation is running, setting a parameter will directly update the running simulation with a new parameter value. (Parameters can also be set in the variable browser.)



Changing parameters in the parameter window (or variable browser) does not always require a new translation (it is still necessary if the modifier is too complex, the parameter had been evaluated during translation, or if parameters of a different model were changed). The model is changed and you will be asked to save it when exiting Dymola. To use this for top-level parameter see previous chapter, section “Advanced model editing”, sub-section “Components and connectors” sub-sub-section “Working with replaceable components”.

Interactive parameter tuning.

Dymola supports interactive tuning of parameters during simulation runs. The functionality is enabled by setting the flag

```
Advanced.OnlineParameterChanges = true;
```

Setting this flag allows synchronized online parameter changes both from the variable browser and and from the parameter dialog of the diagram view when simulating. (The flag is by default false.)

Important: The parameter changes are not stored in the result file.

Note. A tunable parameter is an interactive parameter that influences simulation parameters (parameters that appear in simulation equations), but that does not influence parameters which either specify fixed=false or that appear in equations differentiated by Pantelides' algorithm.

Finding components, connections, and connected components in the diagram layer

Note that the following commands are available also in the diagram layer when working with simulations:

- **Edit > Find** to find components.
- **Edit > Find Connection** to find connections, and connected components.

For a descripting of these commands, please see previous chapter.

Filtering objects in the diagram layer

To be able to get an overview where components come from, you can filter the diagram layer to dim components from selected domains and/or components not matching a name. The feature is activated by right-clicking in the diagram, with or without a component selected, and selecting **Diagram Filter**. The feature can also be activated by the command **Edit > Diagram Filter**.

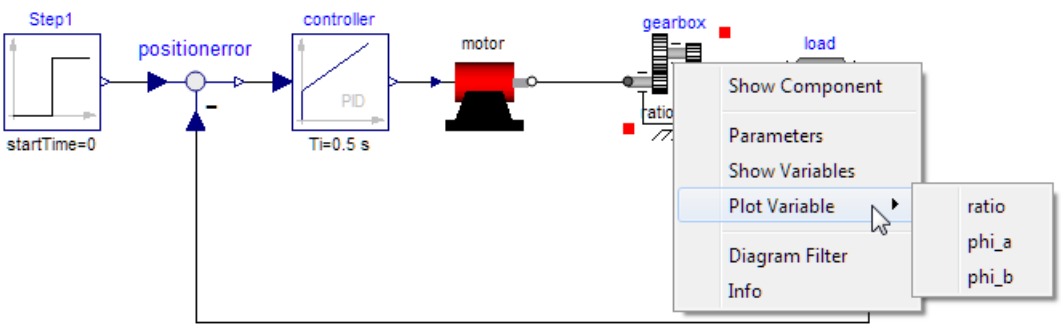
The diagram filter browser appears either as a tab to the variable browser or as a docked browser.

For more information about using this feature, see the page referred in the index entry "*diagram layer filtering*".

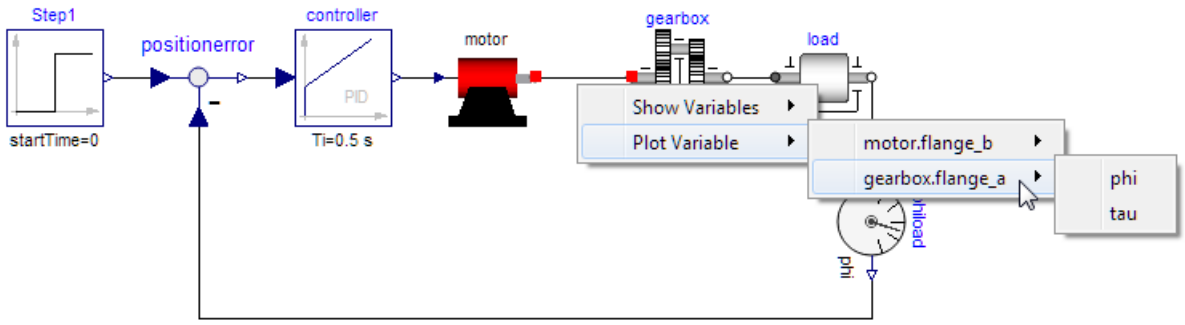
Selecting variables to plot from the diagram layer

You can use the diagram layer when working with simulations to select what variables to plot, the command Plot Variable is available in the context menu for both components and connection lines (and top-level variables).

For components, an example from the Motor Drive demo:

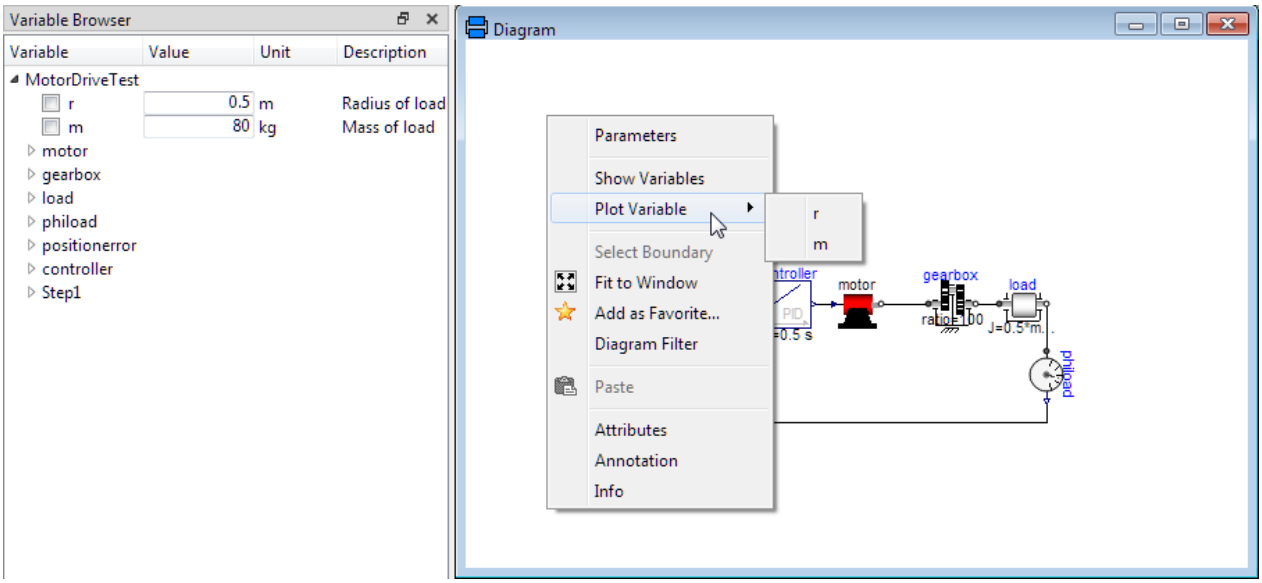


For connections, an example from the same demo:



Notes:

- For connections, you can select to plot variables for both parts of the connection.
- You can plot top-level variables by right-clicking on the background of the diagram layer:



- The command **Plot Variable** is disabled if the corresponding variables do not exist in the result, for example, if the current model is not yet simulated.

Component context menu in the diagram layer

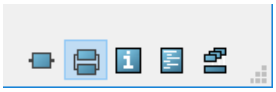
A context menu is available for components and connection lines in the diagram layer when simulating. Commands only available when editing the model are removed instead of dimmed (compare previous chapter). For graphical objects no context menu is available.

For examples of context menus for components are connections lines, see the previous sections. For description of the commands not treated in those sections, see previous chapter.

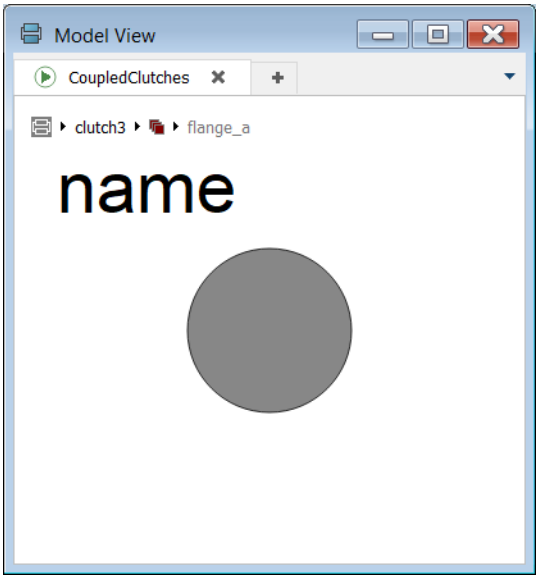
Displaying different model layers in the Model View window

Except presenting and working with the diagram layer in the Model View window, as decribed above, you can also present the other model layers: icon layer, documentation layer, Modelica text layer, and used classes layer.

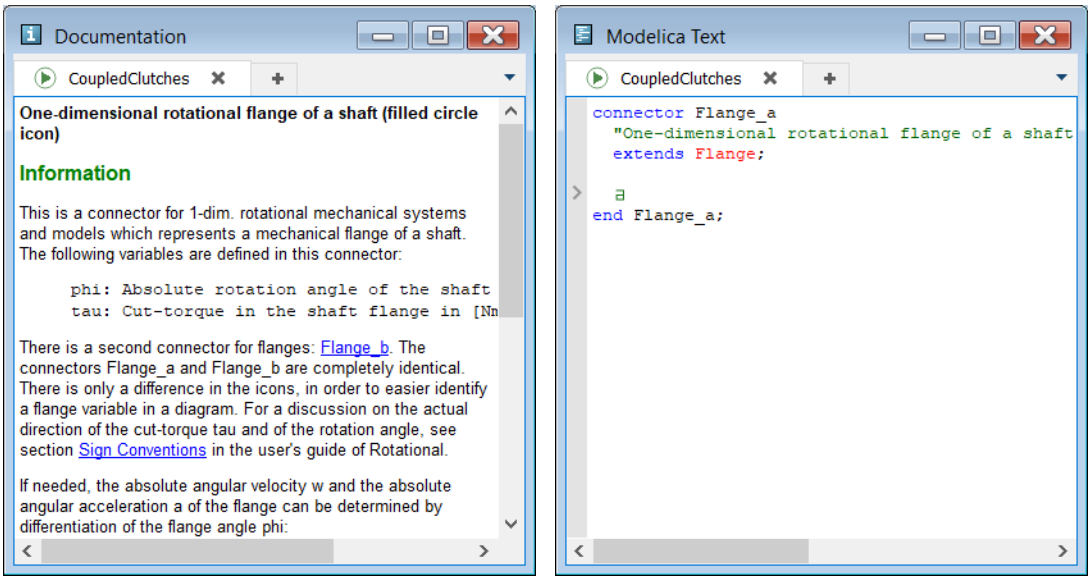
To present any of these layers, click on the corresponding button in the toolbar in the lower right corner of the Dymola window:



As an example, consider the Model View window with the following diagram presented (a flange in clutch 3 in the coupled clutches demo):



Clicking the buttons for documentation layer and Modelica tex layer in the toolbar mentioned above will give:



Notes:

- The layers displayed are always read-only. To edit, you have to go to another tab.
- When you go to another tab, the selection above is remembered; when you open, for example, the **Text** tab, it will display the code of the same component as you had in the Model View window.
- The last selections in previous tabs are also remembered, that is, when you, for example, select to show Modelica text in the Model View window, the text will be displayed with or without mathematical notation, depending on what selection you had in the **Text** tab before going to the **Simulation** tab.
- The shortcuts to model layers (**Ctrl+F1** etc.) do not change the content of the Model View window, they always change the main window tab. For more information about these shortcuts, see the index entry "*shortcuts : to model (class) layers*" in this manual.

5.2.4 Plotting

This section serves as an overview where to find information concerning different aspects of plotting.

Basics

Basics about plotting can be found in the section "Plot results" on page 617 and in the section "Plot window and Plot tab" on page 591.

Selecting signals

The variable browser can be used for selecting what signals to plot. Please see section "Variable browser interaction" starting on page 619 for options here. Please note the use of the **More** mode to be able to plot differences between selected signals etc.

Signals can also be selected for plotting dependencies. See section "Plot dependencies" starting on page 633.

You can select a result file and analyze the numeric integration, including displaying time-plot for selected variables, and information about limits step size, dominates error, and more than 10 % of error. The behavior of the integrator can also be studied by displaying internal integrator step size and order. See section "Analyzing numeric integration" starting on page 649.

Plotting from large result files

For performance reasons, some plot features (e.g. tooltip and curve highlighting) are disabled if a curve has more points than a certain value. This value is specified by the flag `Advanced.Plot.PerformanceLimit`. The default value is 100 000. If more points are to be plotted, without loosing any plot features, this value can be changed. Note that this will increase the time for plotting. If you change this value, you need not to translate again, but you must close and open the corresponding result file.

Plot window interaction

This is described in the next section "Plot window interaction" starting on page 692. For an overview, please see the first sub-section of that section.

Saving a png image of the active plot window

The command **Tools > [Export] Image** can be used to save a png or svg image of the active plot window. Please see the index entry "*tools tab : [export] image*" in the index of this book, to find more information about this command. Scripting can also be used to save a plot, please see below.

Exporting results

It is possible to export plotted variables in files of different formats (CSV, Scientific Data Format (SDF), Matlab and text format). Please see section "Context menu: Variable browser – nodes" on page 869 for more information.

Storing global plot settings between sessions

Global plot settings, as well as some individual plot settings, are automatically stored between sessions in an XML settings file. For more information about this file, please see the description of the command **Tools > Options**, the **Settings** tab, in previous chapter.

Saving the settings of plot windows

The settings of plot windows, including the displayed variables, can be saved by two different commands.

- You can use the command **Simulate > Commands > Add Command**, ticking **Plot window setup** to save the plot windows settings and create a command in the model that retrieves this setting. You can also tick **Automatically run after simulating** to have the command executed after simulation of the model. See section "Simulation > Commands > Add Command" starting on page 793 for more information.
- Alternatively, you can use the command **Script Editor > Generate Script**, ticking **Plot setup**. Please see section "Script Editor > Generate Script" on page 833.

There are some differences between these commands, if you have specific requirements, please compare them in detail.

Plot scripting

A number of built-in functions are available for scripting. For an overview of the functions, please see section "Plot" on page 954 and "Trajectories" on page 971.

Support for predefined plots

Dymola supports Annotations for Predefined Plots, which are proposed as an extension of the Modelica Specification (MCP-0033), but the following should be noted:

- Figure captions are placed in Dymola´s Plot Documentation (the "?" button in the plot window title bar)
- Empty legends are not supported
- Variable replacements and links in text are not supported

Plots in documentation

Plots can be exported as .png images using scripting (see above). They can also be inserted in the command by the user. Please see section "Insert plot and its corresponding command in command window" on page 753. They can also be inserted into the command log automatically. Please see section "Plots automatically inserted in command log" on page 774 for more information. (This can also be handled by scripting.)

5.2.5 Plot window interaction

Overview

The user can interact in the plot window and the plot tab in a number of ways:

- Display several diagrams and work with them in the same plot window, in one column or as 2D layout.
- Display Boolean and enumeration signals, separated or overlapping.
- Display curves negated.

- Duplicate diagram to new plot window.
- Display dynamic tooltips (and copy the values to clipboard if needed).
- Display tooltip for the legend to see more information about the corresponding signal.
- Display tooltip for the x-axis and the y-axis.
- Move and zoom in the window.
- Apply time sliding window.
- Change time unit on x-axis.
- Change the display unit of a signal.
- Using a measurement cursor.
- Drag and drop curves between plots/diagrams.
- Drag and drop of curves between plots and tables.
- Lock plots and tables.
- Add a y-axis to the right (secondary axis).
- Assign different colors to curves from different simulation result files
- Display signal operators.
- Plot general expressions.
- Plot 1D discretizations.
- Plot dependencies.
- Select multiple curves in a diagram (for e.g. copying the curve values to Excel).
- Display a table instead of curves.
- Display constant arrays as matrices.
- Create tables from curves.
- Apply color-coding to simulation result file tables.
- Plot parametric curves.
- Create scatter plots.
- Change color, line/marker style and thickness of a signal.
- Plotting bar charts and area charts.
- Plotting of external files.
- Adding titles and documentation to plot windows.
- Set diagram heading and axis titles and ranges.
- Insert and edit text objects in the plot.

- Change the appearance, layout and location of the legend. (The legend can be located in several different places, for example inside the diagram.)
- Display the component where the signal comes from in a diagram layer.
- Go back to a previously displayed plot window.
- Insert plot and its corresponding command in the command window.

These items will be described below, but to simplify further descriptions, the plot setup menu is described as first item.

Note that several plot windows can be displayed at the same time. The command **Plot: Options > New Plot** will display a new plot window.

Plot setup menu

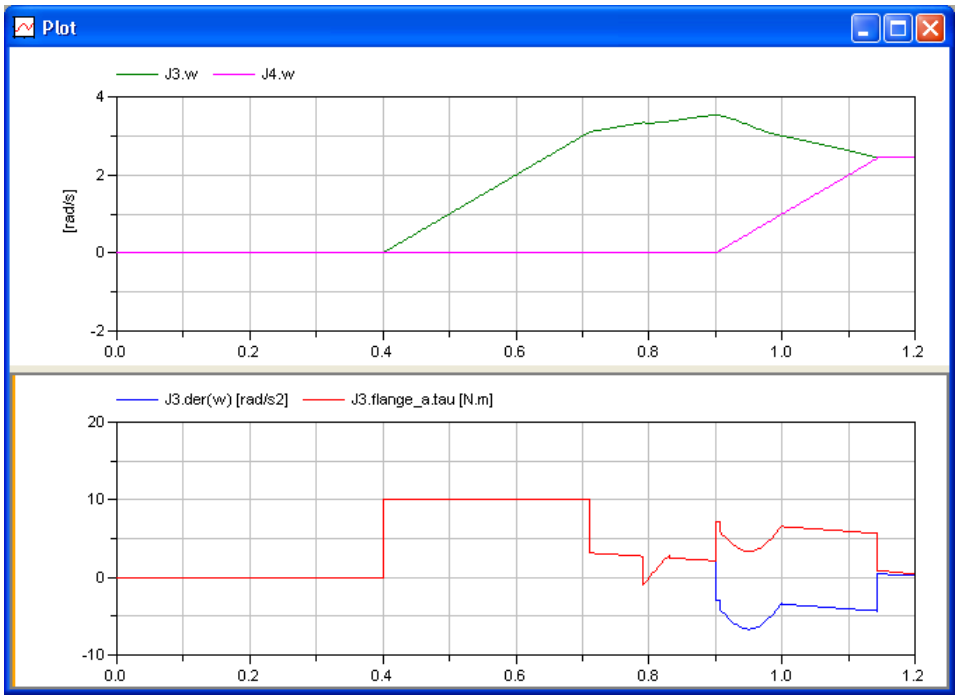


In a number of interactions below, the plot setup menu is used. It can be reached by the command **Plot: Options > Setup**. It is also available as **Setup…** from some plot context menus. We will below refer to it as the "plot setup menu". (More information about it can be found in the section "Plot: Options > Setup" starting on pge 842.)

Several diagrams in a plot window (one column)

By default, a new plot window contains only one diagram. It is easy to add more diagrams and work with them. In this section, we limit ourself to one column of diagram, but you can also apply 2D layout – see next section.

One diagram is always active; this is indicated by an enclosing grey rectangle, and a tiny orange bar to the left in the active diagram. In the figure below, the lower diagram is the active one.



A number of commands/buttons facilitate working with several diagrams. Several of them are available in the context menus, as well as plot tab commands.

New Diagram will create a new empty diagram in the active plot window. For more information, see “Plot: Options > New Diagram” on page 840.

Reset [Window] Layout will scale the diagrams in the active plot window to distribute them evenly in the plot window. **Note** that this command is only available as **Plot: Options > Reset Layout**.

Rescale All will rescale all diagrams (reset them to initial zooming) in the active plot window.

Rescale Diagram will reset the zooming to the initial one for all signals in the active diagram.

Erase Content will erase all curves (and text objects, if any) in the active diagram.

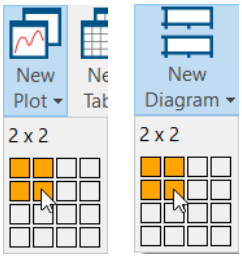
Delete Diagram will delete the active diagram.

You can move the active diagram one step up in the diagram order by **Ctrl+Up**, one step down by **Ctrl+Down**.

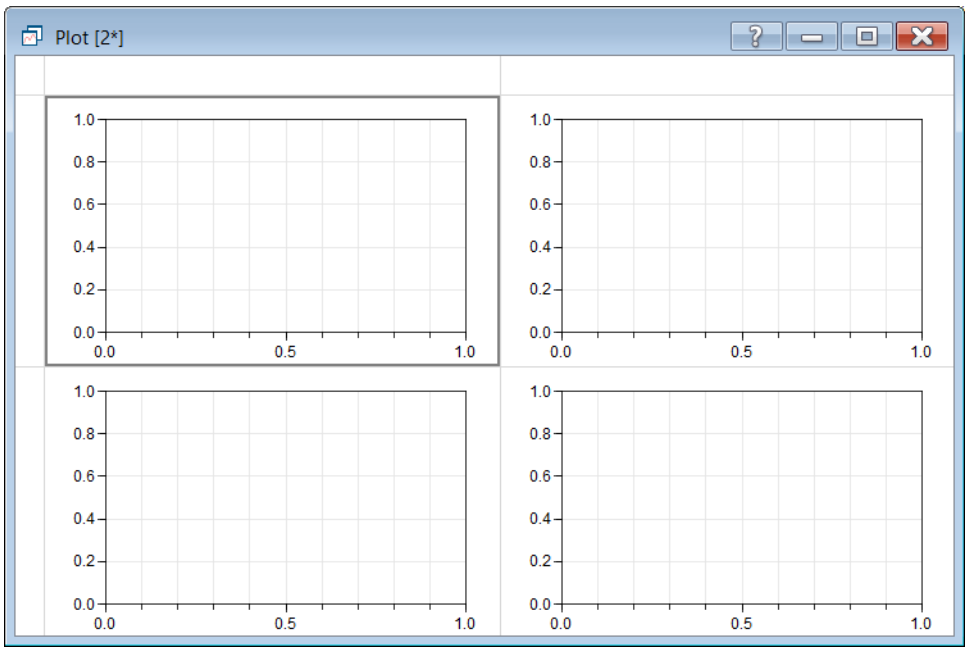
Several diagrams in a plot window (2D layout)

To work with 2D layout of diagrams in a plot window, the subtab **Layout** is available in the **Plot** tab:

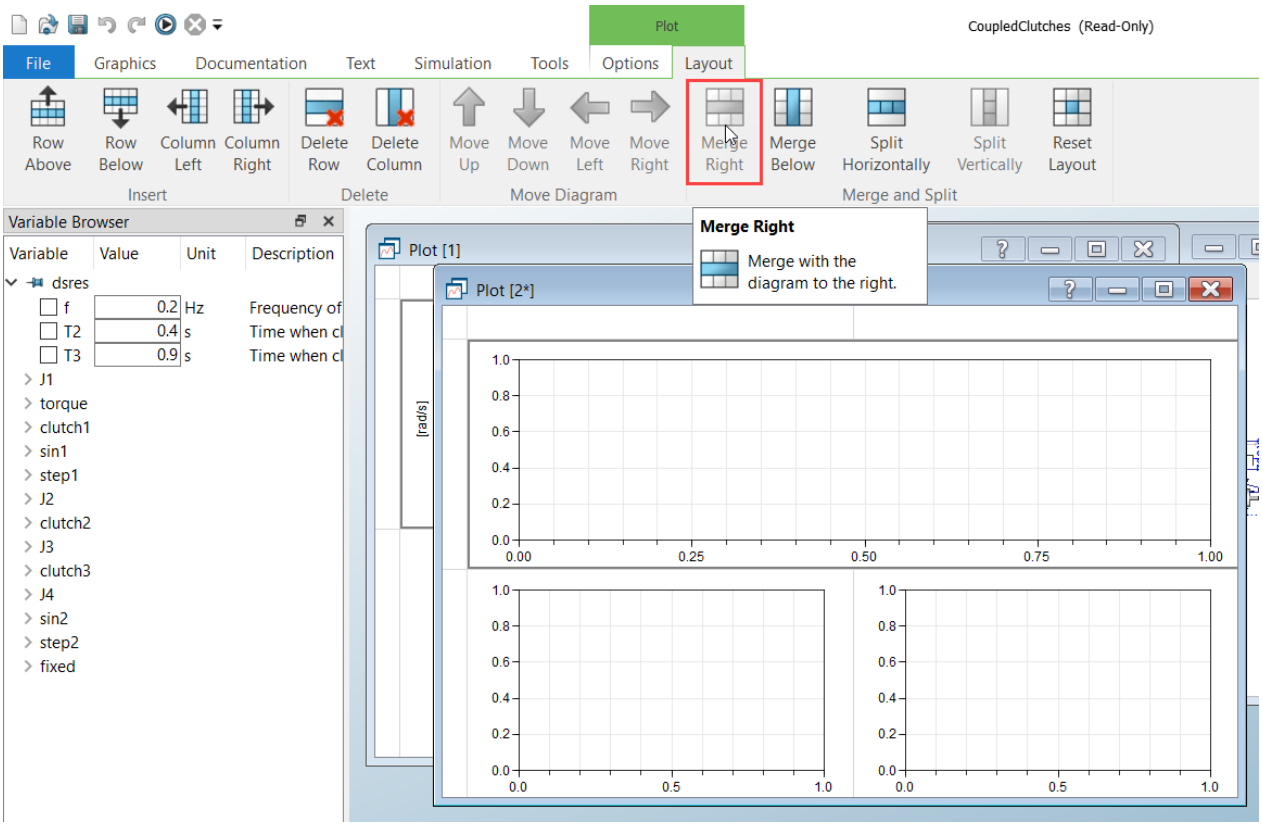
However, it is a good idea to get a good starting point before going to the **Layout** tab. Two commands in the **Options** tab offer good start points: **New Plot** and **New Diagram**, not by clicking them directly but by using the corresponding arrow:



The first selection above gives a new plot window with four diagrams; the second selection configures the active plot window to have four diagrams. Selecting the first option gives:



If you now go to the **Layout** subtab, and click the **Merge Right** button, you will get the result (since the top left diagram is active when giving the command, see above figure):



As you can see from the figure above, you have a number of commands to create and delete rows and columns, move cells, merge cells, and split cells. For details about the commands, see section “Main window: Plot tab: Layout subtab” starting on page 858.

Note! Merging of diagram cells only merges the cells, not the content. The base cell takes over the space and deletes the overtaken diagram.

(As a remark, as long as you don´t need to merge or split diagrams, you can work with columns and rows in the **Options** tab as well, using the context menu of the diagrams.)

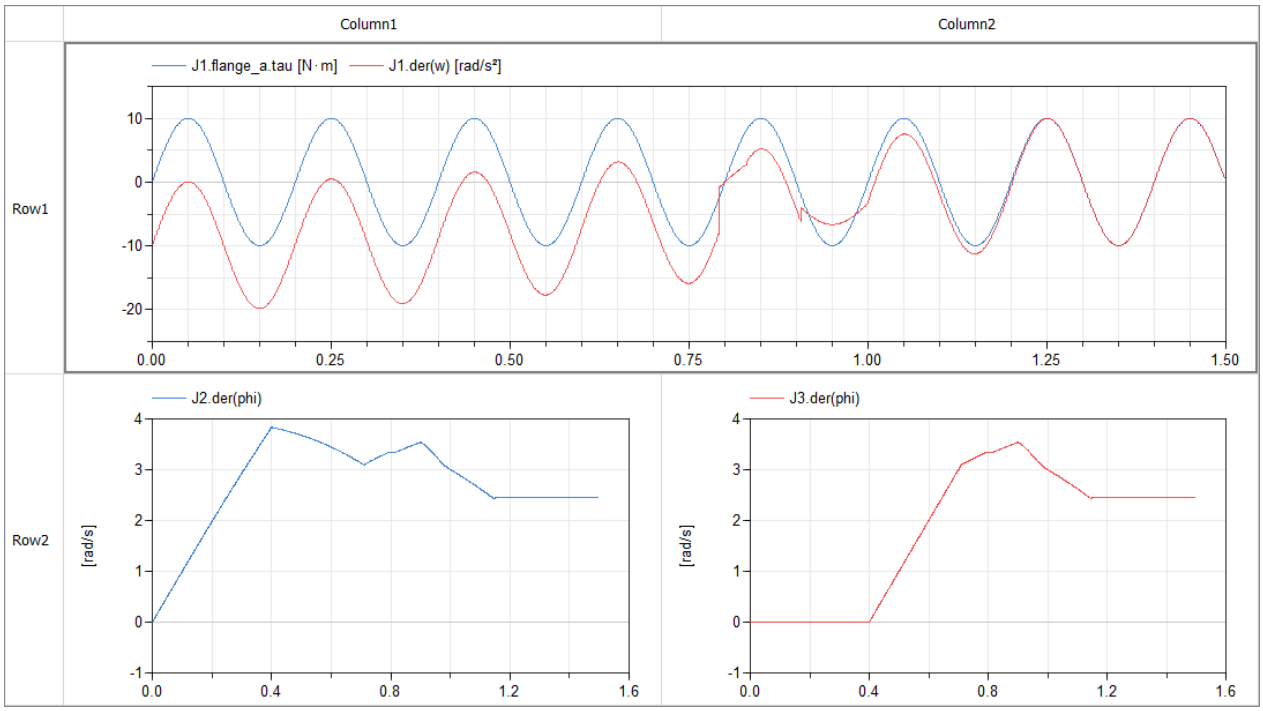
For scripting, see section “createPlot” starting on page 955. **Note**! There is no scripting support for merging diagram cells.

Setting header titles in 2D layout plot windows

You can set headers in 2D layout plot windows using the built-in function `plotRowColumnLabels`. As an example, applying, to the example above (with some curves added), the function call (by, for example, typing it in the command input line of the command window and pressing return)

```
plotRowColumnLabels({"Column1", "Column2"}, {"Row1", "Row2"})
```

the result will be:

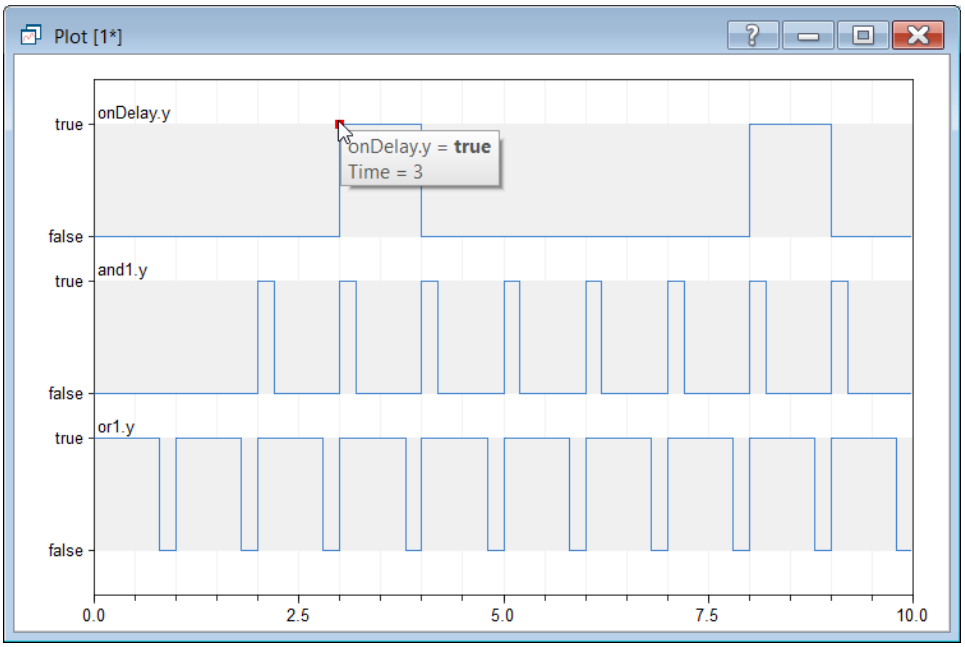


For details about the built-in function `plotRowColumnLabels`, see section “plotRowColumnLabels” on page 965.

Display of Boolean, Integer, and enumeration signals

Plotting the signals separately (default)

By default, Boolean, Integer, and enumeration signals are plotted separately: an example:

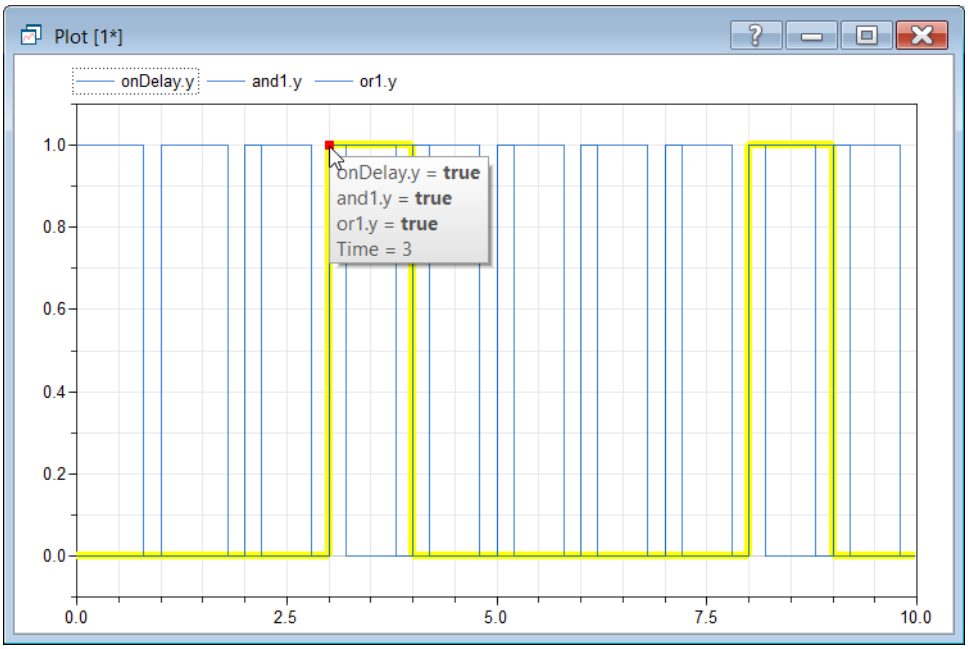


The default display of the signals have the following features:

- Boolean signals are displayed separately.
- Separate scale with labels false and true or enumeration values.
- Specific legend to the right.
- A light grey band is displayed in the background for Boolean signals.
- Only horizontal zoom (see later) is supported when only Boolean/enumeration signals are present.
- The usual scale is only used if other values than Boolean/enumeration signals are present.
- In the description the signal type (Boolean or Enum) and enumeration name is presented.
- This display of the signals separated corresponds to the flag `Advanced.Plot.SplitDiscreteCurves = true`.

Plotting the signals overlapping

By right-clicking in an empty area of the plot window presenting Boolean, enumeration, or Integer values, and deactivating **Split Discrete Curves**, the signals are displayed overlapping. Doing this on the plot window in the above figure, the result will be:



This display of the signals overlapped corresponds to the flag `Advanced.Plot.SplitDiscreteCurves = false`.

Display curves negated

In many cases you need to compare signals that by definition have opposite signs, for example, the current or torque going in and out of a component. Also, it is not uncommon that you may end up with inverted components, for example, in electrical circuits.



To show curves negated, you can use the command **Negated Curve** in the **Plot: Optons** tab. The curve will be shown negated, and a minus sign appears in front of the legend of the curve.

If you work with tables rather than curves, this command is available as for curves.

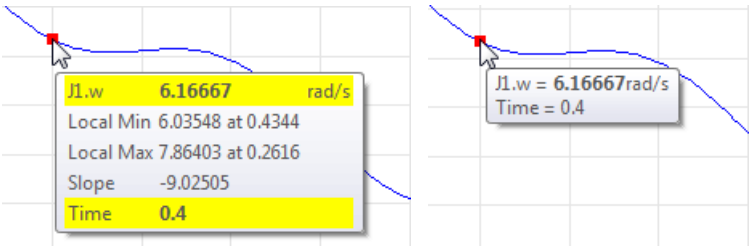
Duplicate diagram to new plot window

A plot diagram, with content, can be copied to a new plot window by the command **Generate > Duplicate Diagram** in the the **Plot: Options** tab.

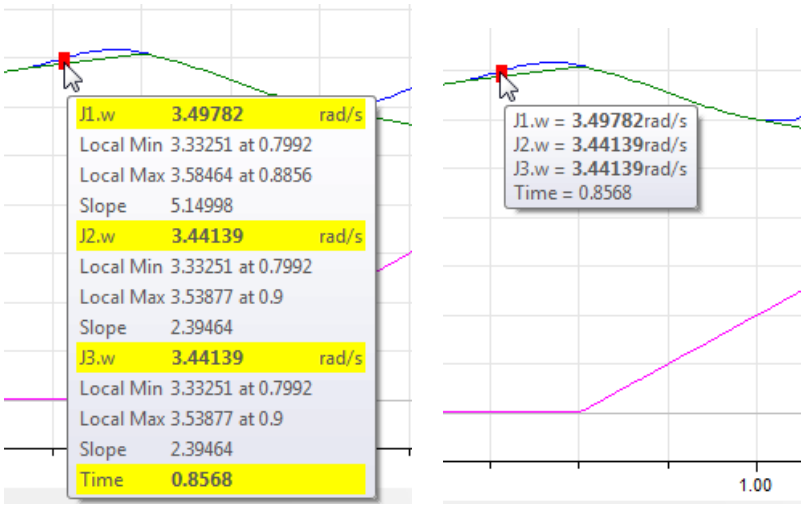
Dynamic tooltips for signals/curves

Resting the cursor over a plotted signal by default displays a tooltip with information about the closest data point and the time. That data point is also highlighted. It is possible to display additional information; the closest local min and max value, as well as the slope, calculated as the slope of the curve between the current point and the next point on the curve in the direction of the cursor. If several signals are close to the cursor, data for all of them are shown; if possible a common value for the independent variable is used.

Tooltip for a single signal; with and without additional information.

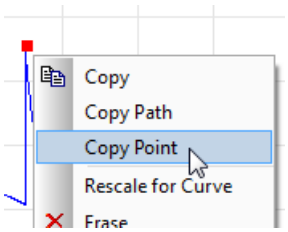


Tooltip for multiple signals and a common independent variable; with and without additional information.



The amount of information in the tooltip, and if the tooltip shall be displayed at all can be selected using the command **Plot: Options > Setup**, the **Options** tab. Selecting **None** no curve tooltip will be shown, selecting **Basic** the tooltip will display value and time, and selecting **Advanced** the tooltip will also display local min, local max and slope. The reason for not displaying a tooltip might be that if very many points in the curves are plotted, the curve tooltip slow down the drawing of the curves.

The entire text of the tooltip can be copied to clipboard by right-clicking and selecting **Copy Point** in the context menu.



This functionality can be used when e.g. wanting to transfer values from Dymola to any other tool for further processing.

Tooltip for legends

Resting the cursor over a legend in a plot, displays a tooltip showing the signal name, description, unit, result file name (including the timestamp, that is, the last time that the file was modified), and sequence number.



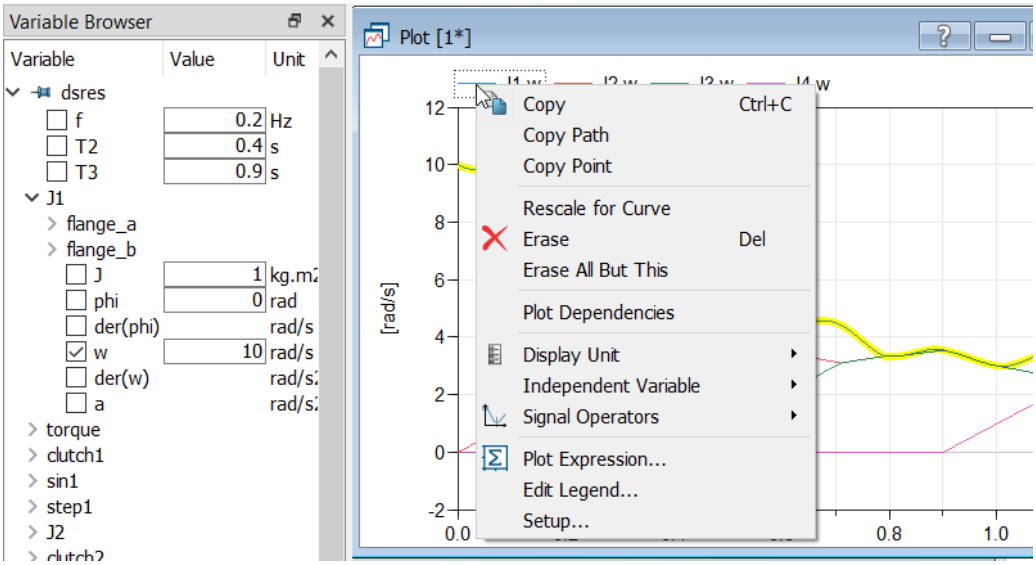
Note that the corresponding curve is highlighted in the plot window. It will stay highlighted as long as the legend symbol is selected.

If wanted, only the highlighting, but no tooltip, can be displayed. If this is wanted, uncheck **Show tooltip for legend** in the **Legend** tab, using the command **Plot: Options > Setup.**

Context menu for legends

Note that the context menu for the corresponding signal will be displayed by right-clicking the legend.

When the context menu for a legend is displayed, the corresponding signal will be highlighted in the plot window.



Tooltip for x-axis and y-axis

Resting over the x-axis gives a tooltip displaying information of the x-axis; for example `Time [s]`. Resting over the y-axis gives a tooltip displaying information about the y-axis.

Moving in the plot window

To move the curves in the plot, press and hold the **Ctrl** key as well as your left mouse button and then move your mouse. The curves will be dragged along with your mouse cursor. To stop, simply let go of the left mouse button.

As an alternative, you can use the mouse wheel to move the curves in a plot along the y-axis and x-axis the following way:

- **Mouse wheel** moves the curves along the y-axis only
- **Shift+Mouse wheel** moves the curves along the x-axis only

The last command is by default concurrent; it works on all diagrams in the active plot window at the same time. However, to only have it work on the active diagram, you can deselect the command button **Plot: Options > Sync Pan/Zoom**. For more information about this command, see “Plot: Options > Sync Pan/Zoom” on page 841.

Zooming in the plot window

The user can interactively zoom in on interesting parts of a diagram and easily zoom back to a previous zoom level. Certain operations reset the zooming.

There are several ways to work with zooming in the plot window. Some of them are concurrent; that is, they work at the same time on all diagrams in a plot window.

The zooming also applicable to the diagram layer is to use **Ctrl** + dragging of the mouse button *to the right*/scrolling the mouse wheel. This alternative is presented first.

The other basic idea is to use the *left* mouse button to zoom in (and create a stack of zoom levels) and to use the context menu for zooming out in that stack (or to reset the zooming level to the initial one).

There are a number of alternatives using the left mouse button:

- Basic zooming – dragging with the left mouse button.
- Zooming with maximum scaling in y direction – pressing **Alt** while dragging with the left mouse button. This zooming is also concurrent.
- Vertical zooming – pressing **Shift** while dragging vertically.
- Horizontal (time axis) zooming – pressing **Shift** while dragging horizontally. This zooming is also concurrent.

Zooming in/out using the Ctrl key and mouse

By pressing the **Ctrl** key while working in the plot window, zooming is possible using the mouse, it two ways:

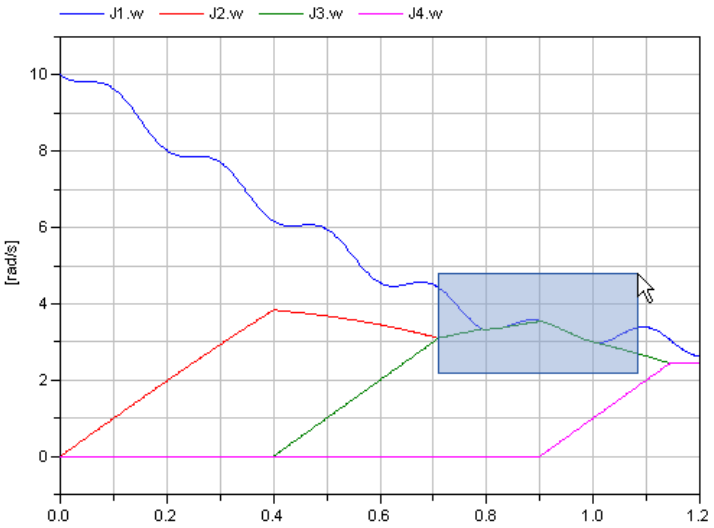
- Mouse wheel: press and hold the **Ctrl** key and scroll the mouse wheel forwards/backwards to zoom in/out.
- Mouse move: press and hold the **Ctrl** key and the mouse button *to the right*; then move the mouse forwards or backwards to zoom in/out.

Releasing the keys will store the present zoom level in the stack of zoom levels, creating such levels will mean that the context menu can be used to zoom out using the **Zoom out** command. For that command, and other ways to zoom out/rescale, please see the sections "Zooming out/rescaling using context menus" on page 710 and "Rescaling the plot window using commands/buttons" on page 710.

Zooming in by dragging the left mouse button

This can be said to be the easiest and most basic zooming; pressing the left mouse button and spanning a rectangle[10]. Zooms in on the drawn rectangle, which is scaled to fit the diagram.

[10] On Linux, the rectangle will be transparent, the border dashed.

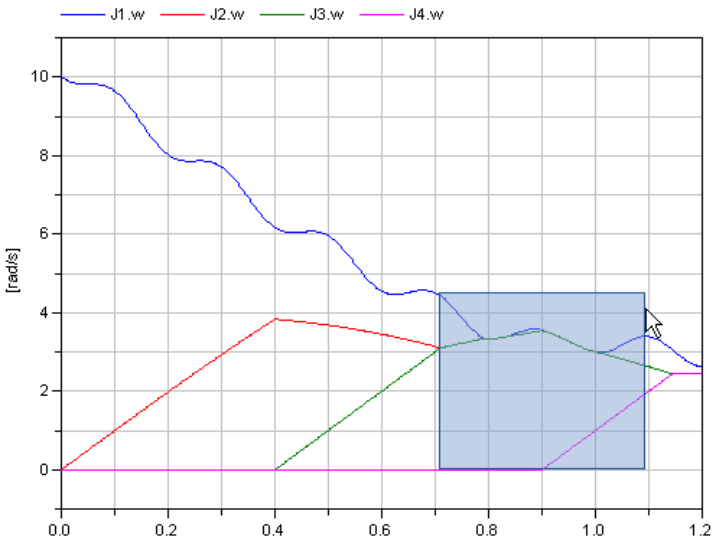


Zooming out/rescaling is performed using the context menu or buttons, please see the sections “Zooming out/rescaling using context menus” on page 710 and “Rescaling the plot window using commands/buttons” on page 710.

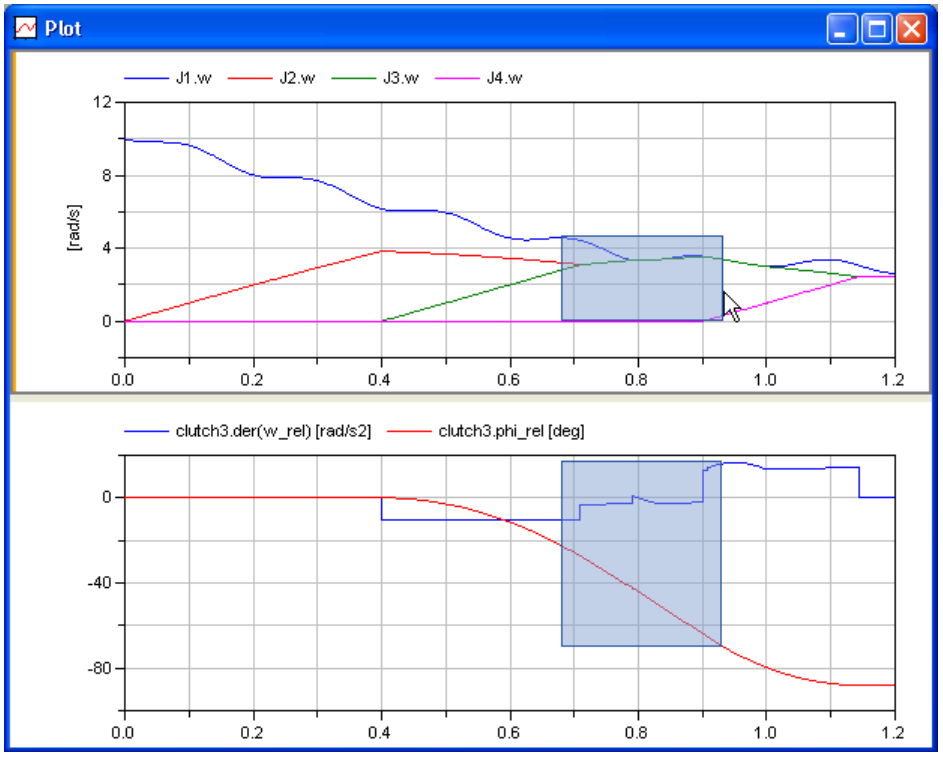
Zooming with maximum scaling in y direction

By pressing **Alt** while pressing the left mouse button and spanning the rectangle to be zoomed in[11], zooming with maximum scaling in y direction is obtained. This includes all curves that are in the spanned rectangle.

[11] On Linux, **Ctrl+Shift** have to be used instead of **Alt**.



Zooming with maximum scaling by default supports concurrent zooming; the rectangle to be zoomed will be spanned in all diagrams in the active plot window:

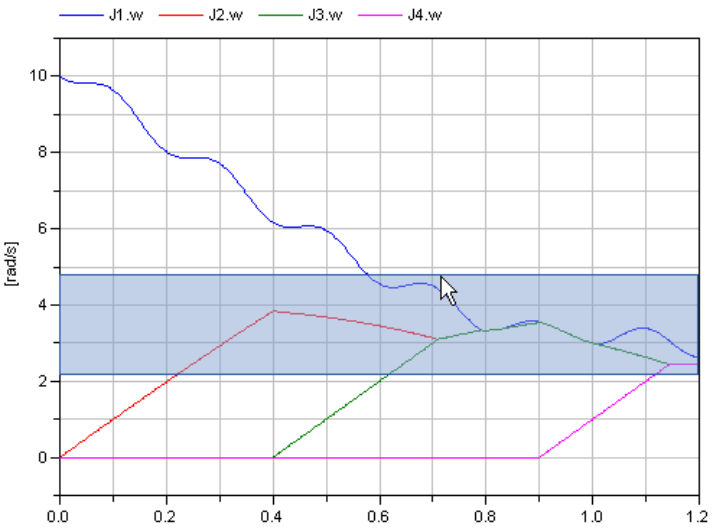


However, if you want to have the command to only work on the active diagram, you can deselect the command button **Plot: Options > Sync Pan/Zoom**. For more information about this command, see “Plot: Options > Sync Pan/Zoom” on page 841.

Zooming out/rescaling is performed using the context menu or buttons, please see the sections “Zooming out/rescaling using context menus” on page 710 and “Rescaling the plot window using commands/buttons” on page 710.

Vertical zooming

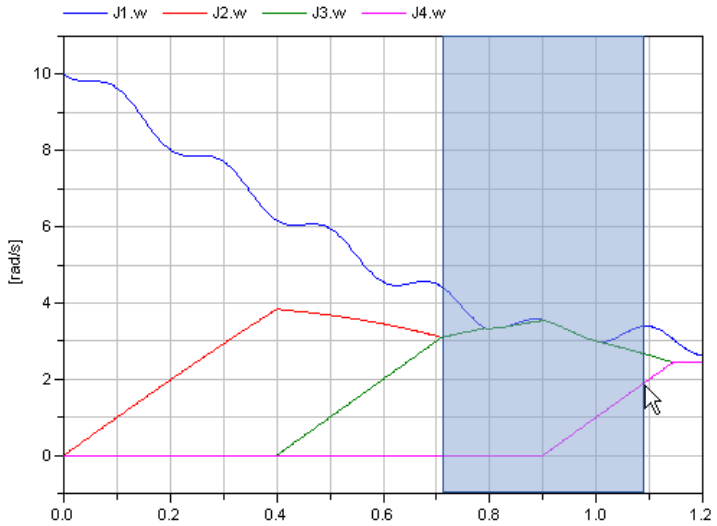
By pressing **Shift** while dragging the left mouse button up or down, a pure vertical zooming is obtained:



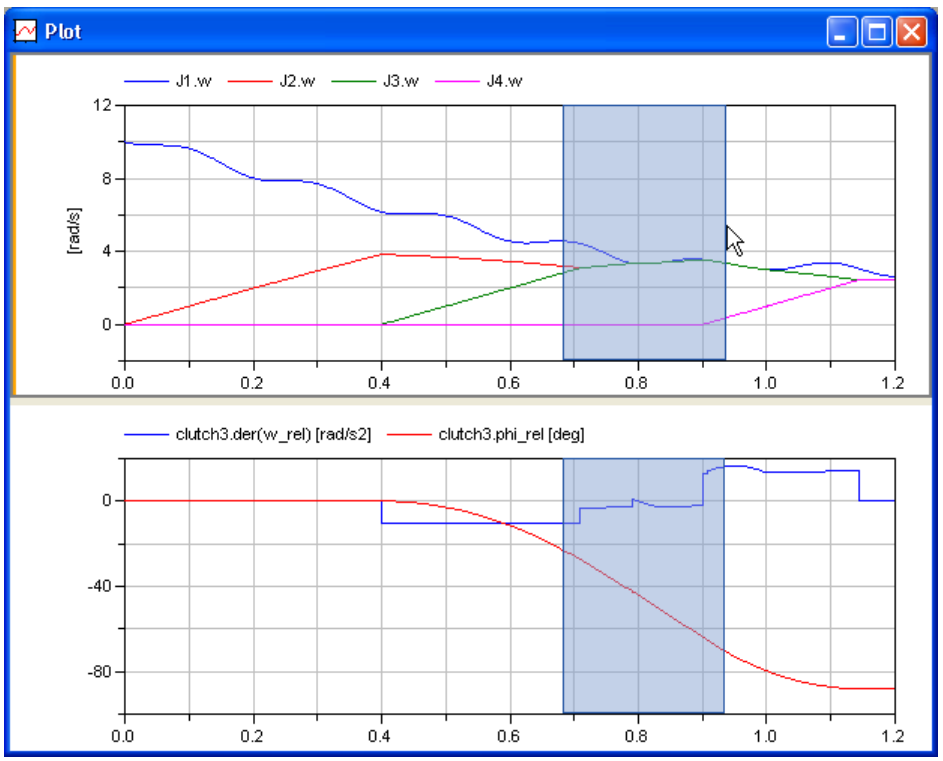
Zooming out/rescaling is performed using the context menu or buttons, please see the sections “Zooming out/rescaling using context menus” on page 710 and “Rescaling the plot window using commands/buttons” on page 710.

Horizontal (time axis) zooming

By pressing **Shift** and dragging the left mouse button to the right or to the left, a pure horizontal (time) zooming is obtained:



Horizontal (time axis) zooming by default supports concurrent zooming; the time to be zoomed will be used for all diagrams in the active plow window.

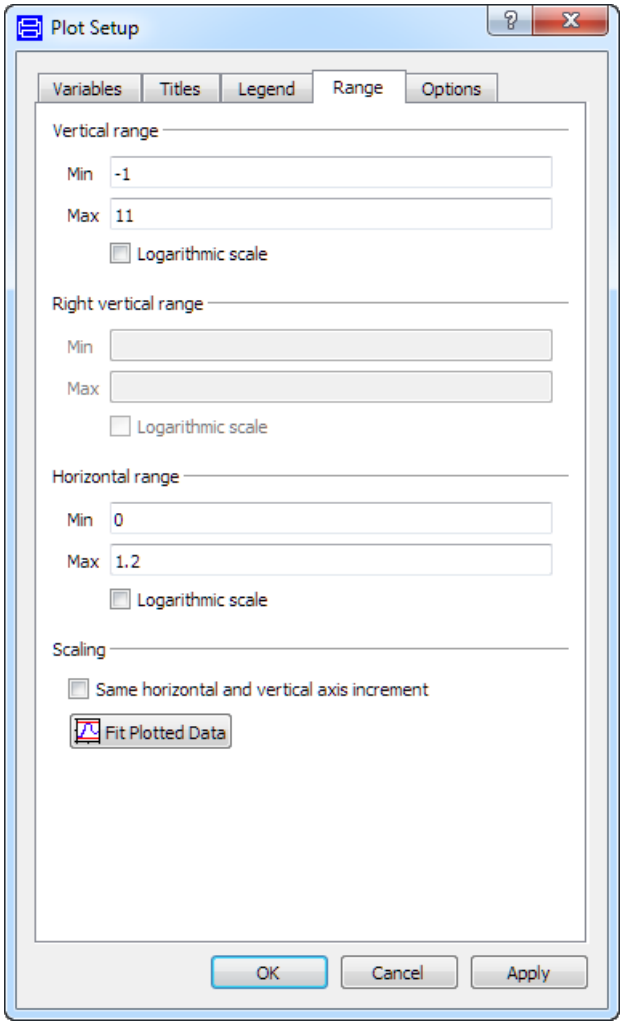


However, if you want to have the command to only work on the active diagram, you can deselect the command button **Plot: Options > Sync Pan/Zoom**. For more information about this command, see “Plot: Options > Sync Pan/Zoom” on page 841.

Zooming out/rescaling is performed using the context menu or buttons, please see the sections “Zooming out/rescaling using context menus” on page 710 and “Rescaling the plot window using commands/buttons” on page 710.

Zooming in/out using the Range tab in the plot setup menu

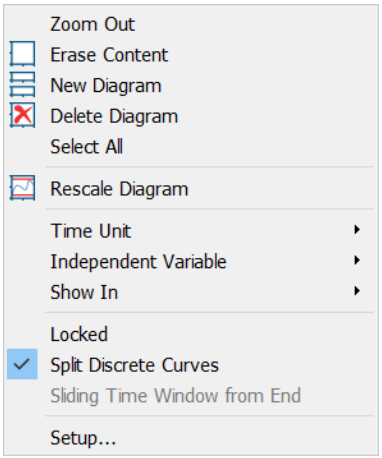
The tab looks the following:

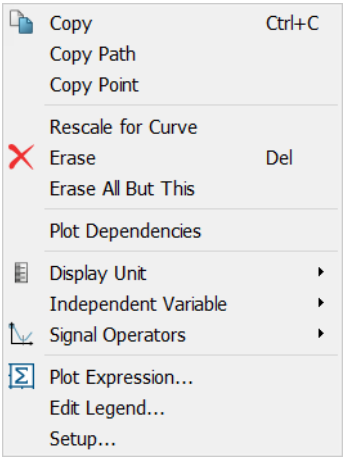


For more information, please see the section “Plot: Options > Setup” starting on page 842, the **Range** tab.

Zooming out/rescaling using context menus

By right-clicking in an empty area of a diagram a context menu is available (below left), by right-clicking on a curve or a legend another context menu is available (below right).





Using these menus, the following zooming actions can be performed:

Zoom out returns to the previous zoom level. Limits defined in the **Range** tab in the plot setup menu are also used in the context.

Rescale Diagram will reset the zooming to the initial value for all signals in the diagram. For rescaling of all diagrams in the active plot window, please see below.

Rescale will be available only if a curve is selected (e.g. by right-clicking on it). This command will rescale the current diagram to display the *selected* curve with maximum scaling.

Zooming in is not part of the context menu, it can be performed in a number of ways, please see above.

(The remaining entries in the context menu are described in section "Context menu: Plot window" on page 874 and "Context menu: Plot window – curve and legend" on page 876.)

Rescaling the plot window using commands/buttons

Two buttons in the plot toolbar (and corresponding commands) can be used to rescale an active plot window/active diagram.



The command **Plot: Options > Rescale All** will rescale all diagrams (reset them to initial zooming) in the active plot window. This command is not available in the plot window context menu.

The command **Plot: Options > Rescale** will reset the zooming to the initial value for all signals in the diagram. This command is also available in the plot window context menu.

Applying time sliding window

You can apply a time sliding window, that is, displaying a window of a selected time range that follows the simulation in such a way that the last simulate result is displayed to the right in the window. You can do this in two ways.

(A good signal to use when testing this feature is `ElectrifiedPowertrains.Examples.ElectricDrives.LossySwitchInvertor.multiSensor.power` during a simulation with the stop time 1 second.)

Applying time sliding window from zooming

During simulation, zoom in a curve, right-click the diagram (outside the curve) and select **Sliding Time Window from End**. This will create a time sliding window, with the time range the same as the zoomed time range. This is indicated by a check mark in front of the option in the context menu. The time range will also be inserted in the corresponding option in the plot setup, see below.

Important! Once you have activated such a sliding time window, you can only change the time range of it using the plot setup, see below. If you zoom such a window, you will bet a static zoomed window.

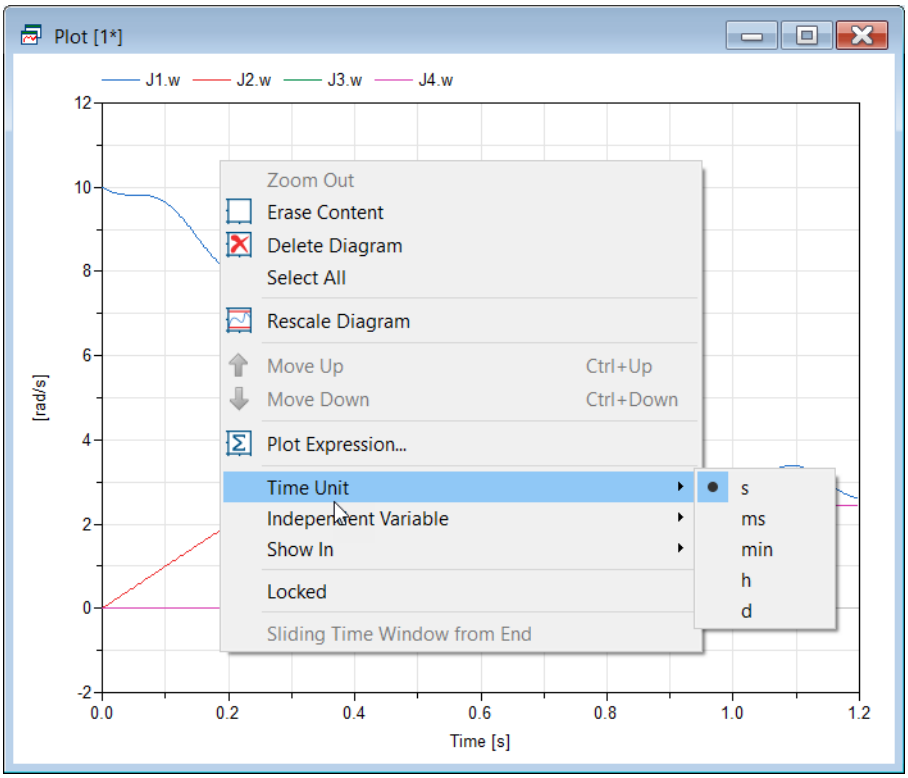
Applying time sliding window from the plot setup

During simulation, select the command **Plot: Options > Setup**, the **Options** tab. If you now set the value of **Time window from end [s] (if > 0)** to a value larger than 0, a time sliding window with the time range of that value will be applied when you click **OK** or **Apply**.

Changing time unit on x-axis

The default time of the x-axis when creating a new plot window is the simulation time unit. This is a general setting that can be changed both from the simulation setup and from the animation toolbar. See section “General tab” starting on page 800 for details.

The local time unit of the x-axis in the active plot window can be selected by the command **Time Unit** in the context menu of the plot window. (Note that no curve etc. can be selected, neither must the cursor be over a curve or a legend if this command is to be shown in the context menu.)



The default time is the simulation time unit (which itself has a default value of s (seconds)), If the finally selected time unit is another than seconds, that time unit selection will be displayed in the tooltip of the x-axis, as e.g. `Time [hours]`, and as a legend. The selections available are the same as the defined displayUnit for time, and can be changed by changing the displayUnit for time. For more about the use of displayUnit in general, see next section.

Note that if the simulation time unit (the general setting) is changed, the time unit of the currently active plot window is changed as well; it overruns the local time setting.

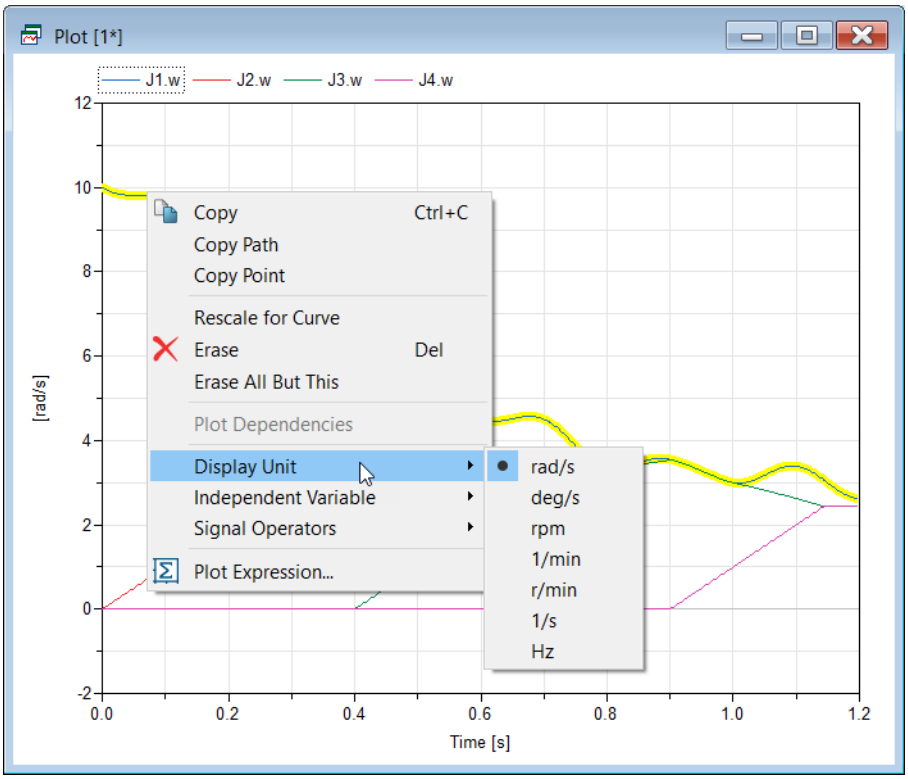
Changing the displayed unit of signals

Changing of the displayed unit of signals can be done in two ways, either directly (by the command **Display Unit** in the **Plot** tab, or by the curve context command **Display Unit**), or using the settings available in the plot window.

Selecting the displayed unit directly in the context menu

Taking up the context menu for a curve in a plot window and selecting **Display Unit** might display the following:

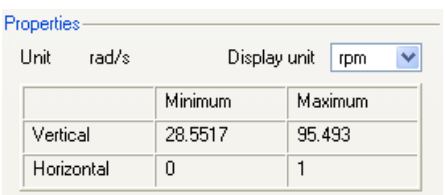
Selecting display unit.



In this case rad/s (radians per second) is displayed, but it is very easy to select another display unit.

It is possible to change the display unit of a signal in the plot window, for example from *Pa* to *MPa* or from *rad/s* to *rpm*, if suitable unit conversions have been defined. Changing the display unit for a signal will change the display unit for *all signals* in the diagram with the same display unit. Diagram tooltips show units.

After a signal has been plotted, the display unit can be changed using the plot setup menu (reached by **Plot: Options > Setup**). The display unit is chosen from a list of known derived units. The last selected display unit becomes the default display unit when another signal with the same fundamental unit is plotted.



Unit conversions and the initial default display unit can be specified in command scripts as needed. For example,

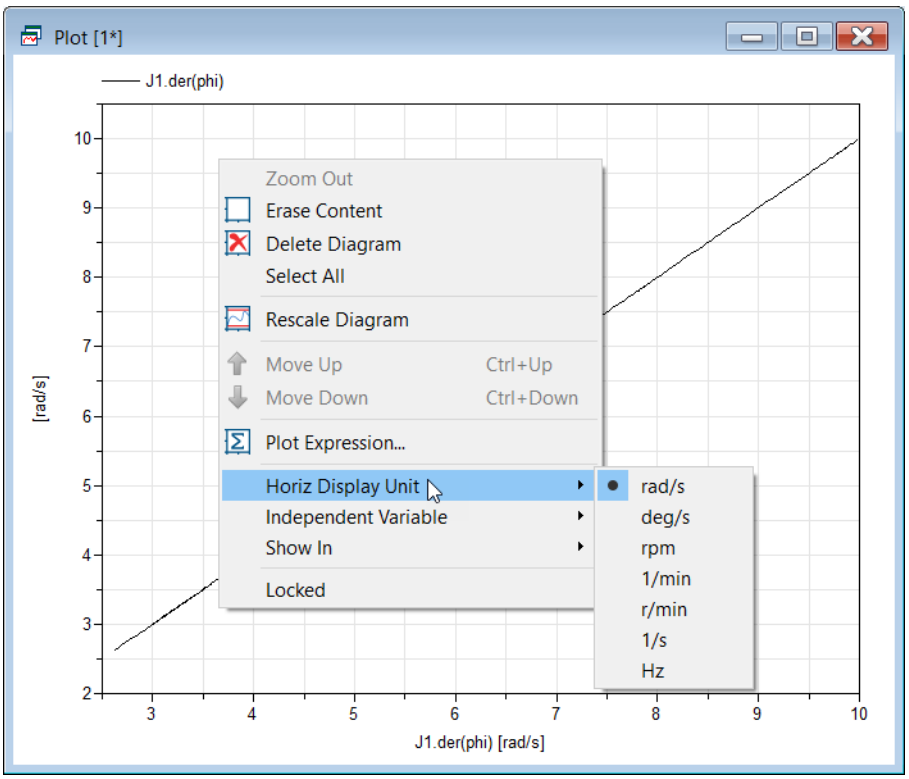
```
defineUnitConversion("Pa", "MPa", 1e-6, 0);
    // scale and offset
defineDefaultDisplayUnit("Pa", "MPa");
    // use MPa by default
```

Three files, located in the folder `Program Files\Dymola 2023\insert` deals with unit conversion. The set of common unit conversions which are used by default can be found in `displayunit.mos`. Additional US unit conversions are defined in `displayunit_us.mos`; which are used by default can be changed in `dymola.mos`. (Changing any of these files might require administrator rights.)

Please see chapter “Developing a model”, section “Advanced model editing”, sub-section “Display units” for more information.

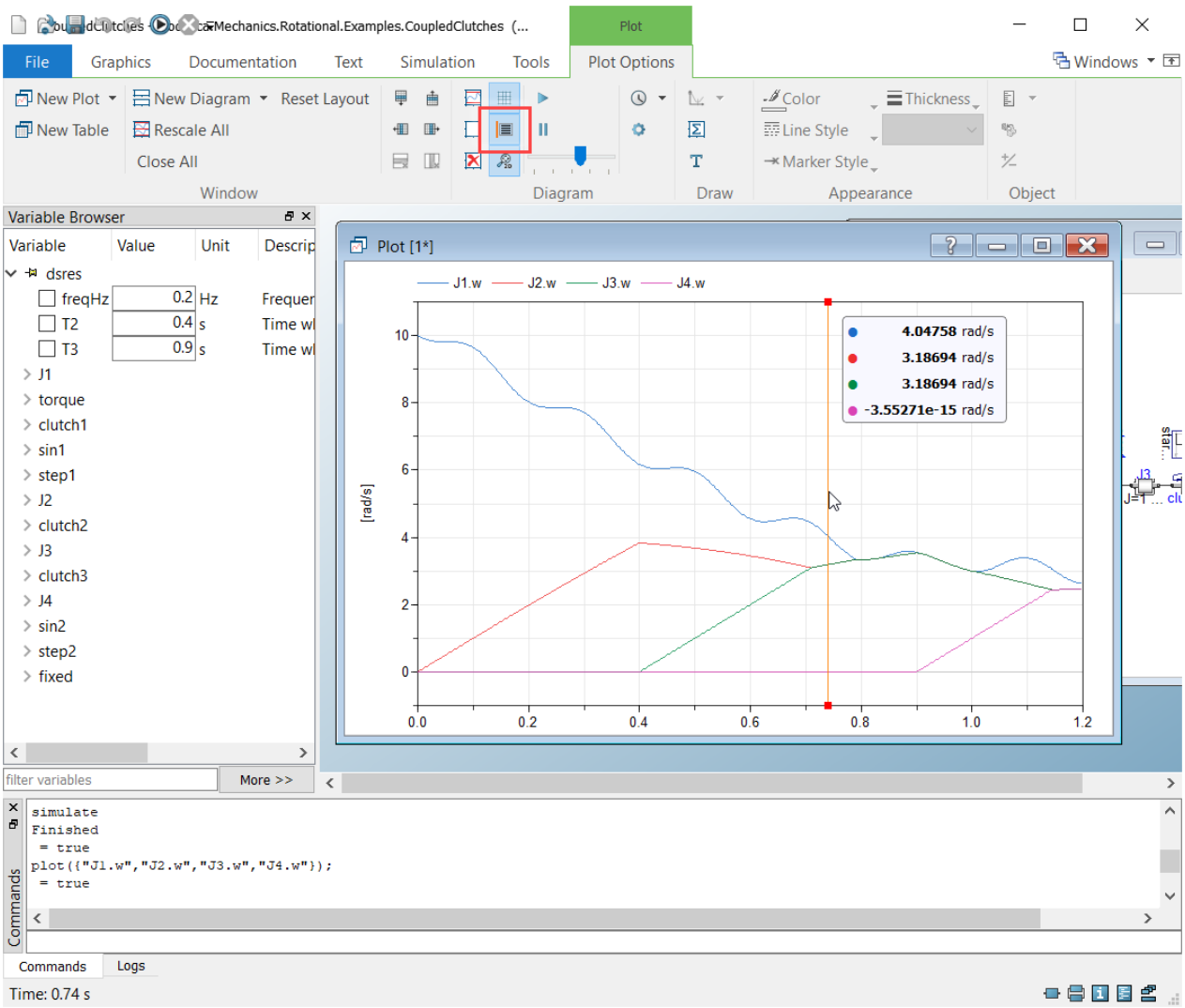
Changing the displayed unit of x-axis when having other variable than time

When you have selected another variable than time for the x-axis, you can select display unit for the x-axis by the context command **Horiz Display Unit** for the plot diagram, when you have nothing selected:



Using measurement cursor for plots

A measurement cursor can be used for plots:



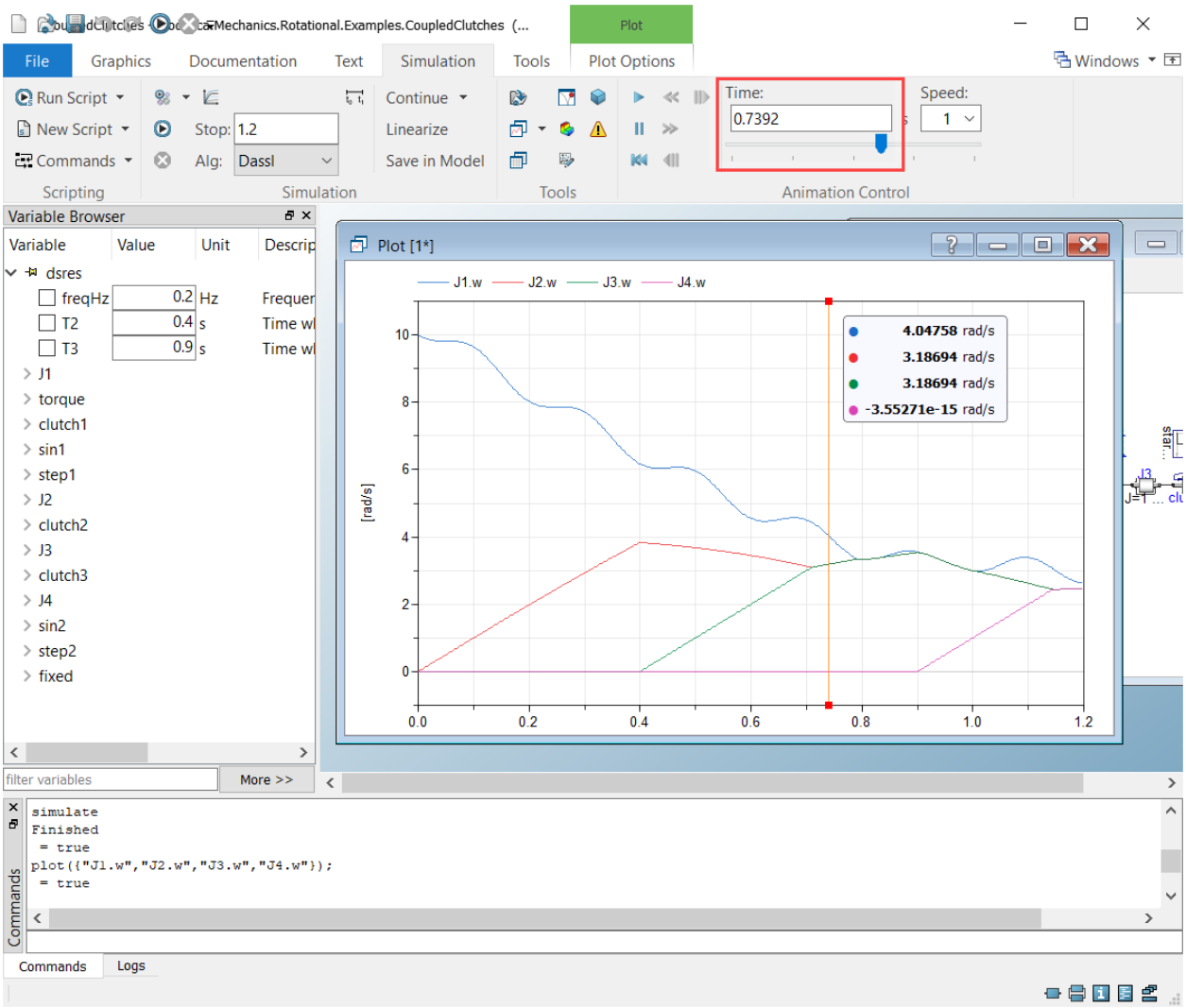
By default, the measurement cursor is located over the left y axis. You move it by dragging the cursor line itself.

The x-axis value (mostly time) is displayed in the very lower left corner of the Dymola window when you move the cursor; see image above. (Note that the measurement cursor can be used when the x-axis is not time.)



To display the measurement cursor, use the command **Plot: Options > Measurement Cursor** (see framed command in image above).

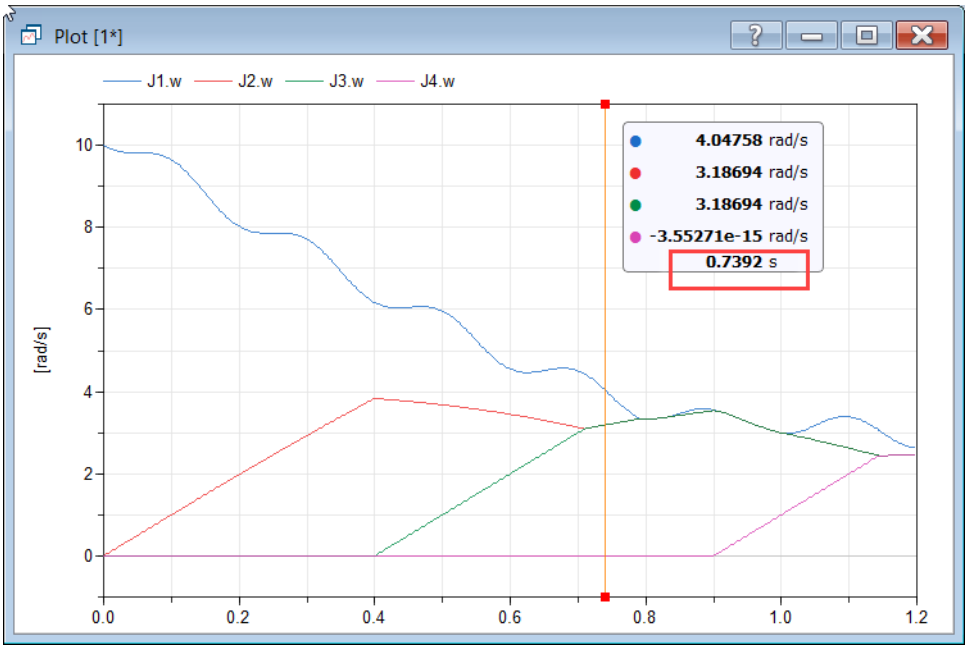
To have a better display of the x-axis value if it is time (it tends to just disappear if you move the cursor away from the measurement cursor), open the **Simulation** tab instead, you will then have the time, and a handle to drag:



However, if you want to see the time always, the best is to set the flag

```
Advanced.Plot.MeasurementTime = true
```

This activates displaying the time in the data box:



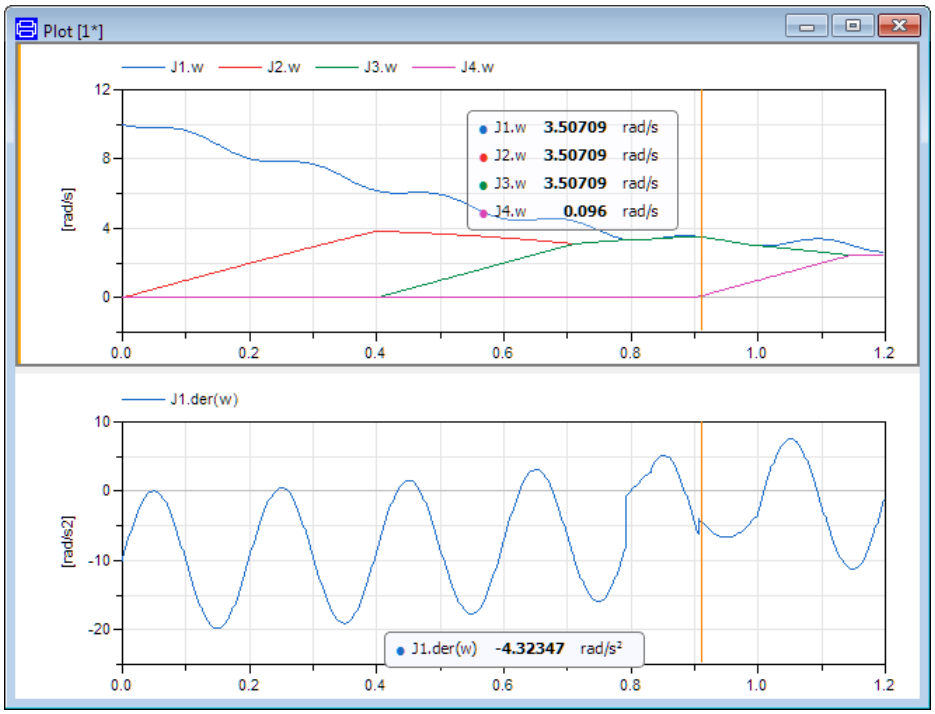
(This flag is by default false.)

Concerning the values displayed in the data box, note that they are interpolated values, not the values of the closest points.

By default, the data box is docked to the measurement cursor; it moves with the cursor. You can undock it by dragging it. By double-clicking on an undocksed tooltip, you dock it again.

The data box has by default a background color; you can click on an empty area inside it to get it transparanet. By clicking you toggle the visibility of the data box background.

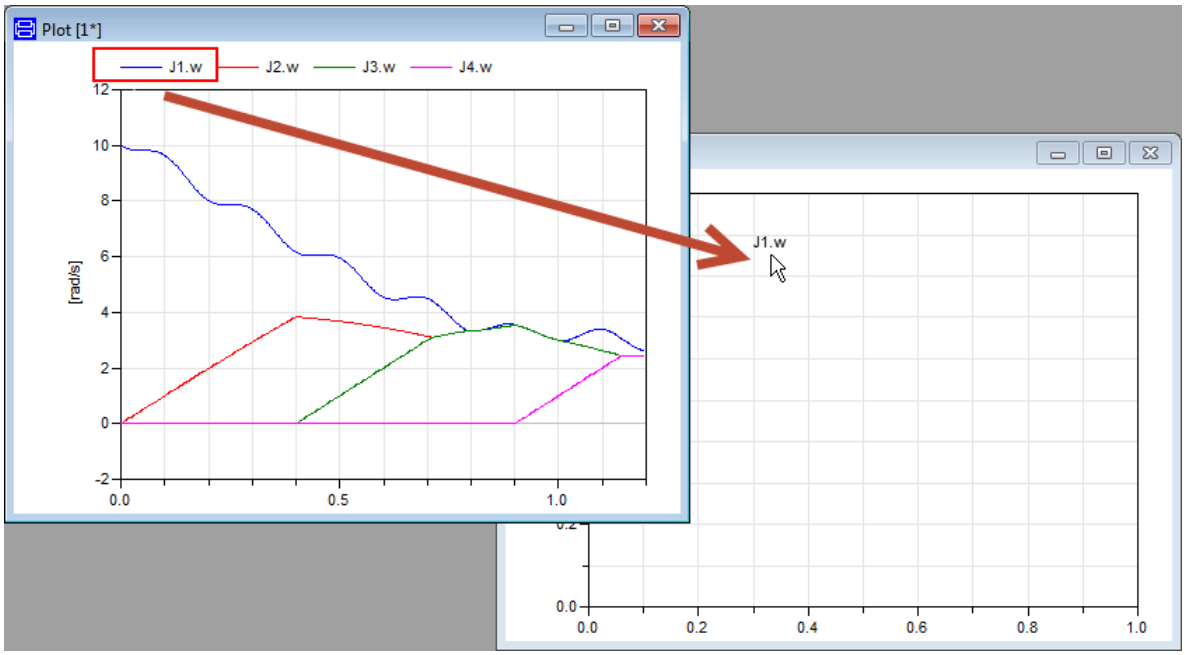
If you have several diagrams in a plot window, the cursor line is displayed in each diagram, but the cursor data box can be changed individually when it comes to docking and background visibility:



If a plot is displaying signals of an animation, and the measurement cursor is displayed, the measurement cursor will be dynamically updated to display the current animation time. The measurement cursor is updated in all present windows.

Drag and drop of curves between plots/diagrams

Curves can be dragged and dropped between plots or diagrams by dragging and dropping the corresponding curve legend:



Signal operators are included, if any.

Note that you can drag and drop several curves at the same time, by first selecting several curve legends by clicking on them.

Curves can be copied by keeping **Ctrl** pressed while performing the dragging of the corresponding legends.

Drag and drop of curves between plots and tables

You can drag curves between plot windows and table windows. When dragging from table windows, drag the header in the left column.

You can copy by keeping **Ctrl** pressed while performing the dragging.

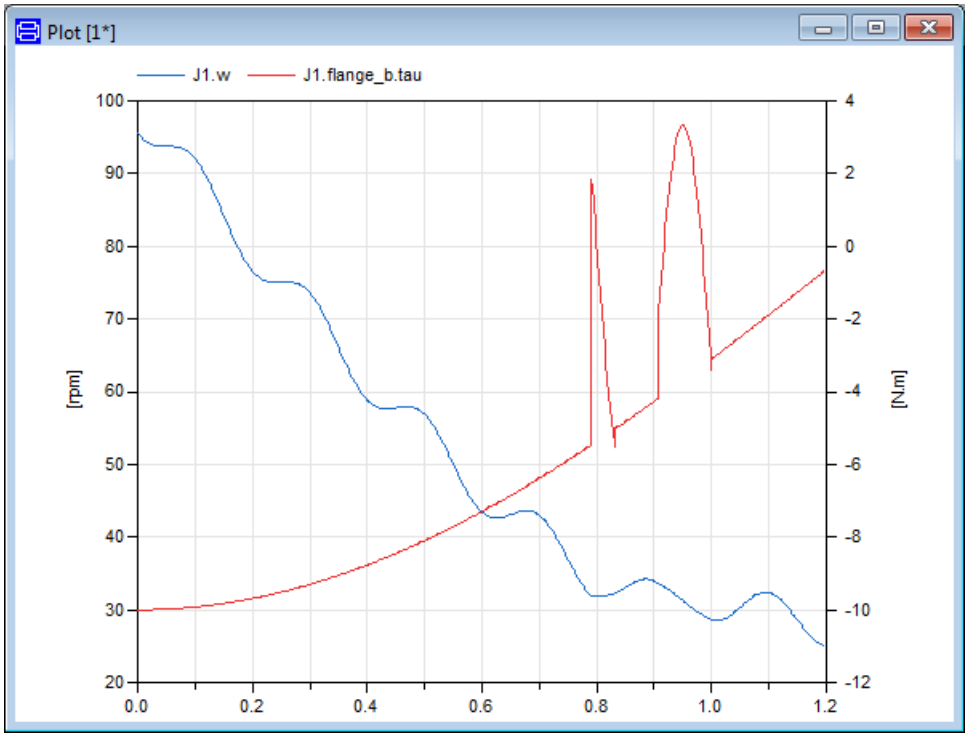
Lock plots and tables

To prevent changing plots or tables by accident, for example adding or deleting curves by clicking in the variable browser, the context command **Locked** can be used to lock the plot or table.

When locking is applied, the plot window or table window cannot be changed from outside (by for example selecting/deselecting signals in the variable browser, or by resimulation). By default, the locking is not applied.

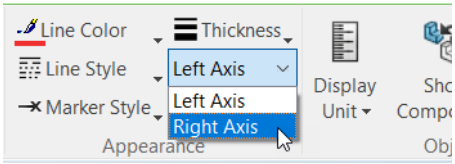
Muliple y-axis in plot window

Two vertical axes (y-axes) are supported in a plot window, one to the left (the primary axis), and one (secondary) to the right.



Gridlines are drawn for the primary axis only.

Plotting always uses the primary (left) axis by default. To use a secondary axis for a variable, select the corresponding curve and, from the Plot tab, select **Right Axis** from the menu:



You can also use the plot setup menu, in particular to move multiple variables to the secondary axis. In the plot setup menu, in the **Variables** tab, select one or more variables, and then, in the **Vertical axis** combo box, select **Right**. The variable(s) will be associated with the secondary axis to the right. If such an axis does not exist, it will be created.

Editing title and range for the secondary y-axis is done similar to the primary y-axis, in the plot setup menu.

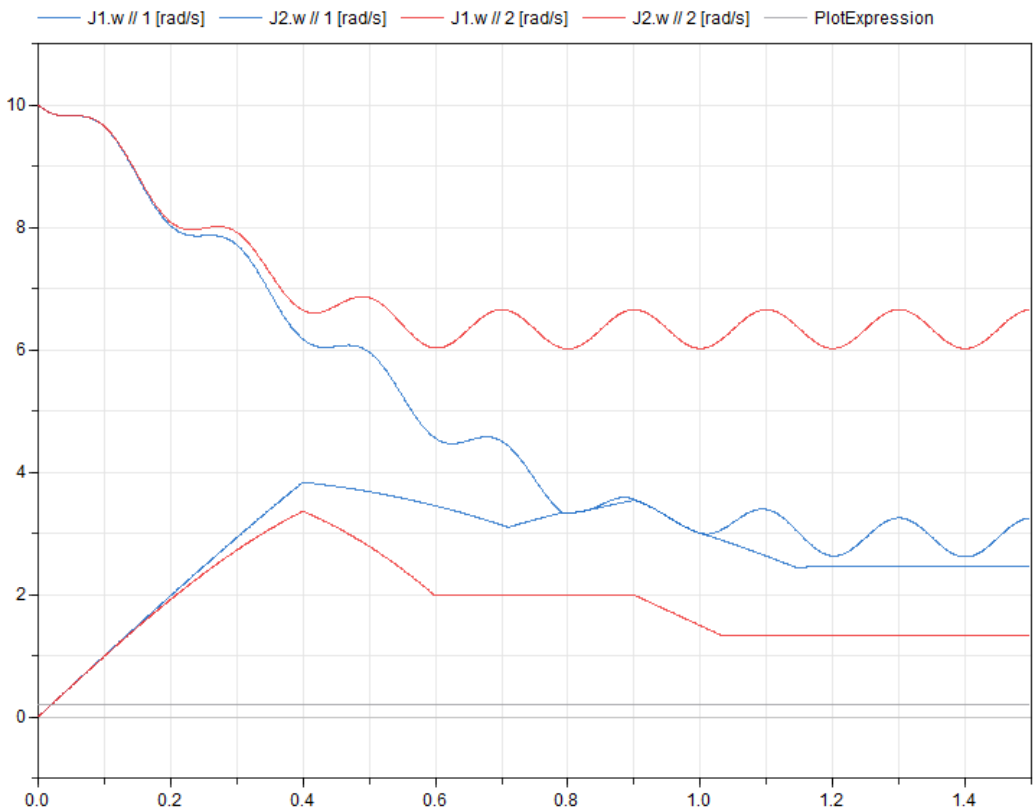
Assign different colors to curves from different simulation result files

You can apply a color scheme that assigns a unique color for curves based on the simulation result file, so all curves from one result file get one color, and all curves from the next result file get another color.

The feature is controlled by the setting **One color per result file** in the plot setup, reached by the command **Plot: > Options > Setup**, the **Options** tab. The setting is by default not activated. This corresponds to the flag `Advanced.Plot.ResultColors = false`. The setting is saved between sessions.

Note! This feature is an option when displaying curves,but when using **Compare Results** in the variable browser, the feature is always applied, independent of if the feature is activated or not for other cases.

An example of the feature:



An example of a plot expression is added in the image above; curves that does not belong to a result file, for example, plot expressions, are displayed in gray color.

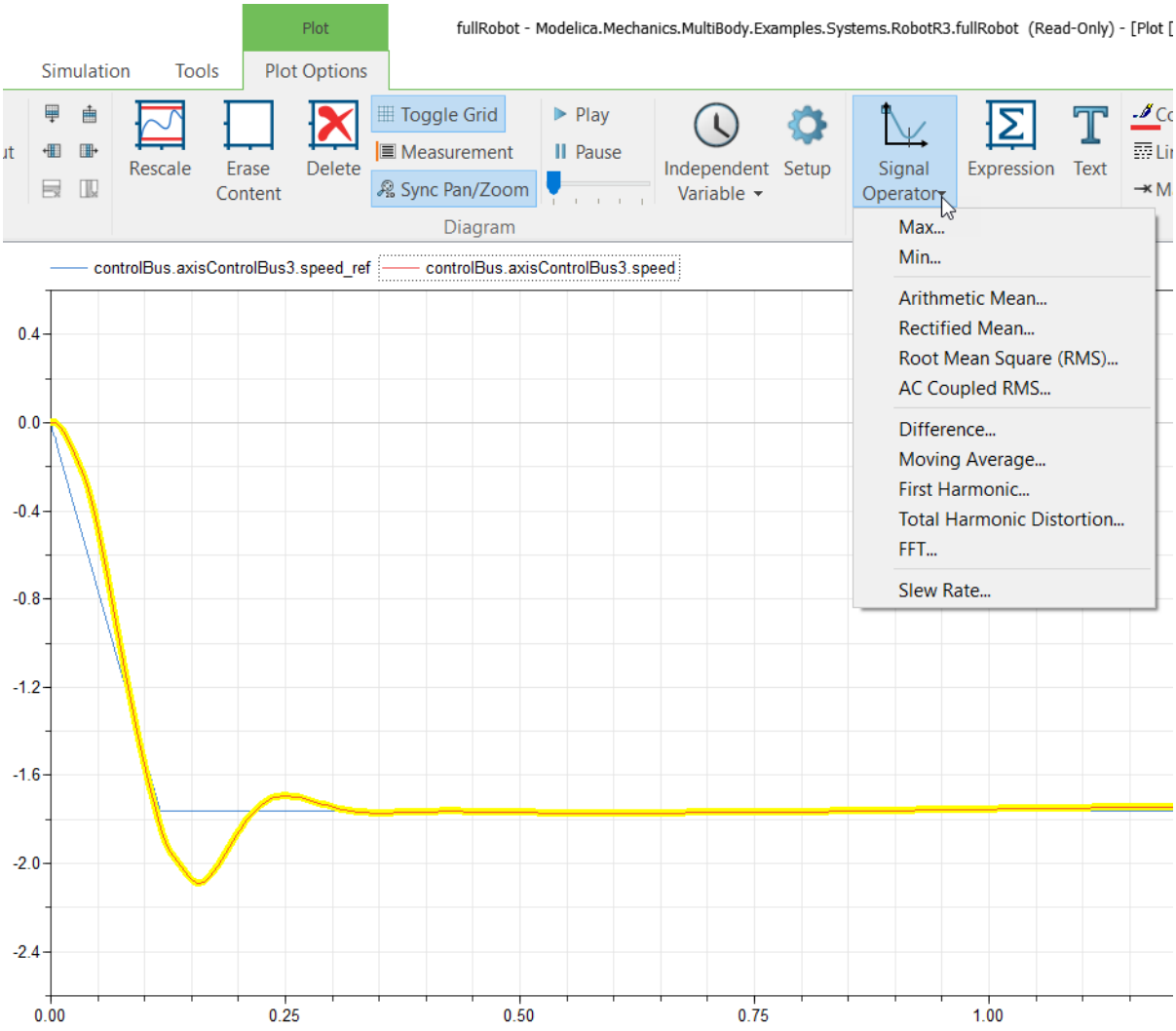
Displaying signal operators

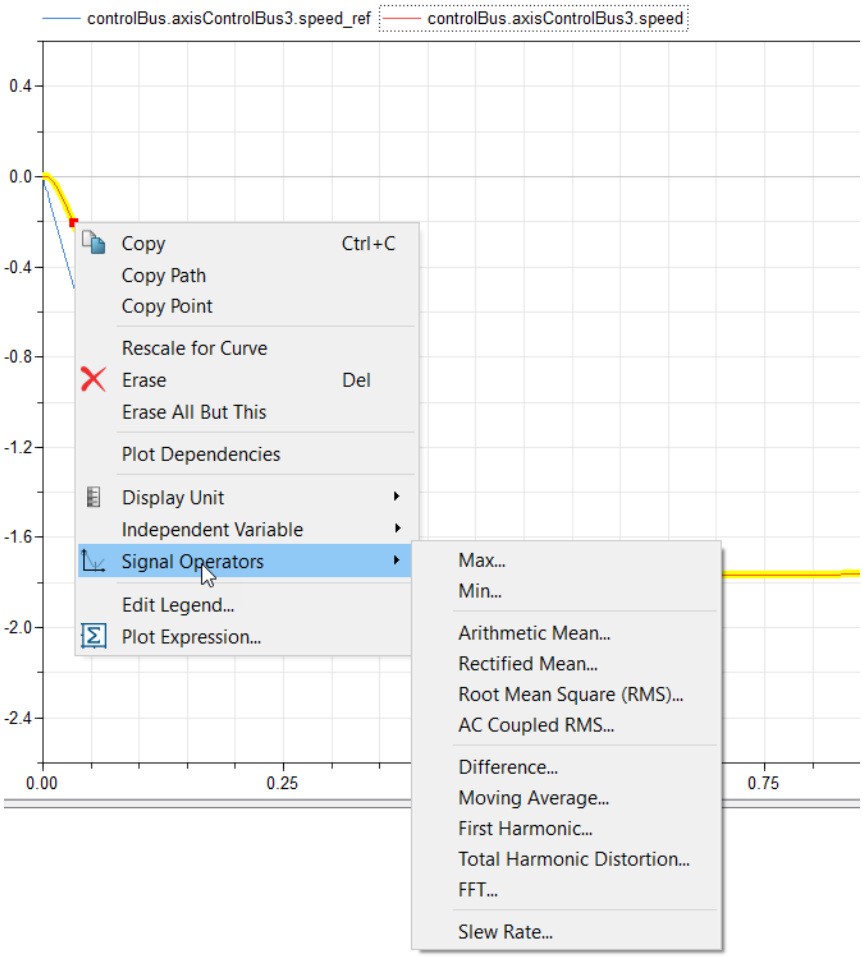
Available signal operators

A fixed set of signal operators are available using the context menu of signals in plot windows. The operators are applied over a user-defined time range, and the following operators are available (for other independent variables than time, see below):

- Minimum and maximum value
- Arithmetic mean
- Rectified mean
- Root-mean-square (RMS)
- AC-coupled RMS
- Difference
- Moving Average (with possibility to also compute and display first-order derivatives)
- First harmonic
- Total harmonic distortion (THD)
- FFT (Fast Fourier Transform)
- Slew rate

The **Signal Operator** selection can be reached from the **Plot** tab (when a curve is selected) or from the context menu of the curve:


Plot
fullRobot - Modelica.Mechanics.MultiBody.Examples.Systems.RobotR3.fullRobot (Read-Only) - [Plot [
Simulation
Tools
Plot Options
Rescale
Erase Content
Delete
Toggle Grid
Measurement
Sync Pan/Zoom
Play
Pause
Diagram
Independent Variable
Setup
Signal Operators
Expression
Text
Max...
Min...
Arithmetic Mean...
Rectified Mean...
Root Mean Square (RMS)...
AC Coupled RMS...
Difference...
Moving Average...
First Harmonic...
Total Harmonic Distortion...
FFT...
Slew Rate...
controlBus.axisControlBus3.speed_ref
controlBus.axisControlBus3.speed
0.4
0.0
-0.4
-0.8
-1.2
-1.6
-2.0
-2.4
0.00
0.25
0.50
0.75
1.00



The signal operator functionality is also supported by scripting, using the built-in function `plotSignalOperator` (except **Difference**, **Moving Average**, **First Harmonic**, **Total Harmonic Distortion**, and **FFT**). For more information about `plotSignalOperator`, please see section "plotSignalOperator" on page 967.) For the scripting support of the exceptions:

- Concerning **Difference**, please see section "plotSignalDifference" on page 966.
- Concerning Moving Average, see section "plotMovingAverage" on page 965.
- Concerning **First Harmonic** and **Total Harmonic Distortion**, please see section "plotSignalOperatorHarmonic" on page 968.
- Concerning **FFT**, see Example 2 in that plotSignalOperator section (use the above reference).

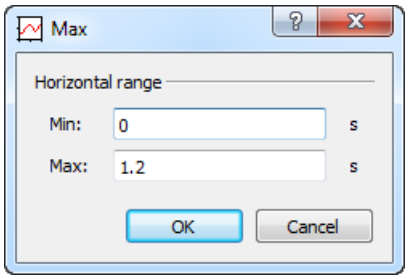
Signal operators for curves with other independent variables than time

You can apply all signal operators except First Harmonic, Total Harmonic Distortion, and FFT, on curves with other independent variables than time, if the independent variables are rising over time. If not, you will get an error message when trying to apply the signal operator.

The above also means that the built-in functions `plotSignalOperator` and `plotSignalDifference` are supported with the above exeptions and limitations.

Inserting signal operators

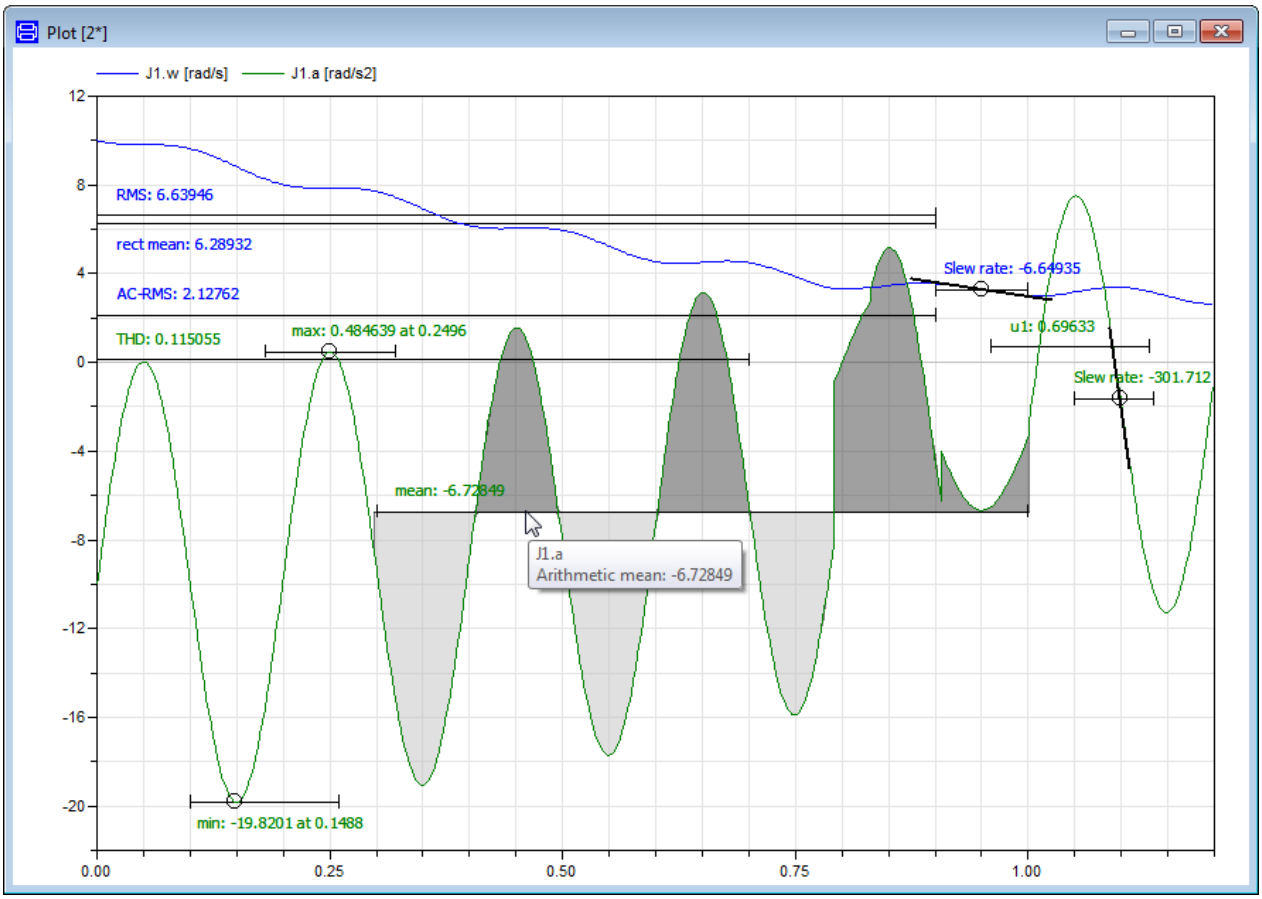
Selecting an operator launches a dialog where the user can define the time range over which the operator will be applied. The default time range is from the start time to the stop time of the simulation. For the operators **First Harmonic**, **Total Harmonic Distortion**, and **FFT** the period of the signal must be entered, and more selections are also available; see below. For **Moving Average**, the interval length must be specified, see below.



An error message is raised if the specific time range is out of bounds or if the minimum is larger than the maximum.

The **Appearance** settings can be used to make the labels more visisble when for example having several curves. See illustration below in the section about the context menu.

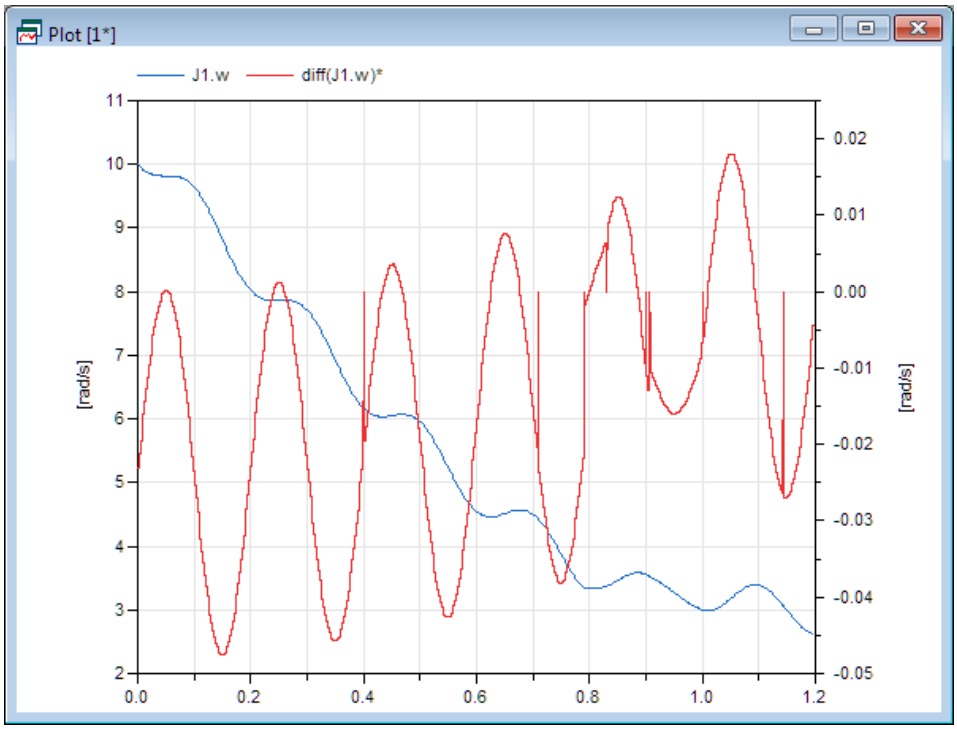
The result is then visualized in the plot window in different ways depending on which operator that was selected. The time range and the associated text are using black color by default. If the plot contains more than one curve, the text is written in the same color as the plotted signal. The picture below shows the appearance of most presently available operators (for Difference, Moving Average, and FFT, see below):



Common for all cases, except Difference, Moving Average, and FFT, is that the time range is represented with a horizontal line with clearly marked end points.

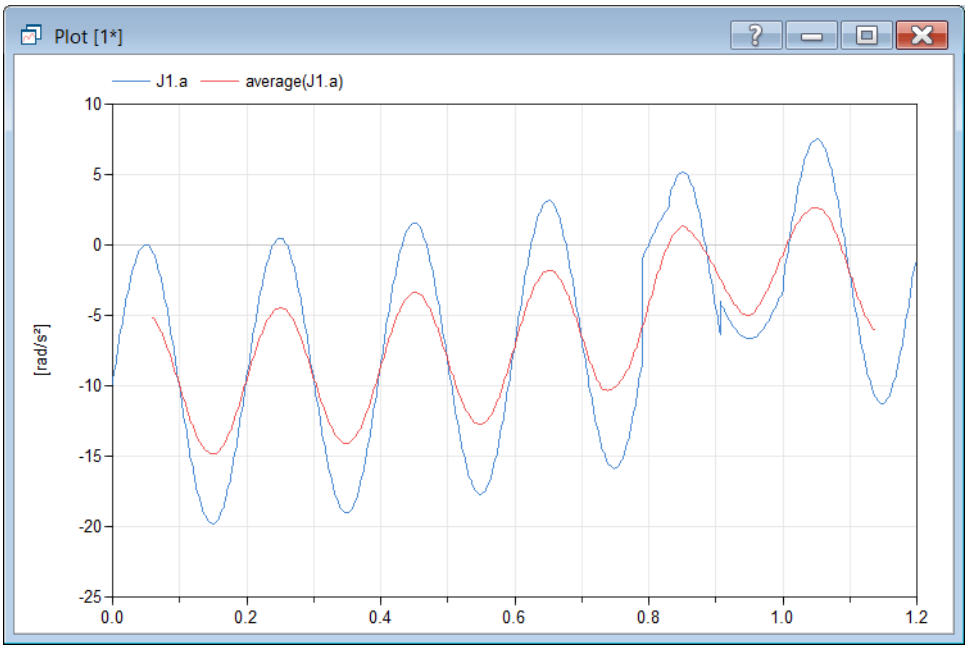
In the figure above also the tooltip of a signal operator is displayed.

For Difference a curve of the difference is added (below the signal operator applied to `J1.w` from the Coupled Clutches demo:

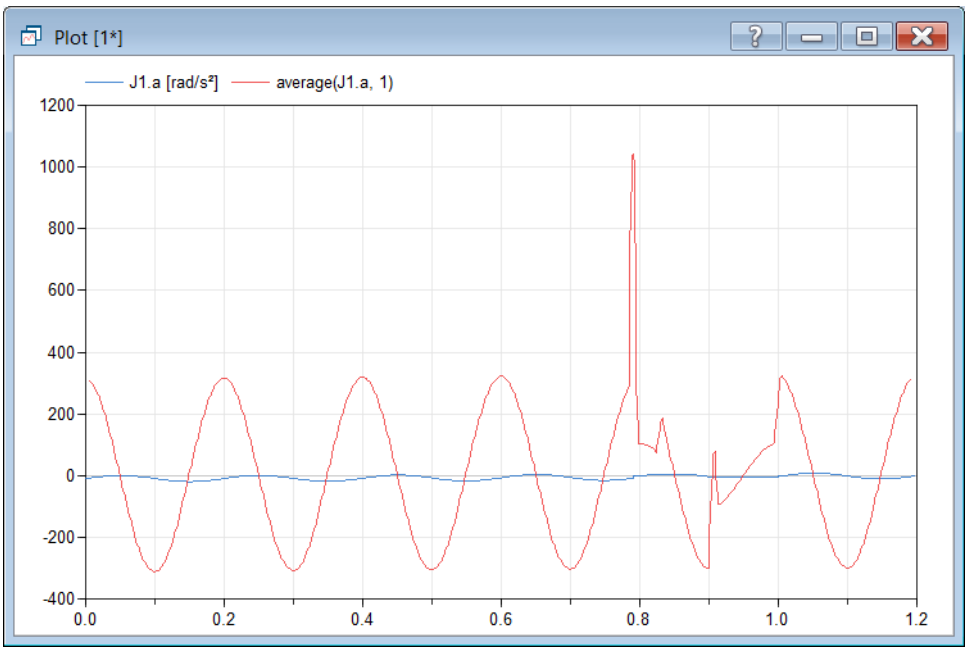


Note that the vertical axis for the difference curve is added to the right; this is also indicated by a star in the legend for that curve.

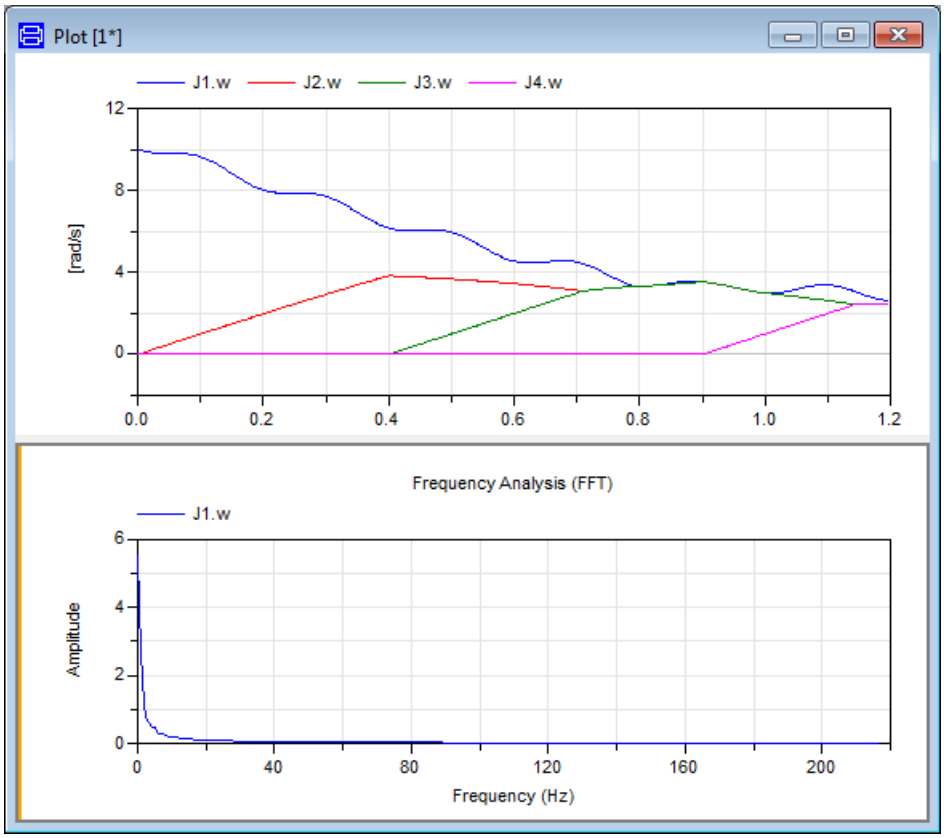
For Moving Average an extra curve is added, displaying the moving average. An example (with large interval length for clarity):



Note that by selecting **Differentiate signal** in the signal operator dialog, you can compute and display first-order derivatives. Doing this for the signal above, but without increasing the default interval length, gives (compare with the J1.a curve above):



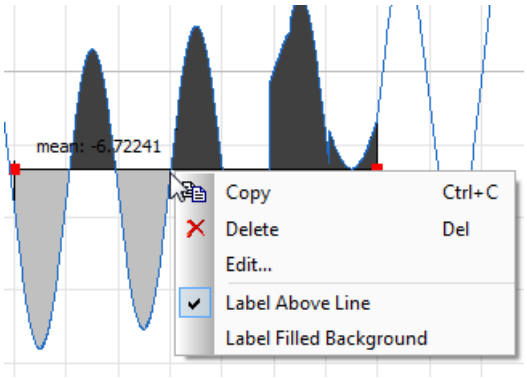
For FFT an extra diagram is created when using the operator:



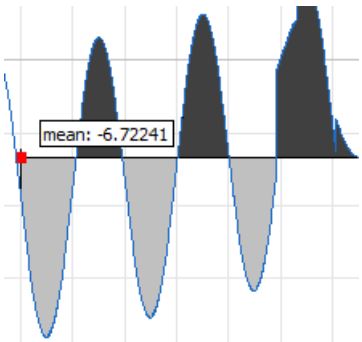
Editing signal operators

It is possible to interactively change the range of signal operators (exept difference and FFT) by drag-and-drop of the end points, or by dragging the whole line (keeping the range fixed). **Note** that the signal operator must first be selected, before the former performing operation.

Right-clicking the signal operator (not possible for difference and FFT) displays a context menu for editing:

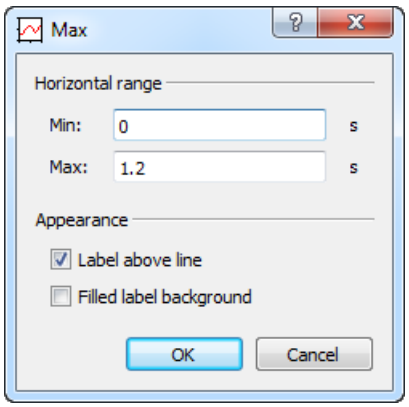


Note the effect of ticking **Label Filled Background** for this example (this setting is also available when creating a signal operator):



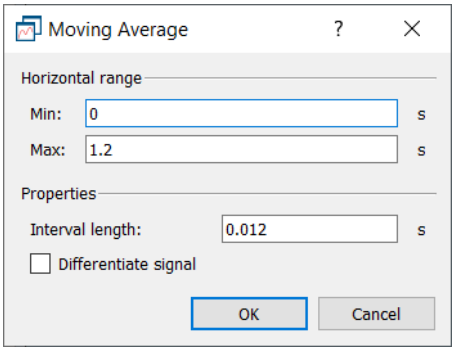
See “Context menu: Plot window – signal operators” on page 877 for more information about this menu.

It is also possible to double-click the signal operator. The same menu as when creating the singal operator is displayed, but an **Appearance** section is also added, to enable change the label location and if the label background should be filled (se above).



Additonal selections for Moving Average

Selecting **Signal Operators > Moving Average…** in the context menu of a curve will display:

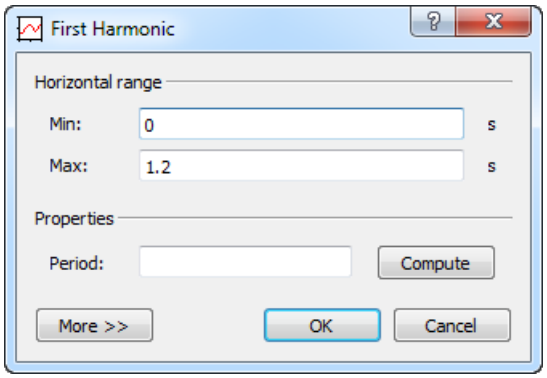


Except **Min** and **Max**, the **Interval lenght** also has to be entered; a default value is given.

By activating **Differentiate signal**, you activate computing and display of first-order derivatives, see example above.

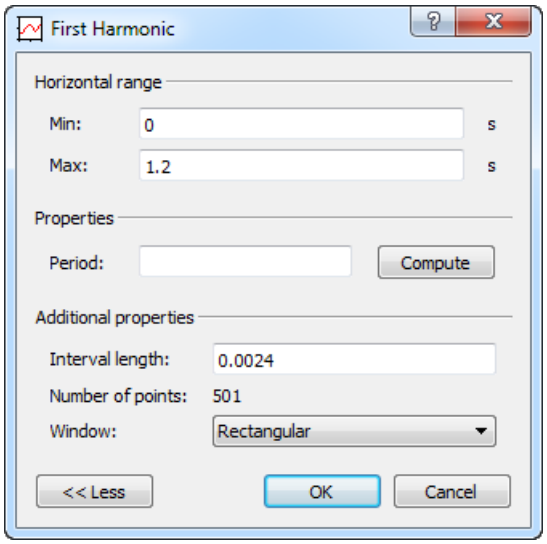
Additional selections for First Harmonic

Selecting **Signal Operators > First Harmonic…** in the context menu of a curve will display:



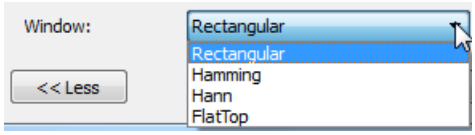
Except **Min** and **Max**, the **Period** also has to be entered; it can however be computed by clicking the **Compute** button.

Selecting **More >>** will expand the menu:



Here the **Interval length** can be selected; together with the **Horizontal range** this gives the **Number of points**.

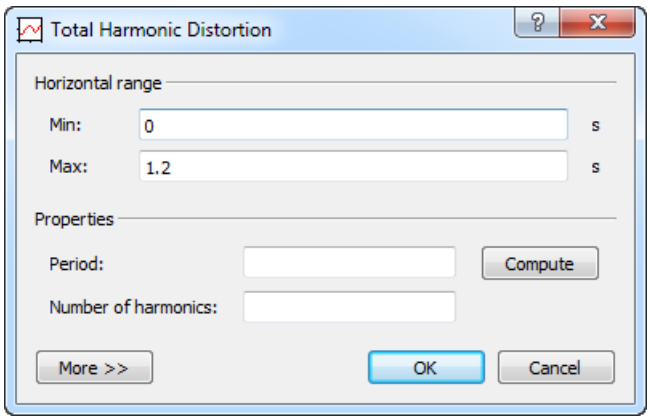
Window can be selected:



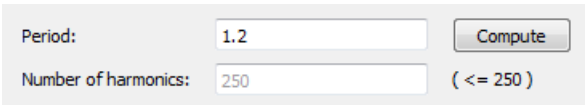
Default is **Rectangular** which gives equal weight to all points.

Additional selections for Total Harmonic Distortion

Selecting **Signal Operators > Total Harmonic Distortion...** in the context menu of a curve will display:



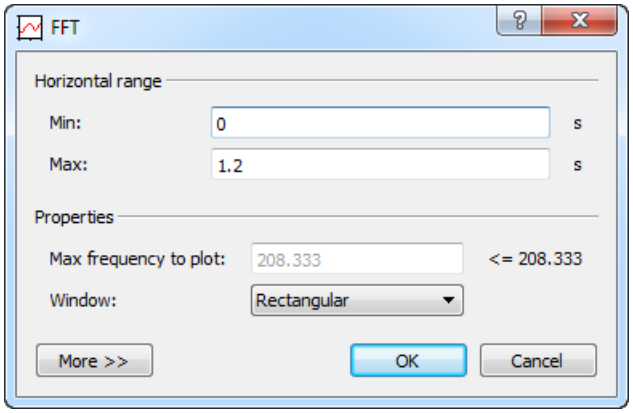
Comparing with the above command **First Harmonics**, the only difference is that for **Total Harmonic Distortion** also **Number of harmonics** can be selected. Entering or computing the period length will display limit for **Number of harmonics**:



The additional section displayed by selecting **More >>** is the same for both commands, see the previous section.

Additional selections for FFT

Selecting **Signal Operators > FFT…** in the context menu of a curve will display:



Except from **Min** and **Max**, the user can also select **Max frequency to plot**.

The **Advanced** is similar to the one for **First Harmonics** and **Total Harmonic Distortion**, except that the **Window** option is not in the **More >>** section, it is always displayed. The reason is that the **Window** option is more used for this operator.

Mathematical definitions of the signal operators

The following definitions are used for the signal operators (min and max are not listed here):

Arithmetic mean

This operator computes the arithmetic mean of the curve on the selected range. It is computed with the following formula (trapezoid approximation):

$$\text{Mean(a)} := \frac{\int_{\text{min}}^{\text{max}} a(x)dx}{\int_{\text{min}}^{\text{max}} dx} = \frac{\sum_{i=1}^{n-1}\left(\frac{a_i + a_{i+1}}{2}\right) * (x_{i+1} - x_i)}{\text{max-min}}$$

where n is the number of points of the curve in the selected range, and "a_i" is the value of each of these points.

Rectified mean

This operator computes the average rectified mean of the curve on the selected range. It is computed by taking the mean-value of the absolute values:

$$\mathrm{RMean(a)} := \frac{\int_{min}^{max} |a(x)|dx}{\int_{min}^{max} dx} = \frac{\sum_{i=1}^{n-1} f(a_i, a_{i+1})*(x_{i+1}-x_i)}{\text{max-min}}$$

The function f takes the average of the absolute values and if both values have the same sign it is just $\frac{|a_i|+|a_{i+1}|}{2}$; and if they have different sign: $\frac{a_i^2+a_{i+1}^2}{2|a_i-a_{i+1}|} = \frac{a_i^2+a_{i+1}^2}{2(|a_i|+|a_{i+1})|}$. The reason for the complex formula is to ensure that any piece-wise linear signal (including triangular waves) is correct.

RMS

In the case of a set of n values $\{x_1, x_2, x_3, \ldots, x_n\}$ representing the values of the point of the curve, the RMS (Root-mean-square) value is given by this formula:

$$\mathrm{RMS(a)} := \sqrt{\frac{\int_{min}^{max} a^2(x)dx}{\int_{min}^{max} dx}} = \sqrt{\frac{\sum_{i=1}^{n-1} \left(\frac{a_i^2 + a_{i+1}^2}{2}\right)*(x_{i+1}-x_i)}{\text{max-min}}}$$

Note that for the important case of equidistantly sampled sine-signal (with arbitrary phase) this formula gives the exact result (below the Nyquist frequency, and if the simulated interval is a multiple of the period of the sine-signal).

AC Coupled RMS

This property is related to the standard deviation in the statistics and can be computed in two different ways (the first one is used to avoid numerical issues with round-off and overflow):

$$\mathrm{ACRMS(a)} = \mathrm{RMS}\big(\text{a-Mean(a)}\big) = \sqrt{\text{RMS(a)}^2\text{-Mean(a)}^2}$$

Difference

The signal operator plots the discrete difference of a signal, defined as y(i)=u(i)-u(i-1).

Moving average

This signal operator is similar to the arithmetic mean signal operator above, comparing with the first formula for that signal operator above, to get the value at x2, we change “Mean(a)” to “Mean(a, x2)”, in the integrations we change “min” to “x2-w/2”, “max” to “x2+w/2”, and instead of integrating “a(x)”, we change to integrating “a(x)/w” where w is the width of the moving average “window”.

In general, the signal operator is implemented as a Centered Moving Average. If any endpoint is missing for the calculation, it is calculated by interpolation.

Frequency-based computations: harmonics

These are computed based on a windowed Fast Fourier Transform (FFT).

The underlying FFT is a generalized FFT that can handle any input length, not only powers of 2. Windowed FFT means that the signal is pre-multiplied by a windowing-function; this is customary in signal-processing and gives better result for signals that are not actually periodic (e.g. addition of random noise or a slight trend).

The FFT gives a decomposition of the signal as (computer efficiently):

$$x(t)\cdot \text{window}(t) = u_0 + \sum_{k=1} u_{s,k}\cdot \sin(\omega k t) + u_{c,k}\cdot \cos(\omega k t);\ u_k = \sqrt{u_{s,k}^2 + u_{c,k}^2}$$

The frequency-based computations are based on a number of equidistantly sampled points; the distance between the points should correspond to the Output interval length in the simulation setup.

The FFT can also be used to compute the period/frequency of the first harmonic; a simple formula is added here to take the period corresponding to the highest amplitude in the FFT (excluding the DC-component).

(Amplitude of) First harmonic

The amplitude of the first harmonic is the amplitude of the sine-signal of the given period (the given period should correspond to the base-frequency of the signal); in electrical applications, the frequency/period of the signal is normally known.

Note that the corresponding block in Modelica.Blocks has an output called rms which is the rms-value=amplitude/sqrt(2)). For a periodic signal, the amplitude of the first harmonic can exceed the amplitude of the underlying signal.

Total harmonic distortion

The THD used here is called amplitude ratio THD on Wikipedia (and measures how distorted the signal is compared to an ideal sine-signal):

$$\text{THD} = \frac{\sqrt{\sum_{i=2}^{\infty} u_i^2}}{u_1}$$

In practice the sum is always truncated due to the sampling, and to get comparable result independent of number of points stored the sum could be truncated before the sampling limit.

FFT

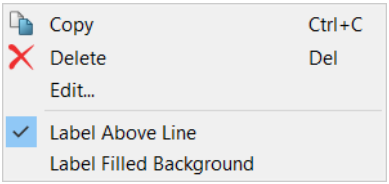
See frequency-based computations above.

Slew rate

The slew rate operator computes the maximum derivative value on the specified interval.

Context menu for signal operators

Right-clicking a signal operator, a context menu is available:



For more information, see “Context menu: Plot window – signal operators” on page 877.

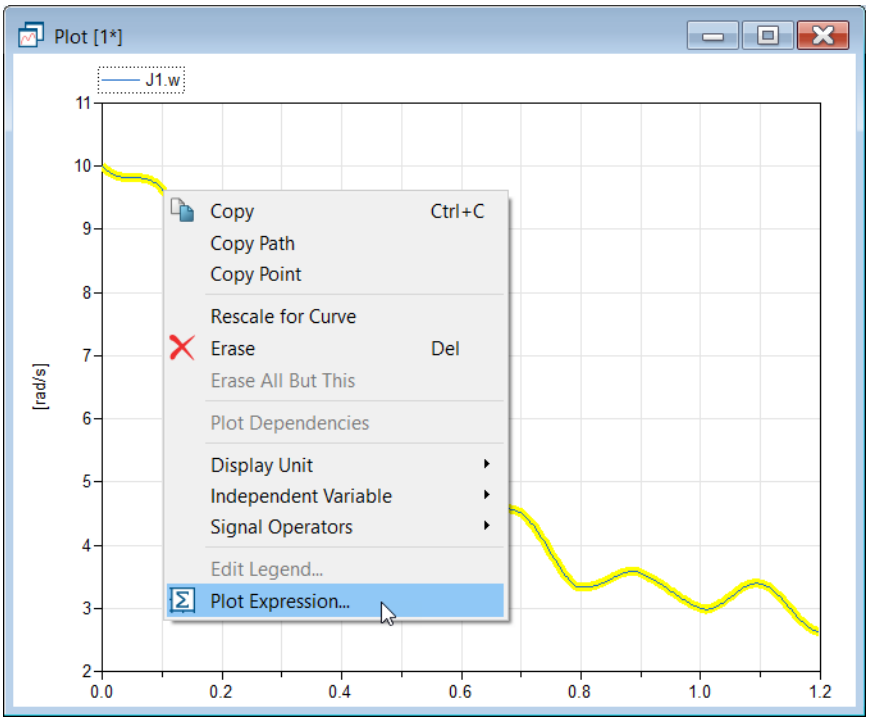
Plotting general expressions

Displaying the Plot Expression dialog



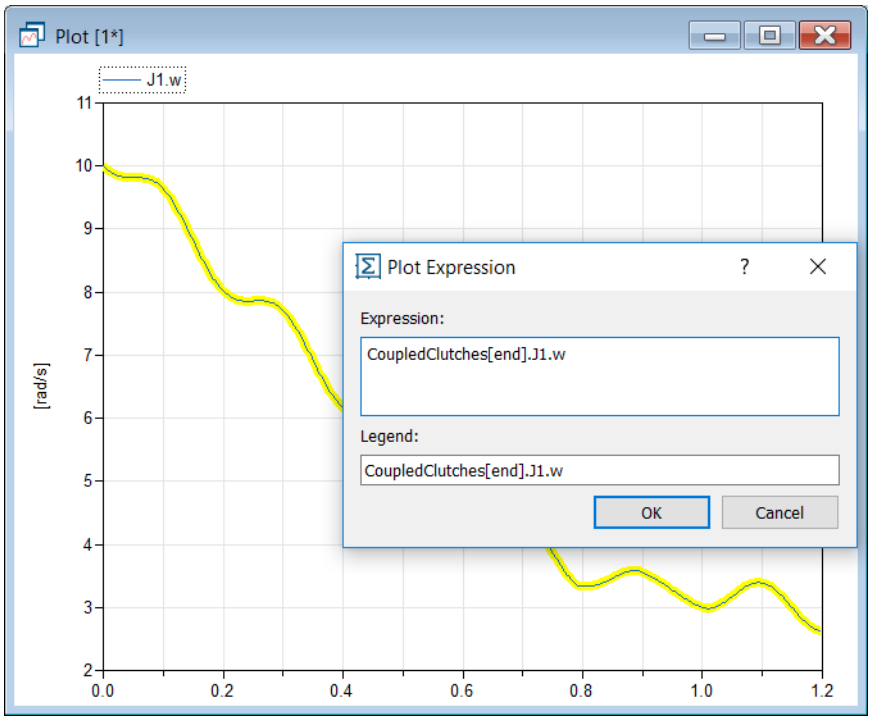
The functionality to create and plot a general expression can be invoked in a number of different ways:

- Through the context menu of a selected signal in the plot window as shown below:



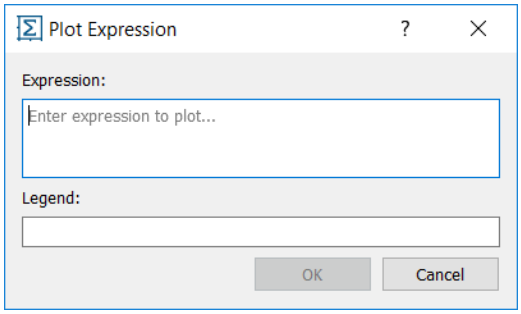
- Through the command **Plot: Options > Plot Expression**

If the first alternative above has been used, the dialog for creating a plot expression will be pre-populated with the selected signal:



Note that the name of the signal also contains the name of the result file and the sequence name (in brackets). The identifiers end and end-1 are generated for the latest and second latest results; in other cases the absolute sequence number of the result is generated.

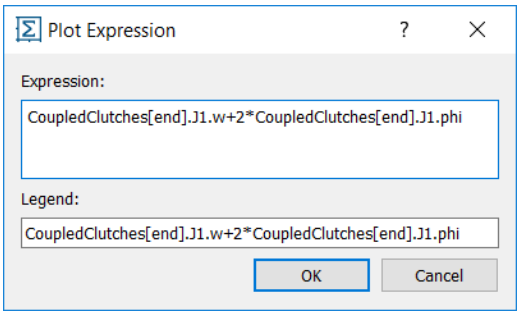
Using any of the other alternatives above, the Plot expression dialog will not contain any pre-populated signal:



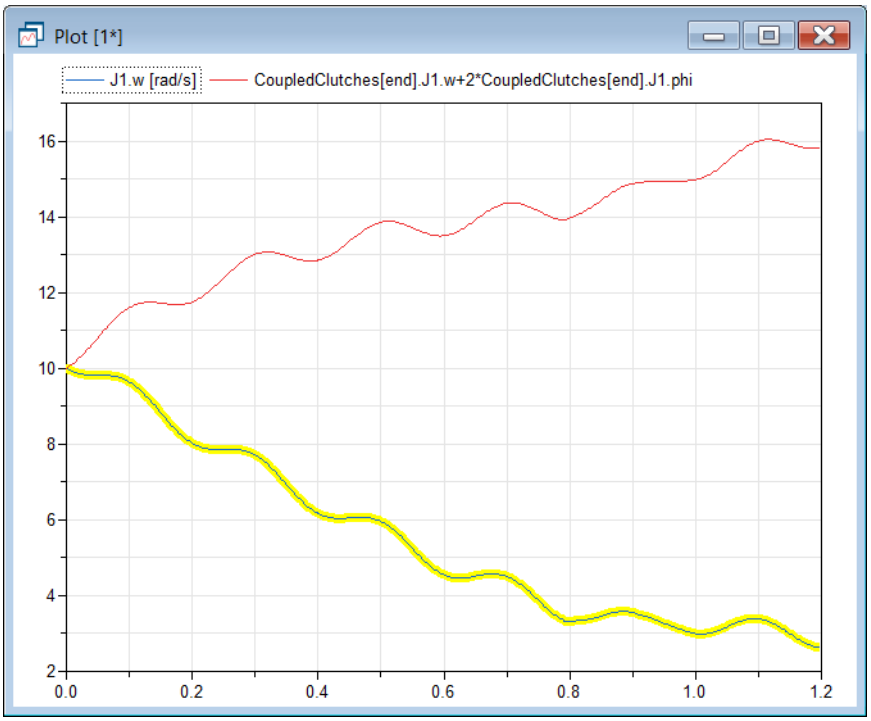
The functionality is also supported by scripting, using the built-in function `plotExpression`. Please see section “plotExpression” on page 964 for more information.

Editing plot expressions

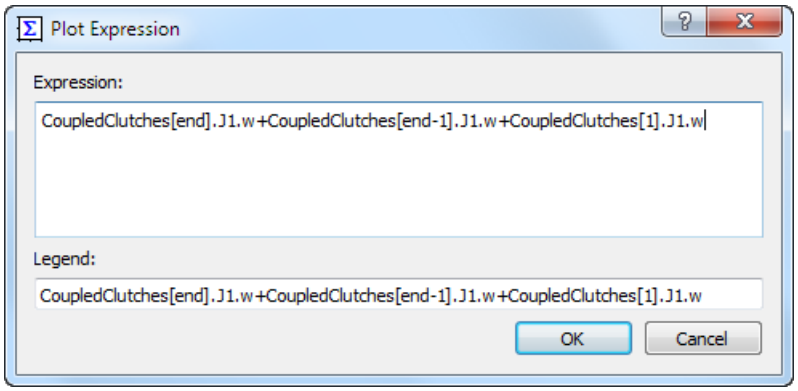
The expression in the dialog can be created by a combination of manual editing and by selecting signals from the variable browser. When a signal is selected in the variable browser, the name of the signal is automatically generated in the Plot expression dialog at the current position of the cursor. An example is shown below:



After pressing **OK**, the expression is parsed, and if valid, it is plotted in the active plot window. By default, the entered expression is also shown in the legend of the plot window. The legend can be changed by entering a different string in the **Legend** field of the Plot expression dialog.



It is also possible to combine signals from different result files when building the expression. As shown already above, the notation used to indicate the result file is `resultFile[resultID]`, see the example below:



It is possible to have unit/display unit on plot expressions by scripting, see “plotExpression” on page 964.

Recalculate plot expressions after simulation

It is possible to recalculate plot expressions curves automatically after simulation. This is done by activating the setting **Recalculate plot expressions after simulation** in the plot setup, reached by the command **Plot: Options > Setup**, the **Options** tab. See section “Options tab” on page 854.

Plot 1D discreatizations

This feature is available both from the variable browser and from the **Simulation > Plot: Options** tab. See “Plotting a 1D discretization” starting on page 631.

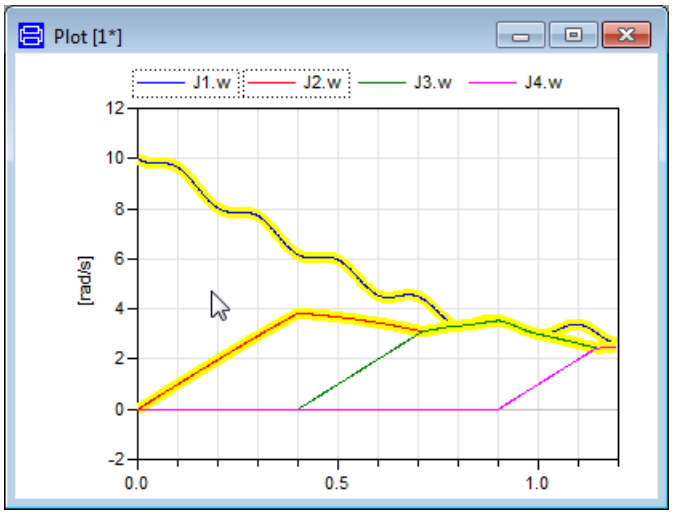
Plot Dependencies

This feature is available both from the variable browser and from the context menu of a curve/label. The feature is presented in “Plot dependencies” starting on page 633.

Selecting multiple curves in a diagram (for e.g. copying curve values to Excel)

It is possible to select more than one curve in a plot window. A curve is selected by clicking either on the curve or its legend. To deselect, the curve/legend can be clicked again.

Selected curves are marked with yellow highlight.



To select all curves in a diagram, right-click and select **Select All**. The command is also available, when a diagram is selected, as **Edit > Select All**. Note that only curves are selected, not for example texts inserted in the diagram.

To clear the selection, you can click the background.

If right-clicking a curve and selecting **Copy** from the context menu, all selected curves will be copied to a tab-separated list with one column per curve. The first column contains the x-value, normally time.

	A	B	C	D
1	Time	J1.w	J2.w	
2	0	10	0	
3	0.0024	9.976903915	0.023999976	
4	0.0048	9.955610275	0.047999769	
5	0.0072	9.936098099	0.071999133	
6	0.0096	9.91834259	0.095997863	
7	0.012	9.902309418	0.11999578	
8	0.0144	9.887956619	0.143992677	
9	0.0168	9.875232697	0.16798833	
10	0.0192	9.864061356	0.191982538	
11	0.0216	9.854390144	0.215975091	
12	0.024	9.846124649	0.239965752	

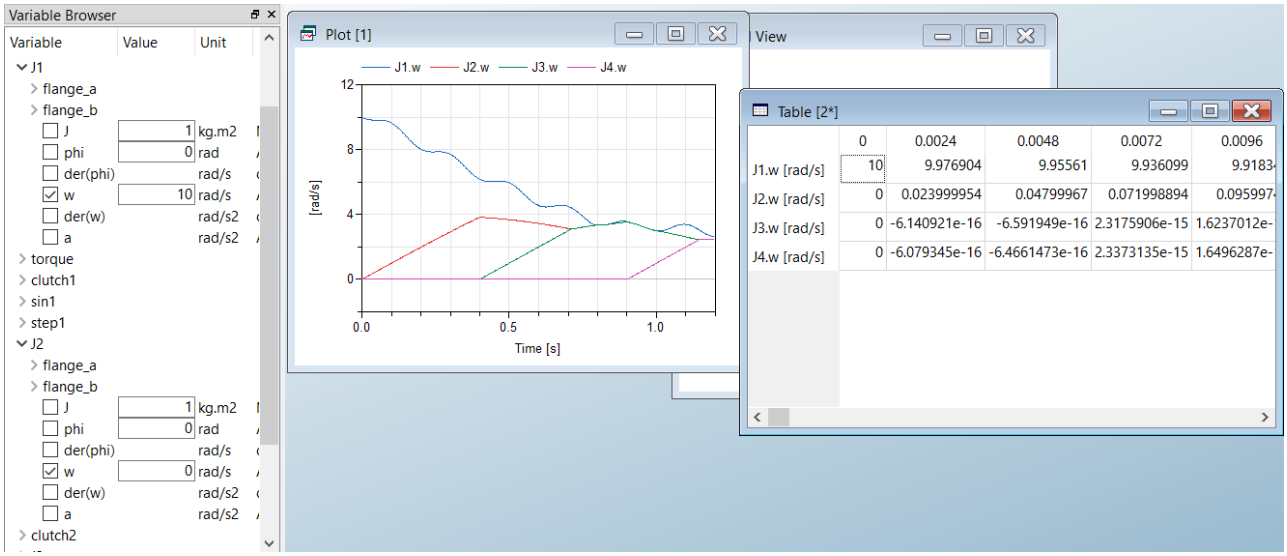
Displaying a table instead of curves

A plot can be displayed as a table instead of a curve. To create a plot table window, do any of the following, depending on which tab you are working in:

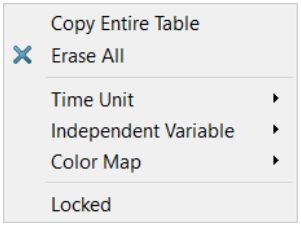
- On the **Simulation** tab, select the command **New Plot > New Table Window**.
- On the Plot tab, select the command **New Table**.

Signals to plot are selected from the variable browser in the usual way.

In the below example, a table has been displayed with the same signals selected as in the curve plot (this result can also be obtained by the next command, see below):

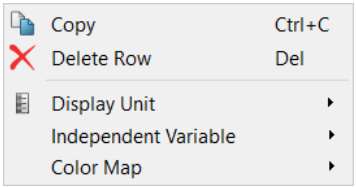


Right-clicking in the table, in an empty area, will display a context menu:



For more information about the alternatives, see section “Context menu: Table window” on page 879.

Right-clicking a value will display the context menu:



For more information about the alternatives, see section “Context menu: Table window - value” on page 879.

Creating tables from curves

The curves in a plot diagram can be copied to a new table window by the command **Generate > Table Window** in the **Plot: Options** tab.

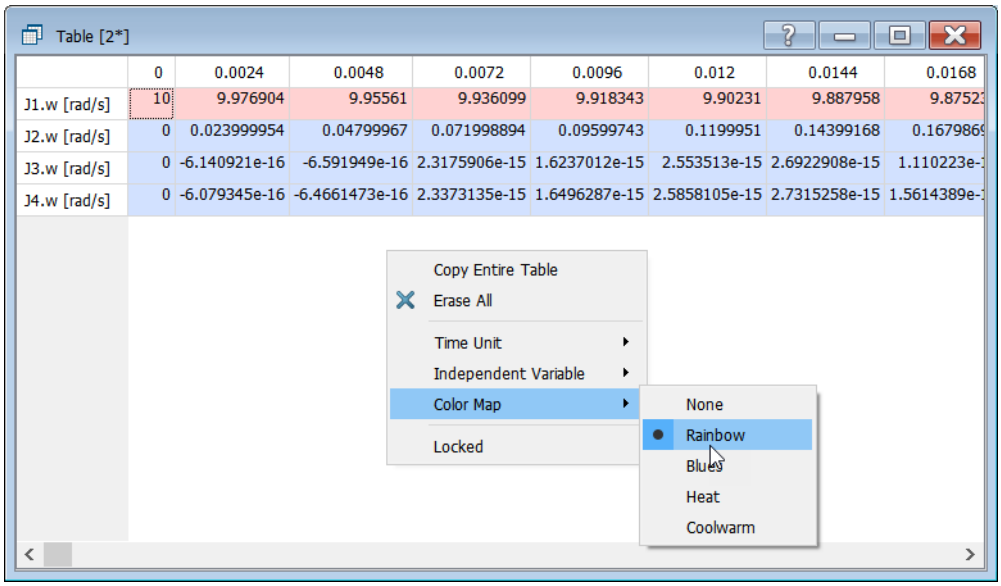
Please also compare the previous command.

Display constant arrays as matrices

This feature is available from the variable browser. See section “Displaying constant arrays as matrices” on page 630.

Applying color-coding to simulation result file tables

You can apply color-coding of table cells to simulation result file tables to indicate approximately the values. To activate the feature, right-click inside the table, in a cell, or in empty space like below, and then select **Color Map <theme>**. An example:



By default, this feature is not active (**None** is the default selection).

Note that color-coding of tables is currently only supported for tables consisting of curves from result files.

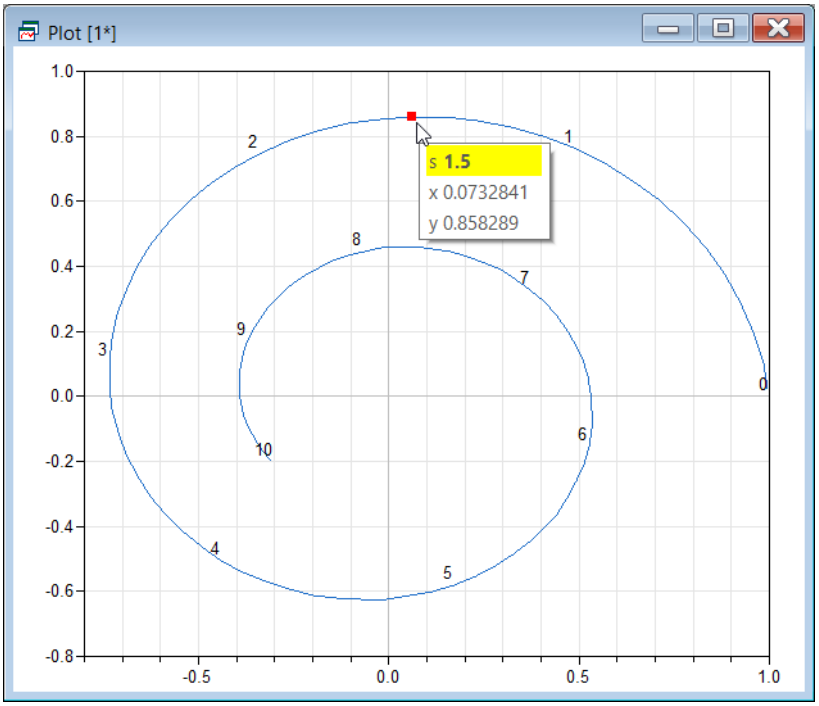
Displaying parametric curves

Curves defined by a parameter can be plotted using the built-in function `plotParametricCurve` or `plotParametricCurves`. For more information about these built-in functions, see section “plotParametricCurve” on page 965 and section “plotParametricCurves” on page 965.

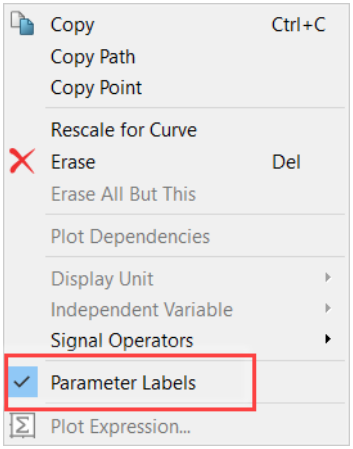
As an example of the first function, giving the following commands in the command input line of the commands window (or as a `.mos` script):

```
s=0:0.1:10
y={sin(t)*exp(-0.1*t) for t in s}
x={cos(t)*exp(-0.1*t) for t in s}
plotParametricCurve(x,y,s,labelWithS=true);
```

produces the following plot:



The labels in the plot can be turned on and off using the context menu command **Parameter Labels** of the curve:



See also section “Context menu: Plot window – curve and legend” on page 876.

Scatter plots

You can create scatter plots using the built-in function `plotScatter`. For more information about this function, and an example, see section “plotScatter” on page 965.

Changing color, line/marker style and thickness of a signal

Selecting the **Variables** tab in the plot setup menu enables the user to select a signal and change color, line/marker style and thickness of the signal. Custom colors can be used for the signals. For more information, please see the section “Plot: Options > Setup” starting on page 842, the **Variables** tab. This feature is also supported by scripting, by the built-in functions `plot()`, `plotArray()` and `plotArrays()`; please see these commands in section “Plot” on page 954.

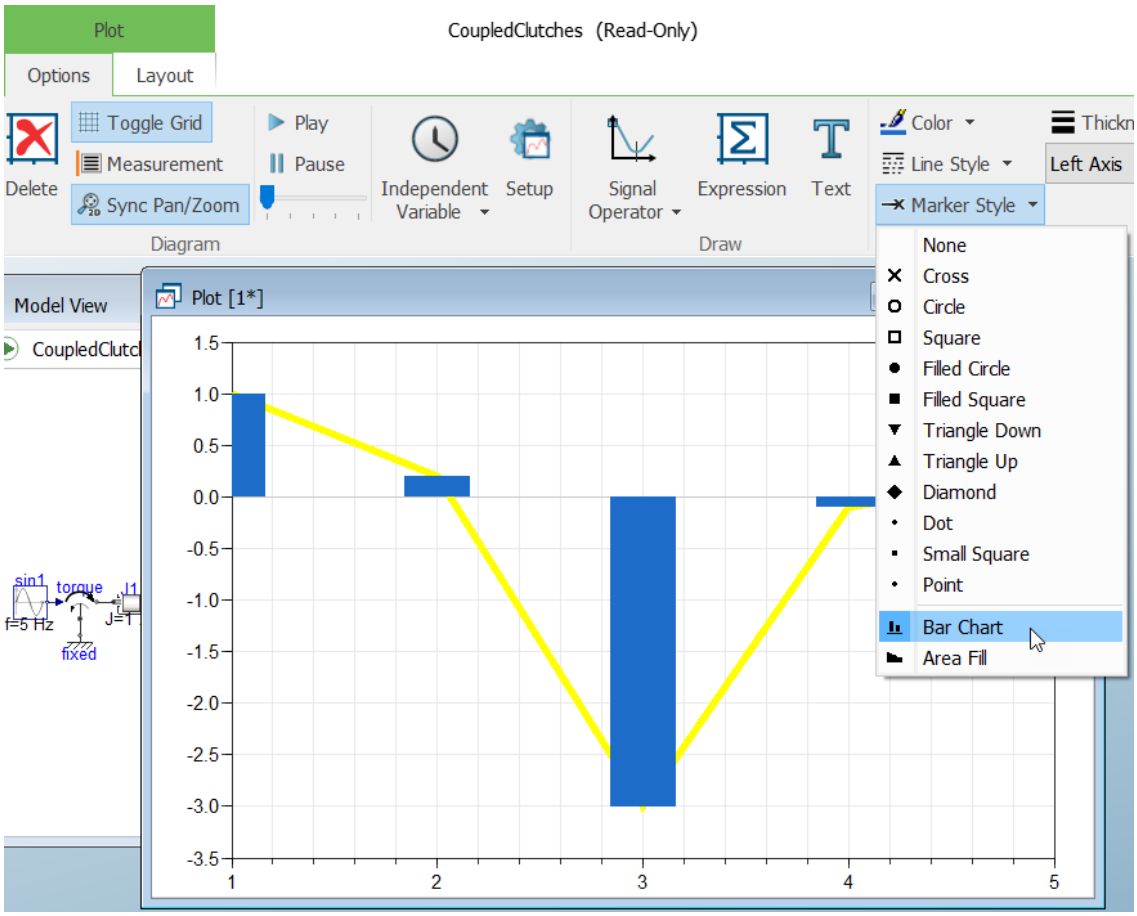
Plotting bar charts and area charts

You can plot bar charts and area charts by using the marker styles **Bar Chart** and **Area Fill**.

For an example of bar charts, enter, in the command input line of the **Simulation** tab, the plot command

```
plotArray({1,2,3,4,5}, {1,0.2,-3,-0.1,0.2})
```

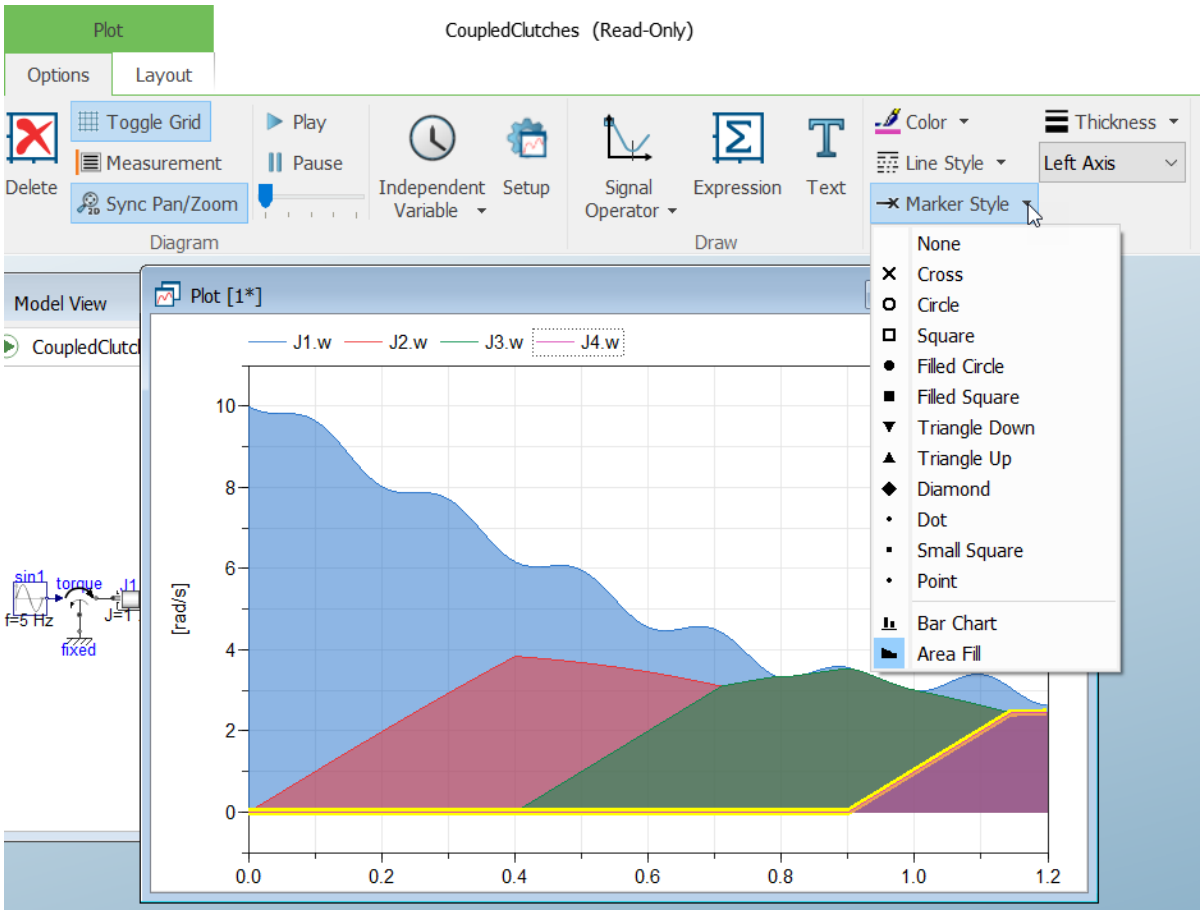
and then select the curve and apply **Marker Style > Bar Chart**:



Notes:

- The yellow representation of the original curve disappears by clicking outside it.
- To select the undelying curve, you must click any corresponding point, that is, at the center of any bar top.

For an example of area charts, the below is the demo Coupled Clutches with the curves selected and the marker style **Area Fill** applied on each of the curves:



Notes:

- The opacity can be controlled by the flag `Advanced.Plot.AreaFillOpacity`. The default value of the flag is `0.5`.
- Thickness and line style can be applied to the curves as well.

Plotting of bar charts and area charts are supported by scripting, the marker styles for bar chart and area fill are present as `MarkerStyle.BarChart` and `MarkerStyle.AreaFill`, in the same way as other marker styles.

Plotting of external files

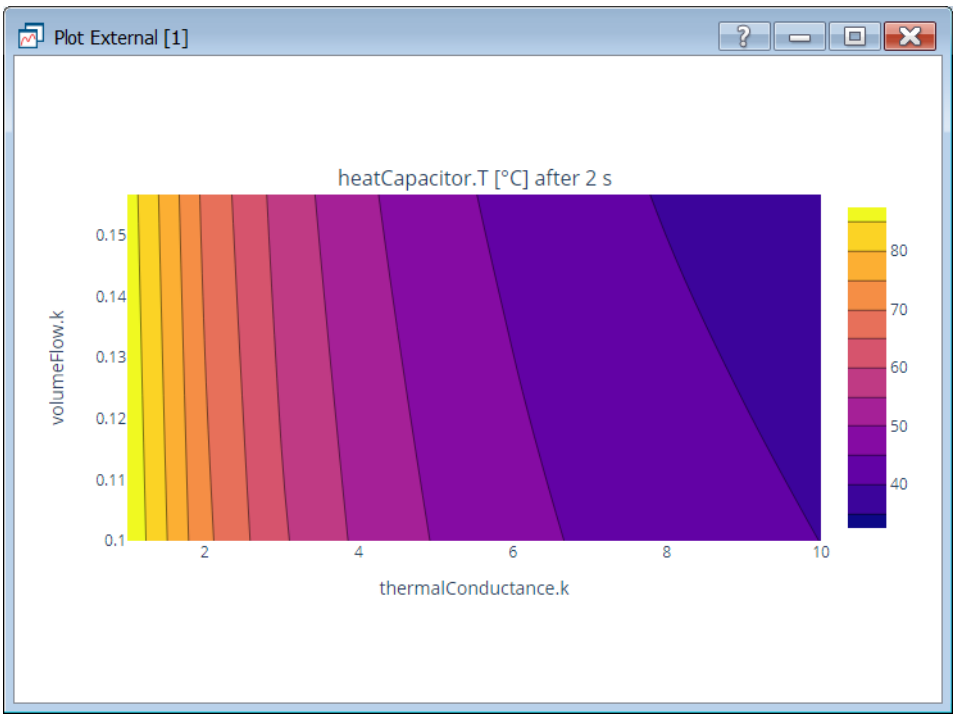
Dymola supports plotting of external files in the HTML, SVG, or PNG format. In addition, HTML plotting scripts generated in Python by, for example, matlabplot or plotly are supported.

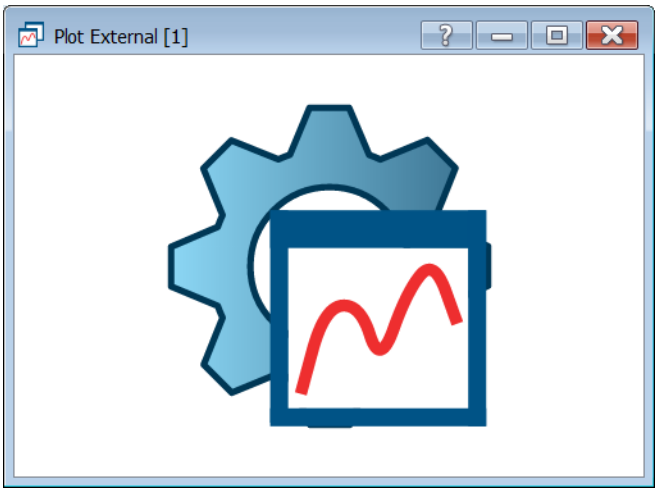
The files can be plotted by the built-in function `plotExternal`:

```
function plotExternal "Show external document or image"
```

```
   input String filename "Path or URL to HTML, SVG, or PNG
file";
   output Integer window_id;
end plotExternal;
```

Examples of plotted files, from a plot generated by plotly, and an SVG image:





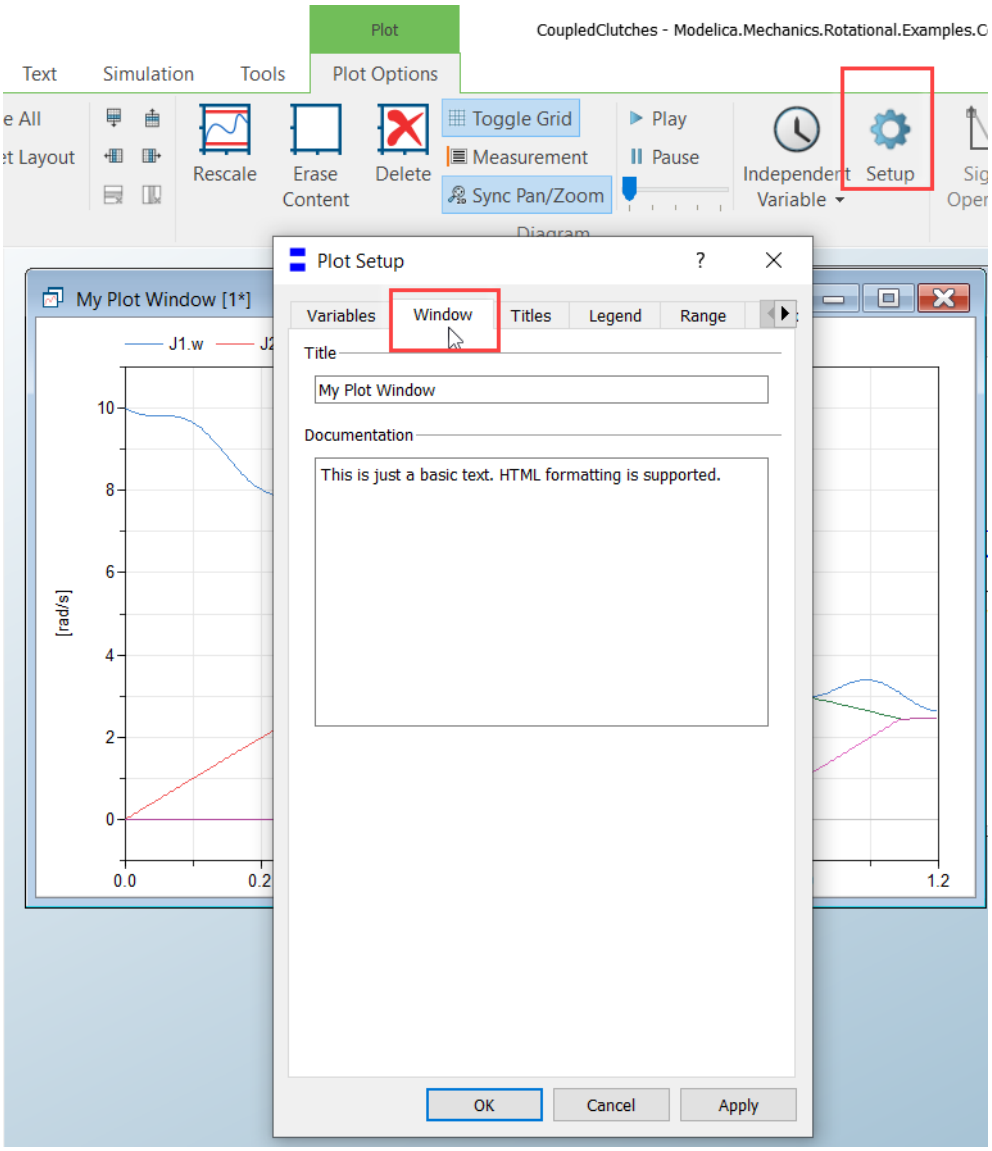
The result of the built-in function supports:

- The built-in functions `plotTitle` and `plotDocumentation`.
- Exporting an image by the command **Tools > Image** or the built-in function `ExportAsImage`.
- Copying an image to clipboard by, for example, the command **Tools > To Clipboard**.
- Printing the result.

Note that the argument `fileName` in the built-in function also supports URL´s, in addition to file paths.

Adding titles and documentation to plot windows

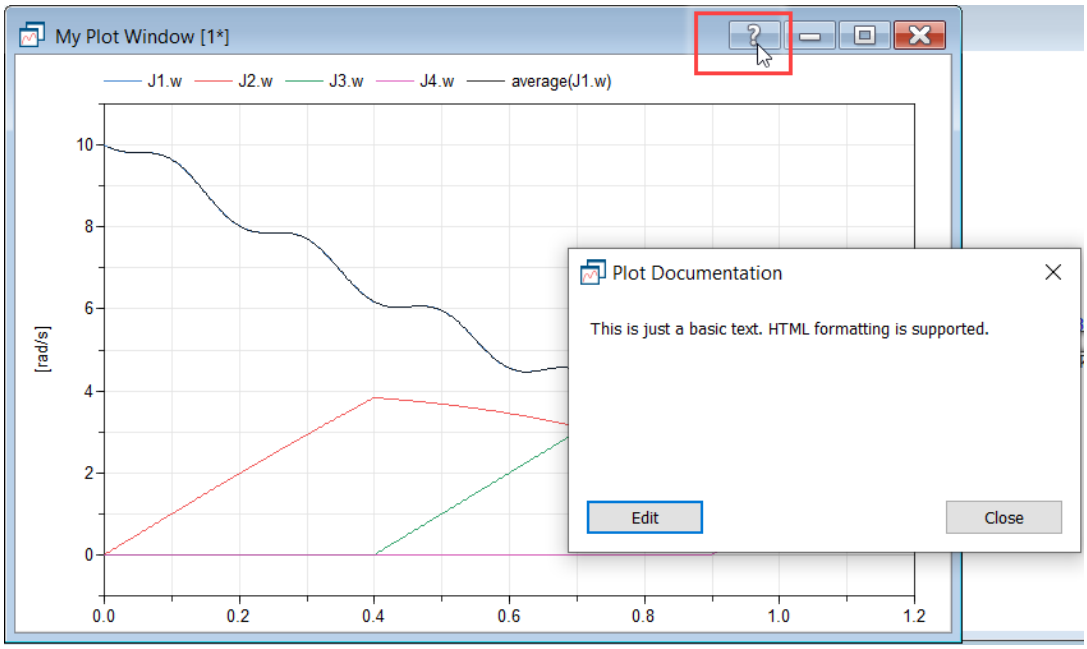
It is possible to add titles (text in the title bar), and documentation, to plot windows. To add title and documentation to the active plot window, use the command **Plot: Options > Setup**, and select the **Window** tab:



Now you can add a window title and documentation, as in the example above. Note that HTML formatting is supported for the documentation.

The plot window title is seen immediately after clicking **OK**, as in the plot window in the figure above.

To see the plot window documentation, click the question mark in the header of the plot window. You will see the following, for the example above:



Note that the **Edit** button, clicking this button will open the **Window** tab of the plot setup, as in the first figure in this section.

The window title and documentation features are also supported by scripting, see the sections plotTitle" on page 970 and "plotDocumentation" on page 963.

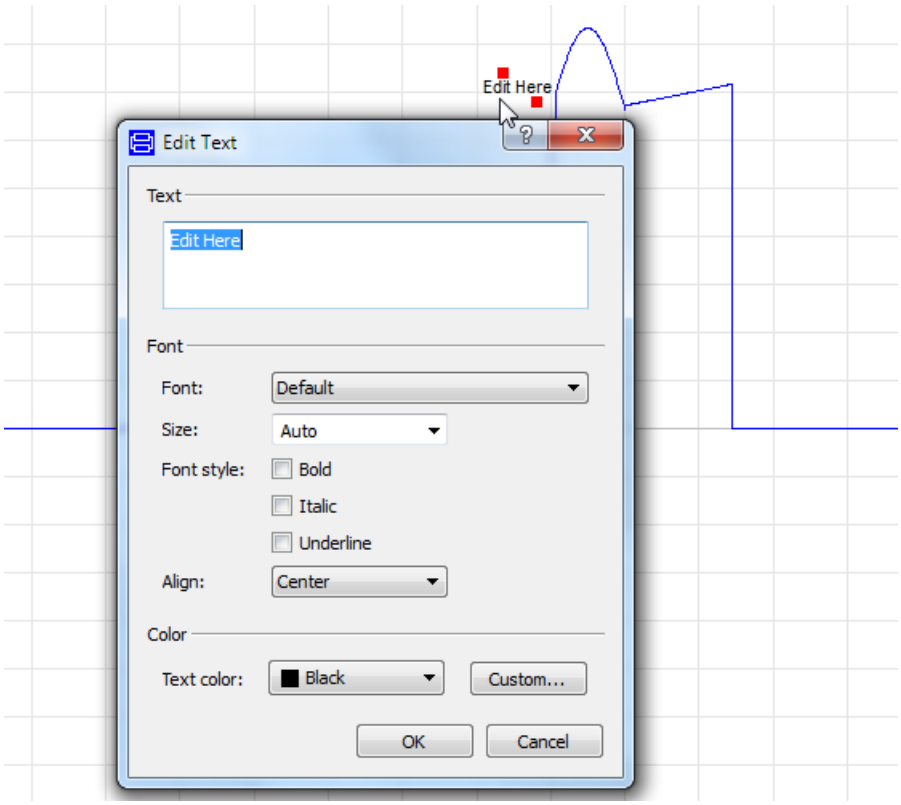
Set diagram heading and axis titles and ranges

Using the **Titles** tab and the **Range** tab in the plot setup menu enables the user to set diagram heading and axis titles and ranges of the plot. For more information, please see the section "Plot: Options > Setup" starting on page 842, the **Titles** tab and the **Range** tab.

Note that the diagram heading is treated as a text object, that is, it is possible to change font size, color alignment etc. The context menu is also the same. Please see below.

This feature is also supported by scripting, using the built-in function `plotHeading`. See section "plotHeading" on page 964.

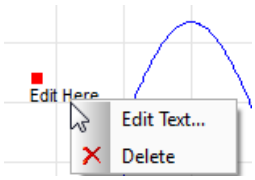
Insert and edit text objects in the plot



Text objects can be inserted in the active diagram in a plot window using the command **Plot: Options > Text**.

The handling of a text object is the same as for text objects in the diagram layer (except the default color being black and another context menu). This includes moving texts by dragging them and changing the extent by dragging the handles etc. Please see “Texts” in previous chapter, section “Basic model editing”, and sub-section “Creating graphical objects”.

A context menu is available by right-clicking on the text object:



Edit Text… will display the dialog box for text objects (like in the figure above). **Delete** will remove the text object from the plot. The context menu is also described in section “Context menu: Plot window – text objects” on page 878.

Inserting text objects in plots are also supported by scripting. Please see section “plotText” on page 969 for more information. Note that for plot headings another function is used, see previous section.

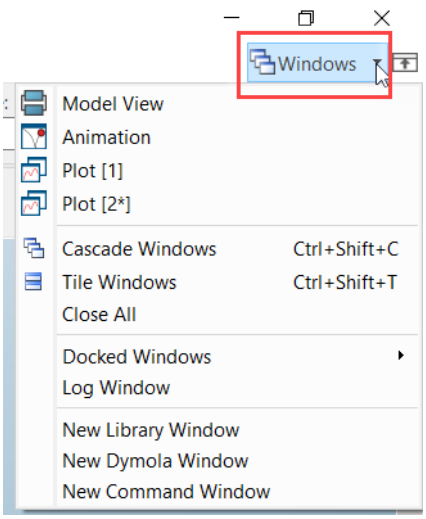
Changing the appearance, layout and location of the legend

Using the **Legend** tab of the plot setup menu, it is possible to change the appearance, layout and location of the legend. The legend can be located in several different places, for example inside the diagram. By default, the legend is drawn with transparent background not to hide curves in such a location. For more information, please see the section “Plot: Options > Setup” starting on page 842, the **Legend** tab.

Displaying the component where the selected signal comes from

The component that contains the variable corresponding to the selected signal can be displayed by right-clicking in the plot window and selecting **Show Component** in the context menu. The component will be shown selected in a diagram layer that will be displayed.

Going back to a previously displayed plot window



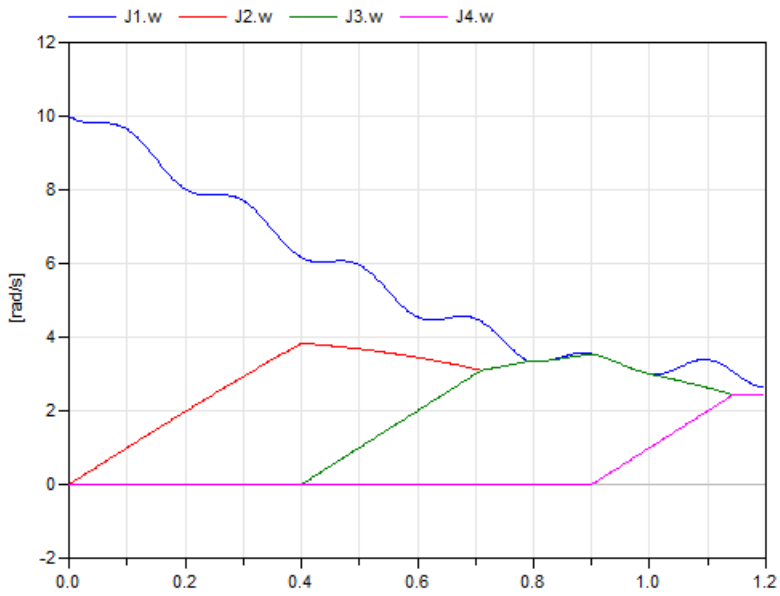
The button **Windows** in the top right of the Dymola window will toggle between the present sub-windows when clicking on it directly. Clicking the arrow in the button displays a menu with all sub-windows available (plot, animation, diagram layer, vizualiser). This button makes it easy to go back to any previously displayed sub-window. You can also use shortcuts, **Ctrl+Tab** activates the subwindow one step down in the list relative the currently active subwindow, **Ctrl+Shift+Tab** activates the subwindow one step up in the list relative the currently active subwindow.

Insert plot and its corresponding command in command window

The plot and corresponding command can be inserted in the command window by the command **Generate > Image in Log** in the **Plot: Options** tab.

An example of the result of this command:

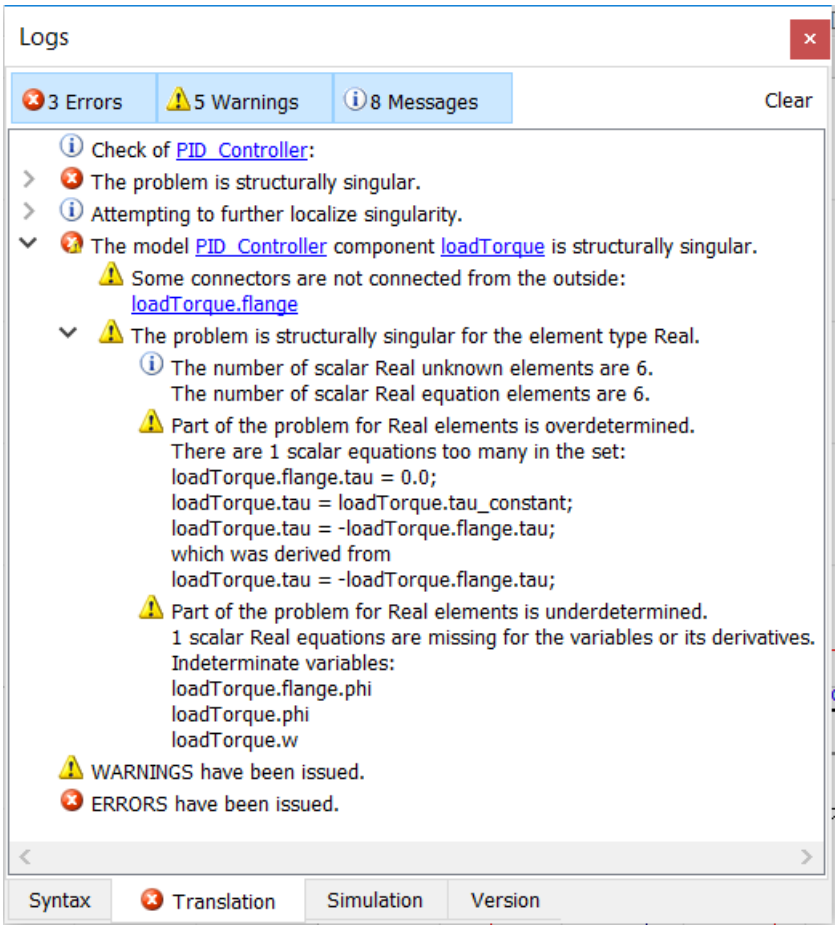
```
createPlot(id = 2,
 position = {698, 93, 536, 381},
 y = {"J1.w", "J2.w", "J3.w", "J4.w"},
 range = {0.0, 1.2000000000000002, -2.0, 12.0},
 autoscale = true,
 autoerase = true,
 autoreplot = true,
 description = false,
 grid = true,
 color = true,
 online = false,
 filename = "dsres.mat",
 leftTitleType = 1,
 bottomTitleType = 1,
 colors = {{0,0,255}, {255,0,0}, {0,128,0}, {255,0,255}});
```



5.2.6 Errors and warnings when translating

Any errors/warnings generated during translating or checking a model are displayed in the **Translation** tab in the log window. The information is described below, but see also “More about information in the Translation tab of the log window” on page 976.

Below is an example of translating an incorrect model:



In this section the usage of the **Translation** tab when having errors/warnings is described more in detail, for a general description of the log window and the contained tabs please see section “Log window” on page 603.

Displaying information

The Translation tab displays

- errors
- warnings
- information messages

The number of errors, warnings, and messages are always displayed in the top of the log window.

Note that the tab itself indicates if an error or warning is present in the log.

If parent icons contain messages of other type, subicons are presented in the parent icon, e.g.

-  - Error with warning submessages.
-  - Information messages with warning submessages.

This is useful for not missing submessages more severe than the parent message.

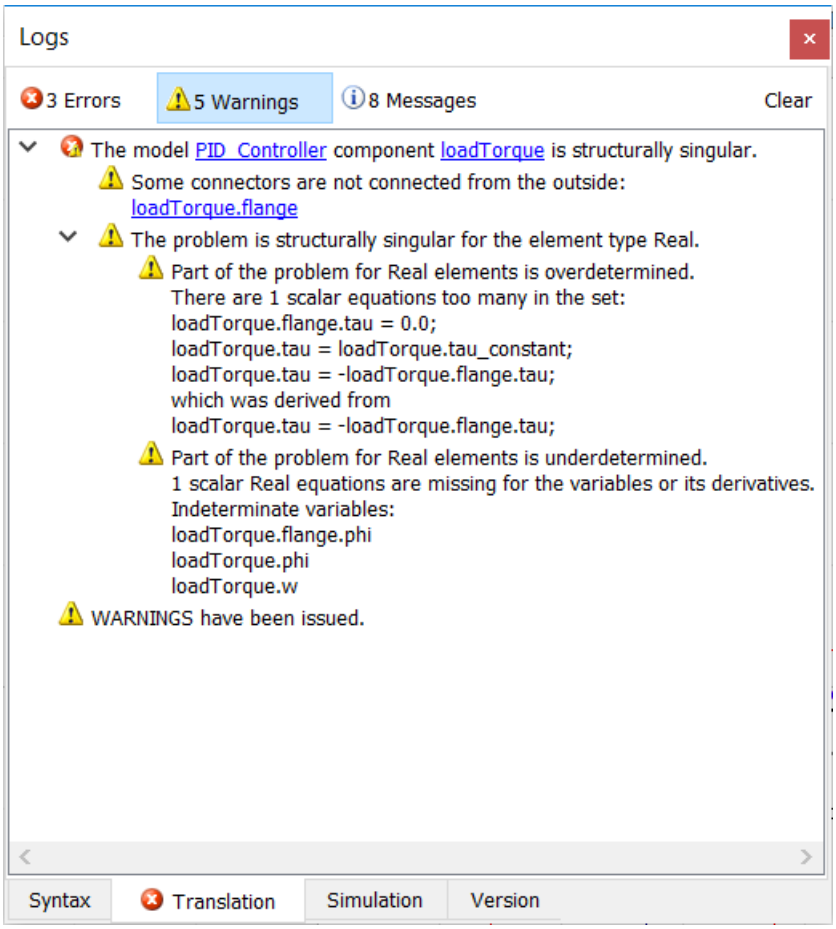
The messages are displayed in a hierarchical tree view, giving a good overview and facilitating detailed reading by expanding/collapsing message groups.

By default, the message groups containing errors or warnings are displayed expanded whereas information message groups are collapsed. You can however use the flag `Advanced.UI.AutoOpenLog` to control the number of open levels. The possible values of the flag are:

- 0 (default) – No expansion.
- 1 – First level of warnings and errors are expanded.
- 2 – All levels of warnings and errors are expanded.

The last alternative corresponds to right-clicking and selecting **Expand All** in the context menu. However, the flag value is kept during the session, while the result of the context command is only kept until next check.

Filtering can be applied using the three buttons in the header. The above image shows all messages being presented. By clicking on the buttons, the user can select what type of messages to display. The below shows only warnings (compare with pervious figure).



Note that the parent icon is also included since it contains warnings.

When moving the mouse over the window, message groups are highlighted with light-blue background. A selected message groups is displayed with a blue background.

The tab can be searched for text by right-clicking and using **Find…** in the context menu.

Linkage to components and classes

The messages may contain links to related components/classes. A tooltip will display what the link will display when clicking on it; e. g whether a component will be shown in the diagram layer …`#diagram:`*`componentname`* or a class will be shown in Modelica text. A component will be shown highlighted in the diagram layer. Note that clicking to such a link will change the Dymola mode to Modeling.

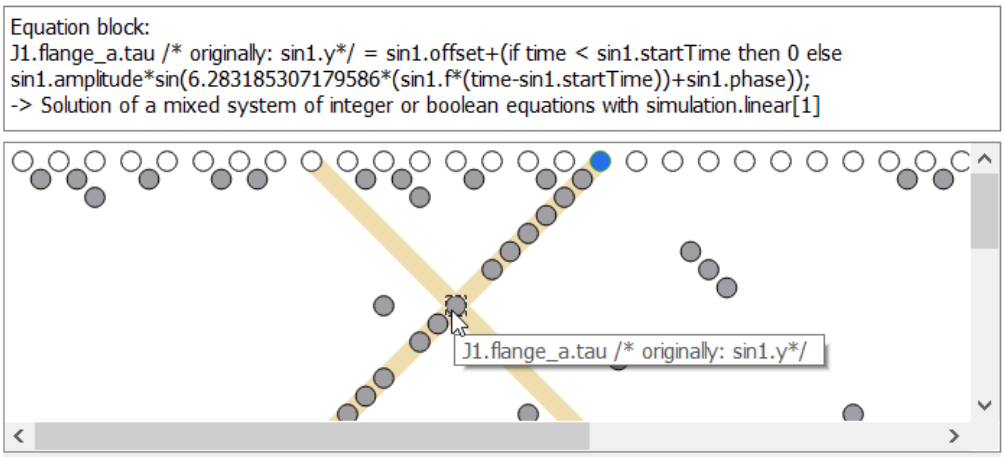
Handling of alias names of variables in error messages and other outputs

From Dymola 2023, you can use the flag `Advanced.Translation.KeepAliasInMessages` to decide what alias information to present. The value of the flag can be:

- 0: Use alias name in texts; this value corresponds to the alias name use in previous Dymola versions (before Dymola 2023).
- 1: Use original variable name in texts; this is the default value of the flag.
- 2: Use alias name in texts, but also present the original variable name as comment in the texts.
- 3: As "2", but use this also in the file `dsmodel.mof`.

The flag is used when creating error messages, information messages, logging, plot dependency information, and equation incidence texts.

As an example, consider an equation incidence window with the flag value 2:



In the equation block, and the tooltip, the alias name `J1.flange_a.tau` is used, and the original value `sin1.y` is presented in a comment.

5.2.7 Animation

Defining Graphical Objects for the animation window

Graphical 3D objects can be created by using one of the ready-made models in the `Modelica.Mechanics.Multibody` package. A number of shapes are supported by using the model `Modelica.Mechanics.Multibody.Parts.BodyShape`. That model support the shapes box, sphere, cylinder, cone, pipe, beam, gearwheel and spring.

However, the shapes are fixed. If corresponding shapes (and all other describing variables like color) should be dynamically variable, the model `Modelica.Mechanics.Multibody.Visualizers.Advanced.Shape` can be used.

External shapes are specified as DXF files (AutoCAD R12/LT2 format, only 3DFace is supported), or STL files (only ASCII STL files are supported). DXF files in libraries can be referred to using URIs, e. g. "`modelica://PackageName/file.dxf`".

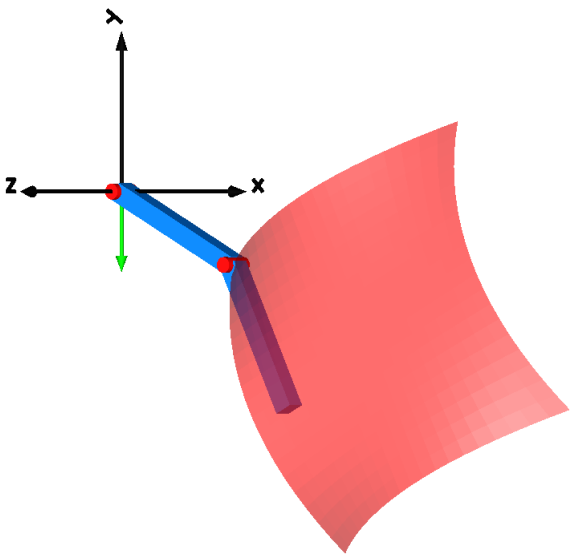
Since the DXF files contain color and dimensions for the individual faces the corresponding information in the model is currently ignored, but the specular coefficient is utilized.

The DXF files/STL files are found relative to the directory of the current model.

Please see the documentation for these models in Modelica Standard Library for more information.

If transparency and frame-input should be used, the visualization object `SurfaceFramed`, can be used. Data (except array sizes) can be changed dynamically during a simulation. It is presently not possible to export such animation to VRML, or apply it to other objects. The model is located in the folder `Program Files\Dymola 2023\Modelica`.

Example of surface-object with transparency.



Handling of result files in the variable browser

The handling of result files used for animation, for example, how to open one or several additional result files in the variable browser is described in section “Handling of simulation result files in the variable browser” starting on page 674.

Animation of one result file

If you open and simulate a model containing animation data, a simulation window containing the animation is opened automatically. If you already have simulation windows open, the

content in all these windows will be replaced by the animation data of the model you have opened.

(If the model you have opened does *not* contain any animation data, present animation windows will be blank when simulating that model.)

If you have several result files in the variable browser and you want to animate one, you can do that by using the variable browser context command **Animate** for that result file. Using this command, that result file is animated in the currently active animation window. If no animation window is present, you must first open one with the command **Simulation > New Animation**.

Animation of several result files together

By using the variable browser context command **Animate Together** several result files can be animated together in the same window. As an example, if you want to animate two certain result files in the active animation window, right-click the first result file in the variable browser and select **Animate** (see above for more information about this command), then right-click the second result file in the variable browser and select **Animate Together**. Now these two result files will be animated together in the animation window. You can add more result files if you want to, using **Animate Together**.

Exporting animations

The commands **Tools > [Export] Image** and **Tools > [Export] Animation** provides export of the animation window in several file formats. The latter command is also available as **Animation > Export Animation**. For the export image command, see index entry "*tools tab : [export] image*" in the index, for the export animation command, see "Animation > Export Animation" on page 866, respectively.

Saving animation windows settings

The animation windows settings can be saved by two different commands.

- You can use the command **Simulation > Commands > Add Command...**, ticking **Animation window setup** to save the animation windows settings and create a command in the model that retrieves this setting. You can also tick **Automatically run after simulating** to have the command executed after simulation of the model. See section "Simulation > Commands > Add Command" starting on page 793 for more information.
- Alternatively, you can use the command **Script Editor > Generate Script**, ticking **Animation setup**. See section "Script Editor > Generate Script" on page 833 for more information.

There are some differences between these commands, if you have specific requirements, please compare them in detail.

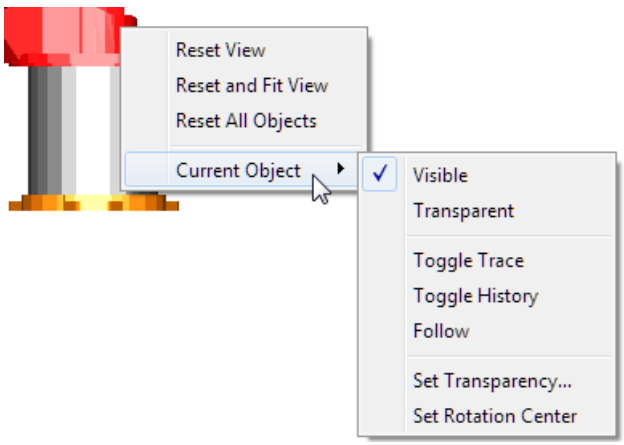
5.2.8 Animation window interaction

Manipulating the view

Using the context menu in the animation window

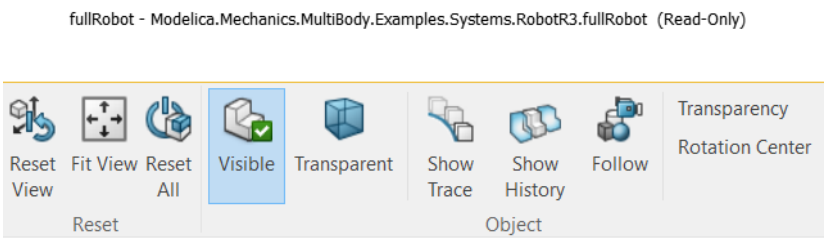
By selecting an object in the animation window (the object will be marked with red color) and then right-click, the following context menu can be used:

The context menu of the animation window.



More information about this menu is given in section “Context menu: Animation window” on page 880.

Note that the commands in the context menu is also available in the **Animation** tab:



For more information about the Animation tab, see section “Main window: Animation tab” starting on page 860.

Using the mouse and Meta keys

Direct manipulation of the view in the animation window using the mouse has been implemented. The view can be moved, rotated and zoomed using mouse movements in combination with Meta keys:

Operation	Meta key	Mouse move (dragging)	Arrow keys
Moving up/down/left/right	none	Up/Down/Left/Right	Up/Down/Left/Right
Tilt (Rotate around x-axis)	Ctrl	Up/Down	Up/Down
Pan (Rotate around y-axis)	Ctrl	Left/Right	Left/Right
Roll (rotate around z-axis)	Ctrl+Shift	Clockwise/Counter-clockwise	Left/Right
Zoom in/out	Shift	Up/Down	Up/Down
Zoom in/out	none	Wheel	
Zoom in/out	Ctrl	Wheel	
Zoom in/out	Ctrl	Right mouse button Up/Down	

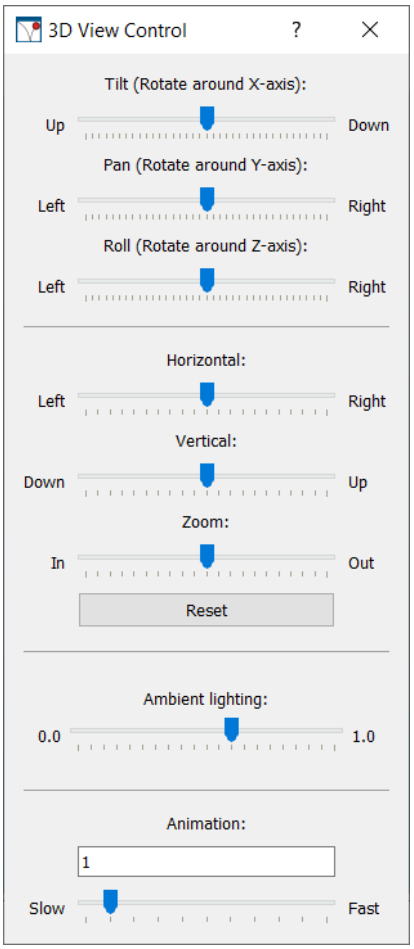
To set the rotation center on a selected component, use the context menu of the object and select **Current Object > Set Rotation Center**.

The arrow keys pan and tilt in fixed increments of 5 degrees, in addition **page up**/**page down** tilt 45 degrees. The **Home** key resets viewing transformation.

Using the command Animation > 3D View Control



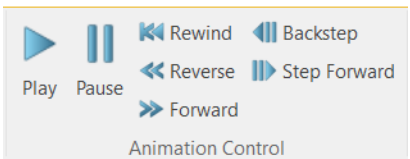
Using the command **Animation > 3D View Control** the following menu will pop up.

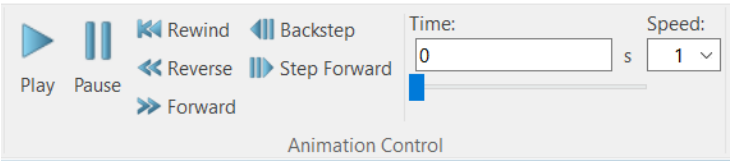


For information about the menu, please see section “Animation > 3D View Control” on page 866.

Running the simulation

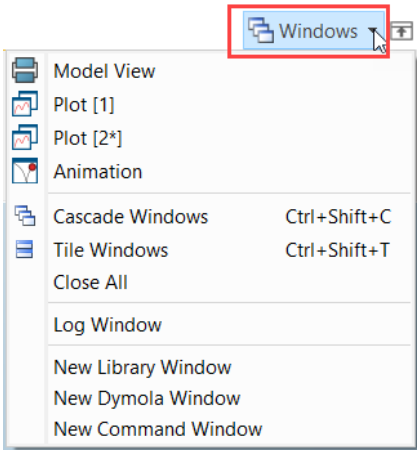
You can run the animation from the **Simulation** tab or from the **Animation** tab, both contains commands for running the animation. Note that the **Simulation** tab (second image below) contains some more commands:





For more information about these commands, see, when it comes to the **Simulation** tab, section "Main window: Simulation tab" starting on page 790.

Going back to a previously displayed animation window

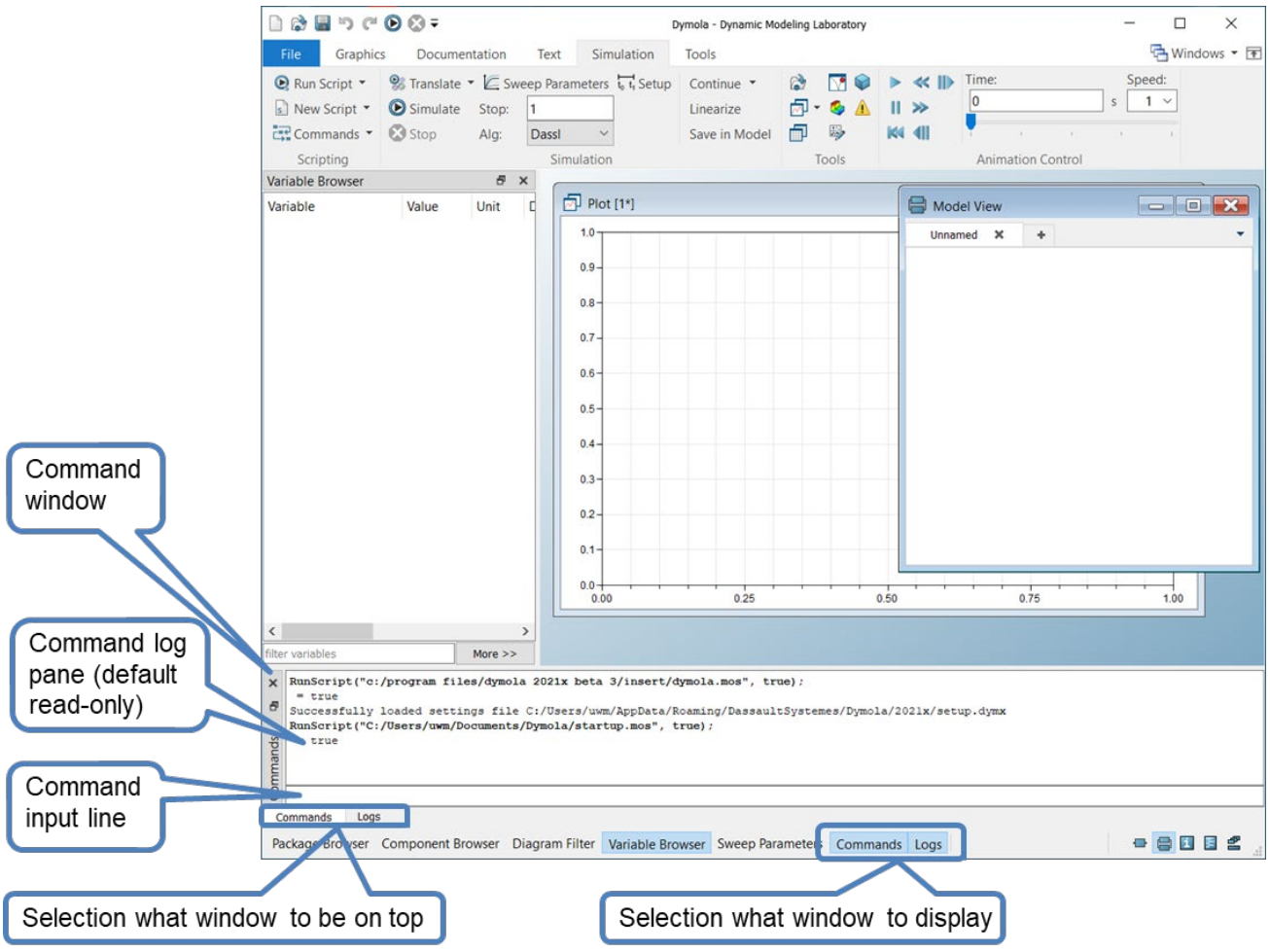


The button **Windows** in the top right of the Dymola window will toggle between the present sub-windows when clicking on it directly. Clicking the arrow in the button displays a menu with all sub-windows available (plot, animation, diagram layer, vizualiser). This button makes it easy to go back to any previously displayed sub-window. You can also use shortcuts, **Ctrl+Tab** activates the subwindow one step down in the list relative the currently active subwindow, **Ctrl+Shift+Tab** activates the subwindow one step up in the list relative the currently active subwindow.

5.2.9 Using the command window

Displaying the command window

When selecting the **Simulation** tab, the Command window is shown by default:



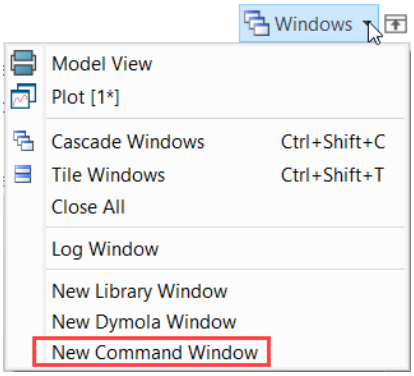
However, often when simulating a model, some log is shown in the Logs window instead; by default the Commands window and Logs window are docked, and the active one is on top. To select the Commands window on top in such a case, click the **Commands** tab just above the bottom line in the status bar at the bottom of the window (see “Selection what window to be on top” in the figure above).

Important! To be able to select like the above, the Commands window must be selected in the lowest line in the status bar (see “Selection what window to display” in the figure above).

The command window is also possible to display as a stand-alone window. One way is by undocking the present command window by using the **Undock** button or by double-clicking on the window header or dragging it (in this case the header is the Commands bar to the left in the sub-window.)

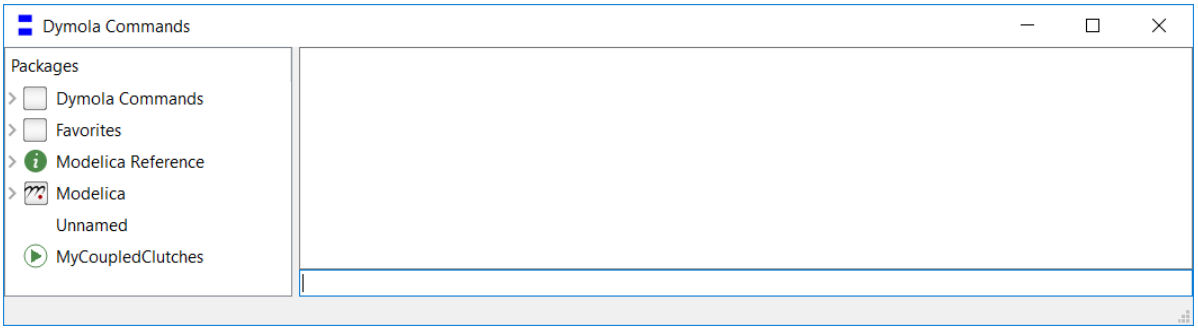
To dock the window again, the easiest way in to double-click on the window header. It can also be dragged in (you will then see how space for it is created automatically before releasing it). When dragging it, please drag it so it enters the main window from bottom right.

Another way is to create a new stand-alone command window by using the command **Windows > New Command Window**.



(It is also possible to undock it if docked to the bottom of the window like in the first figure in this section.)

In that case, it will look like:



A new stand-alone command window will always be empty; it will not display the present log. However, new commands will be displayed.

The stand-alone command window also displays a package browser. This can be used when functions should be tested/investigated. When the function has been located, right-clicking on it in the package browser displays a menu where input values can be entered using the **Call Function…** entry. When clicking **Execute** the result is shown in the log window.

The command log pane

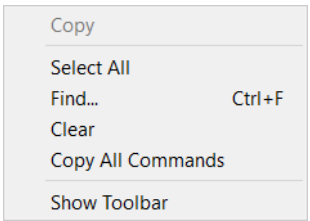
The command log pane can be used in two modes.

Default mode (read-only command log pane)

By default, the command window shows the command log in a read-only mode (as in the first image in this section). The command log contains commands given and the result of these commands. (The commands can be activated by command buttons, scripts or by typing in command in the command input line.) A valuable advantage is that the command log can be saved as e.g. a simple list of commands given (to be used as a script file). This is possible by using the different options for saving the command log (see next section below).

Note that the option to insert images of plots and animations are present also in this mode – they are seen as a result of commands. For more information, see the sections "Plots automatically inserted in command log" on page 774 and "Animation window content automatically inserted in command log" on page 775.

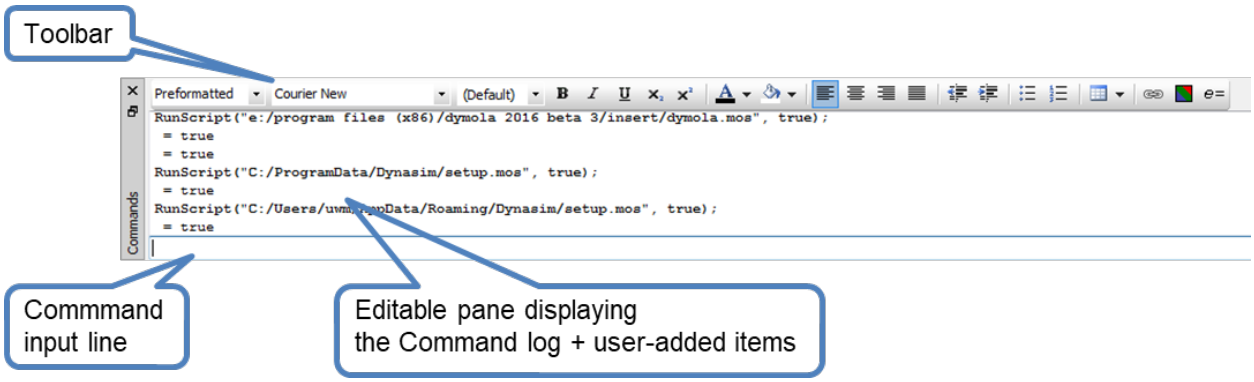
A context menu is available right-clicking the command log in default mode:



This context menu is a minor version of the context menu for the command log pane in documentation mode. Note the command **Show Toolbar**, that command controls if the command log pane in the command window should be in documentation mode or not. For more information about the context menus, see "Context menu: Command window – Command log pane" on page 882.

Documentation mode (editable command log pane)

Right-clicking the command log pane and activating **Show Toolbar** turns the command log pane into an editor that can be used to document the simulations. The command log is still present, but in an editable form. You can add texts, links, images etc. using the commands in the toolbar:



In this mode, the command log pane reflects looking at the command log as a documentation resource, enabling the creation of documentation of simulations containing also formatted descriptions of e.g. simulation results, plots, images, links etc – much more a documentation of a simulation than a pure command log. Math rendering of equations and formulas (with indexing) and rendering of Greek letters is also possible. Of course, it is possible to copy sections of this editor to e.g. Microsoft Word for further work.

A valuable advantage is that the command log can still be saved as e.g. a simple list of commands given (to be used as a script file). This is possible by using the different option for saving the command log (see next section).

The documentation editor is described in a separate section; please see section "Working with documentation in the command window" starting on page 771.

Note that when you add a stand-alone command window with the command **Windows > New Command Window** that window will have the same status of the **Show Toolbar** context command as the internal command window. Once created, the windows are independent of each other when it comes to this setting – this enables you to have, for example, the internal window showing only the command log, while you can have the toolbar enabled in the stand-alone command window.

The command log file

The command log is what is displayed in the command log pane of the command window. The command log can be saved as a file. The content in the command log pane and the content in the corresponding saved file may differ, depending on what format the file is saved in.

- The base for the command log is the commands given (by typing them in the command input line, by clicking on command buttons, by using menus or by using scripts etc.). It is possible to save *only* those commands, without any results. This is done by saving the command log in `.mos` format. Such a file is a script file. The creation and use of script files is described in section "Scripting" starting on page 904.

  It should be noted that all commands given are not displayed in the command log. Some commands do not generate any Modelica code (e.g. some commands that changes the user interface of Dymola; e.g. displaying a new window). Some commands are not displayed until they are considered concluded, e.g. plot commands. The reason for the last item is

that it would not be a wanted feature if any changing of a plot was displayed as a new command – the plot command is displayed when another action is executed; then the plot command is regarded as concluded. Note concerning the plot command that the user can force such a conclusion by clicking the command input line.

- The saved log file can also contain simulation results in text format. This corresponds to saving the file in .txt format, and it also corresponds to using the command log pane in textual mode (without the user exporting images or requesting html output).
- The third possibility is to have the same information saved as is displayed in the command log pane, including all features and formatting/rendering available there. This corresponds to saving the command log in .html format.

Saving the command log using the command **File > Save Log…** gives the possibility of using any of the three formats above. When a script file should be saved, also the command **Script Editor > Generate Script > Command log** can be used.

The default location for saving the log is in the current working directory of Dymola. For more about the current working directory and how to change it, and how to set startup directory, see section "Handling the current working directory" on page 770 below.

Note the handling of image objects and local links in the log, if such are present. In particular note the recommendations for facilitating moving of the log (and additional relevant folders) in such cases.

- Concerning local links please see section "Details of the documentation editor" starting on page 775, information about the command button **Link**.
- Concerning images inserted, please see section "Details of the documentation editor" starting on page 775, information about the command button **Image**.
- Concerning "math" objects created by e.g. the command button **Equation**, see the section "Storage of "math" objects referred by a stored command log" on page 772.

Please note that when working with the command log using the command log pane you have yourself to save the log. Dymola will not warn if the log is not saved when closing Dymola.

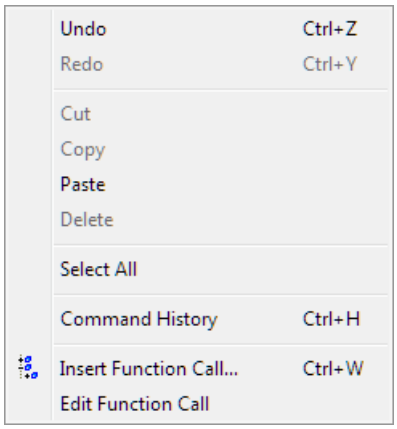
The command input line

The command input line can be used to manually enter commands (e.g. setting flags). Commands given work against Dymola workspace e.g. when defining variables and changing values. Please note that this workspace is also used by the present model; be careful not to interfere with variables in the model if not intended to!

You can go back to previous commands using the **arrow up** key on the keyboard. (**Arrow down** will also work when arrow up has been used previously.) Whey you do that; you also open the commands history window. See section "Working with the command history" starting on page 780.

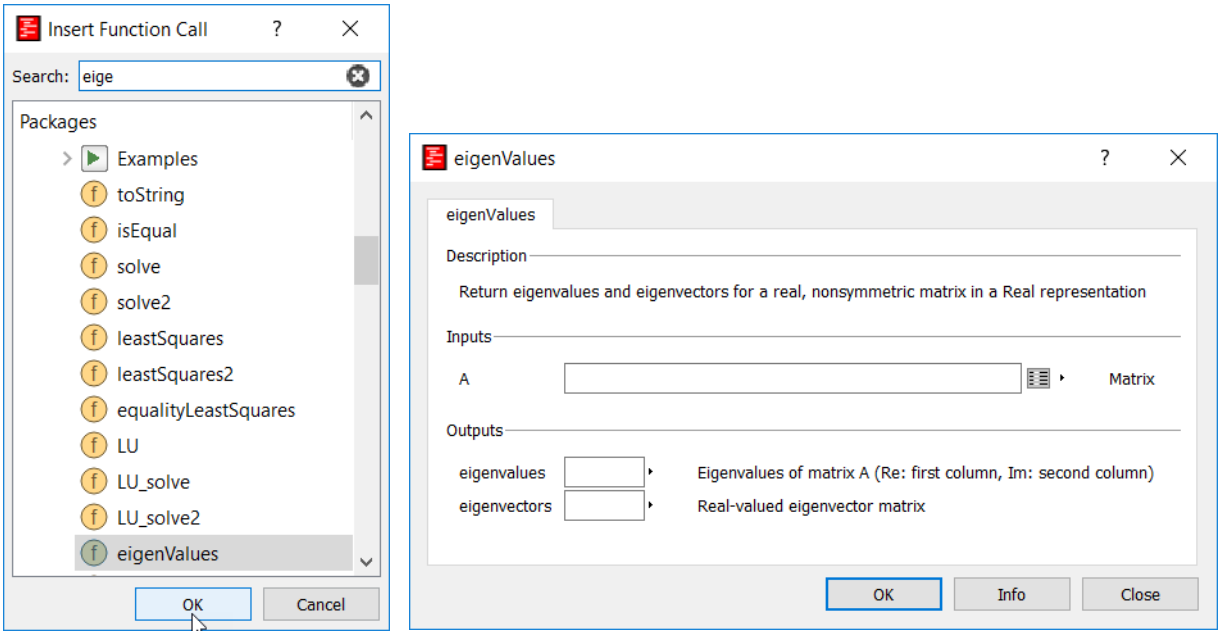
The command input line (and the command log) is important when creating scripts. Please see section "Scripting" starting on page 904 for more information.

By right-clicking in the command input line a context menu is available:



For more information about this menu, please see the section “Context menu: Command window – Command input line” on page 883.

Note that the entry **Insert Function Call…** enables searching (or browsing) for a function and then entering the arguments, allowing easy insertion of a function call. As an example, entering eige in the search input field (in order to find `Modelica.Math.Matrices.eigenValues`) will give the following result (followed by the resulting parameter dialog of the function: that will pop when **OK** is clicked):



Also note the **Edit Function Call** entry, that makes it possible to edit a given *interactive* (library) function call in a very convenient way without having to look into the sometimes lengthy result in the command log pane of the window. (An interactive (library) function is

visible in the package browser. Note that also built-in functions can be displayed in the package browser, see section "Built-in functions in Dymola" starting on page 937.)

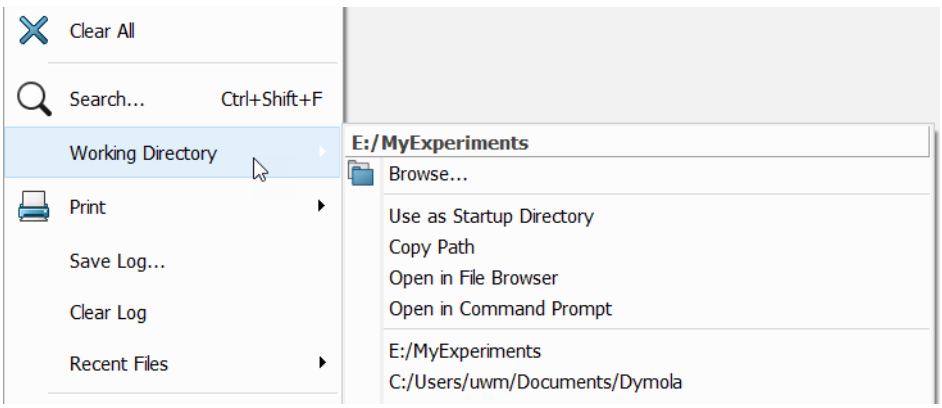
It is possible to insert the name of a function call in the command input line also by code completion, by using **Ctrl+Space**, if (at least) the first character of the function call is typed in first. As an example, entering p and then pressing **Ctrl+Space** gives plot. Keeping **Ctrl** pressed and clicking **Space** again gives the next function, in this case plotArray. Clicking **Space** again (**Ctrl** still pressed) gives plotArrays etc. However, it is presently not possible to do more than inserting the name of a function by code completion.

Handling the current working directory

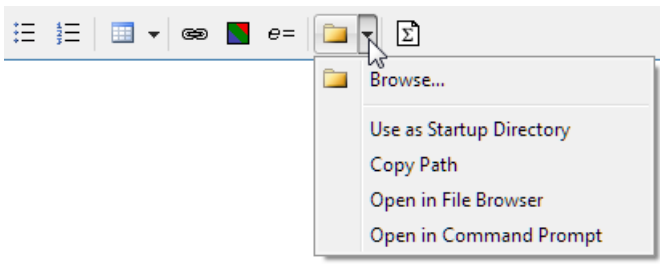
The current working directory can be displayed by entering `cd` in the command input line of the command window (followed by Enter). The same command can be used to change the working directory.

A more elaborate way to display/change/access the current working directory, and also to set the startup directory, is to use the **File > Working Directory** commands. They are available as that command, but (except from the history) also as dropdown in the **Working Directory** button in the Commands window (given that the toolbar is activated in that window).

The **File > Working Directory** commands:



The **Working Directory** button in the Commands window:



The **File > Working Directory** commands are described in previous chapter. See index entry "*directory : current working directory*" in the index in the end of the manual to find the relevant page.

5.2.10 Working with documentation in the command window

General

The documentation editor available in the command window enables a good documentation of simulations. The documentation editor is basically the same as the one available for the documentation layer of the Edit window. For details, see below.

Basic features available in the command log pane

It should be noted that all commands given are not displayed in the command log. Some commands do not generate any Modelica code (e.g. some commands that changes the user interface of Dymola; e.g. displaying a new window). Some commands are not displayed until they are considered concluded, e.g. plot commands. The reason for the last item is that it would not be a wanted feature if any changing of a plot was displayed as a new command – the plot command is displayed when another action is executed; then the plot command is regarded as concluded.

The command **Tools > [Export] Image…** can be used to save a PNG image of the active window (without window borders and plot bar, if any). The image is identical to the image shown in the window, so the size and representation can be changed by first resizing the window. For more information about this command (including support for high resolution images from animation) please see index entry "*tools tab : [export] image*" in the index in the end of the manual. (When it comes to plot and animation windows, please note that those can be included by default using options of the documentation editor in the command log pane. Please see section "Plots automatically inserted in command log" on page 774 and "Animation window content automatically inserted in command log" on page 775.)

Html output generated from commands will be included. Such a command is given by e.g. executing the function `Modelica.Utilities.Streams.readFile`, explicitly specifying an html file (the selectable format is only txt, but typing e.g. MyFile.html will work). The file must start with the tag <html> and end with the tag </html>. Images might not be displayed correctly in the command log pane, nor will links work, but if the command log is saved in html format the images will be correctly displayed and the links will work in that file.

Additional features

A number of additional features are available in the command log pane:

- Documentation editor (if the toolbar is activated)
- Math rendering of formulas and equations etc, (if the toolabar is activated) including Greek characters and index rendering (must be activated by the command button **Mathematical Notation**, see below).

- Plots automatically inserted in command log (must be activated by a flag, see below)
- Animation window content automatically inserted in command log (must be activated by a flag, see below.

Documentation editor

Please see section "Details of the documentation editor" below.

Math rendering of formulas and equations, including Greek characters and index rendering

Rendering as text or not for inputs and outputs



The "math" formatting of input and output is controlled by the command log toolbar command button **Mathematical Notation**. This functionality is by default deactivated. To activate it, click the button. (This corresponds to setting the flag `Advanced.MathematicalNotation.InCommandLog=true`.) The button works in toggle mode. The status is saved between sessions.

The input from the command input line in the command window is analyzed by Dymola. If the input is evaluated as an expression, the input can be presented in a formatted way, "math formatting". If the input cannot be evaluated as an expression, then the input will always be presented as text.

The output is analyzed and presented the same way.

Input/output formatted as "math" can be selected as objects in the command log, however parts of such an object cannot be selected/copied. Input/output presented as text can be marked as usual text. The text in a "math" object will have a somewhat different style compared to a corresponding text.

Rendering for equation/expression inserted using the toolbar command button Equation

Equations/expressions inserted using the button **Equation** will always be treated the same way as "math" objects when it comes to rendering (they are not affected by the button **Mathematical Notation**). However, they can be edited by the user.

Storage of "math" objects referred by a stored command log

If a command log contains any "math" object, a folder *`DymolaLogName`*`_files` including a sub-folder `generated`, will be automatically created when saving the log in .html format by the command **File > Save Log…**. The folder *`DymolaLogName`*`_files` will reside in the same folder as the saved log file. The "math" objects will be handled in the following way:

- Math objects created using the button **Equation** or created when editing such equation/expression will be copied from a temporary folder to the sub-folder `generated`. They will be stored as .png images.
- Math objects created by Dymola as math rendering output to commands will be created in the sub-folder `generated` when saving the log. They will be stored as .png images.

Greek characters

If e.g. a variable name is interpreted by Dymola as a Greek character, Dymola will render that character in the following cases:

- For inputs/outputs that can be evaluated as expressions, if the "math" formatting of inputs and outputs are activated as above, *and* the Greek character rendering is activated (the latter is the default, see below).
- For equations and expressions inserted by the command button **Equation**, if the Greek character rendering is activated (this is the default, see below).

For example, `alpha` is rendered α, and `Gamma` is rendered Γ. This also applies when the Greek character is trivially embedded, as for e.g. `alpha_2`, rendered α_2. Trailing numbers following Greek characters are rendered as subscripts; e.g. `alpha12` is rendered α_{12}.

Greek characters can of course also be used as indexes, e.g. `x_beta` will be rendered x_β.

The rendering of Greek characters is by default active, to disable it, set the flag

```
Advanced.RenderGreekLetters=false;
```

Index rendering

The condition for index rendering for inputs/outputs that can be evaluated as expressions is that the "math" formatting is activated, for equations and expressions inserted by the command **Equation** indexing is always used.

Index within brackets [ ] are rendered as subscripts, e.g. `x[2]` is rendered as x_2 and `alpha12[3]` is rendered as $\alpha 12_3$.

The content after the last underscore _ in e.g. a variable name is rendered as a subscript. E.g. `v_12` is rendered v_{12}. Exception: When indexing using brackets [ ] is used, that will overrun the above, e.g. `v_12[3]` is rendered v_12_3.

Some examples of rendering

Modelica expression	**Rendering**
division **`(a*(1+b))/(1+c)`**	$\frac{a(1+b)}{1+c}$
square root **`sqrt(1/(1+a))`**	$\sqrt{\frac{1}{1+a}}$
power **`(1+a)^(1+sigma)`**	$(1+a)^{1+\sigma}$

absolute values **abs(1/(a-3))**	$\left\lvert \frac{1}{a-3} \right\rvert$
sum **sum({1,2,3})**	$\sum\{1,2,3\}$
simple matrix (declaration) **A=[1,2;3,4]**	$A=\begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix}$
derivative of x **der(x)**	$\frac{\mathrm{d}\,x}{\mathrm{dt}}$
array construct **{i+j for i in 1:3, j in 4:7}**	$\{i+j \text{ for } i \in 1:3, j \in 4:7\}$
if-then-else-expressions **y = if x<0 then -x else if x==0 then 0 else if x==1 then 1 else x**	$y=\begin{cases}-x & \text{if } x<0\\ 0 & \text{if } x=0\\ 1 & \text{if } x=1\\ x & \text{else}\end{cases}$
logical expressions **p and not (p or q)**	$p \wedge \neg(p \vee q)$

The examples above do not cover all features, they just point to the fact that a more user-friendly rendering is implemented. This will of course also be the case when e.g. making a function call, the call

```
Modelica_LinearSystems.StateSpace.constructor([1,2;3,4],[1;2],
[1,2],[0]);
```

will give the result:

$$= \text{Modelica_LinearSystems.StateSpace}\left(A=\begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix}, B=\begin{bmatrix}1\\ 2\end{bmatrix}, C=\begin{bmatrix}1 & 2\end{bmatrix}, D=\begin{bmatrix}0\end{bmatrix}\right)$$

Plots automatically inserted in command log

Whenever a plot function is executed, the content of the plot window can be inserted in the command log. To activate this functionality, the flag

`Advanced.ScriptingWindow.IncludePlot` has to be set to true. (The flag is by default set to false.)

Please note that displaying variables in the plot window by selecting variables from the variable browser does not in itself execute the plot function – then the plot function should be executed each time a variable is selected/deselected which would not be a wanted feature. The plot function will be executed (displayed in the command log) when another action is performed in Dymola (e.g. another simulation). However, the user can force Dymola to execute the plot function when working with the variable browser by clicking in the command input line of the command window.

Please also note that it is easy to insert a plot from the plot window using the command **Generate > Image in Log**. See section "Insert plot and its corresponding command in command window" on page 753.

Animation window content automatically inserted in command log

Whenever the function `animationRedraw()` is executed, the content of the animation window can be inserted in the command log. To activate this functionality, the flag `Advanced.ScriptingWindow.IncludeAnimation` has to be set to true. (The flag is by default set to false.)

The user can execute this function by entering `animationRedraw()` in the command input line of the command window.

Editing in the documentation editor

Editing is facilitated by the toolbar of the documentation editor.

Details of the documentation editor

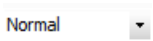
The toolbar available, if the command log context command **Show Toolbar** is activated (default):



Note. The commands in this toolbar are available also for the documentation editor for the documentation layer of a class when editing it (in the **Documentation** tab). All these commands are described in the corresponding section in previous chapter. To find that section, please refer to the index entry "*editor : documentation (of class)*".

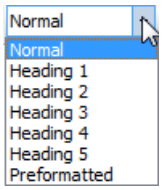
Below only those commands that differ in some way, or needs specific description for the command log, are described (they are marked in the figure above). For the rest of the commands, refer to the index entry mentioned.

(The last command in the toolbar is not for documentation but for handling of the current working directory. For this command, see "Handling the current working directory" on page 770.)



Text style contains presently six selections, **Normal** being default. **Preformatted** is the text style of text generated from Dymola.

Text style alternatives.

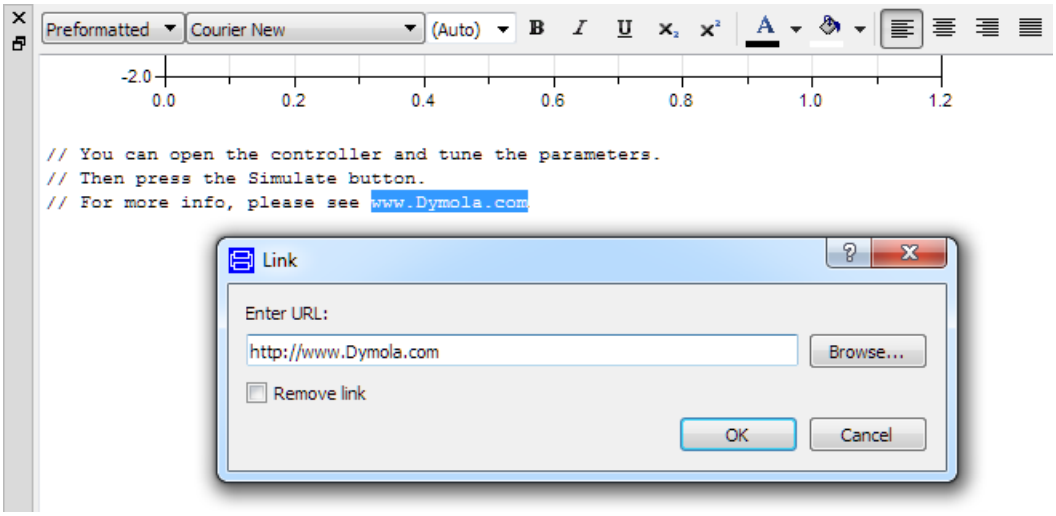


When a text has been entered in a heading style, the next paragraph will by default be of style Normal.

When changing the base font size in Dymola, the text sizes in the command window (and the saved html log) will change accordingly. (E.g. if the base font size is changed from 8 to 10, all text styles will be 2pt larger in the command window; the default size will then be 10pt as well.)



The **Link** button will open a menu for entering of an URL according to usual html rules. If you put the cursor anywhere on a link that is already present and click the **Link** button, that link will be shown.



Note that if you type a text in the documentation editor that can be interpreted as a link this link is automatically created when you do any of **Space**, **Enter**, or **Tab** after having typed the text. (This featue can be disabled by setting the flag `Advanced.AutoLinksDoc=false`.)

In the dialog for the **Link** button, the **Browse** button makes it possible to create local links to e.g. .pdf and .html files. This feature enables creation of documentation that refers to external documentation.

When an URL is entered/browsed and the **OK** button is pressed, the selected text will be underlined and blue.

In the dialog for the Link button, you can use **Remove link** to remove an existing link.

There is no syntax check of the URL. The links cannot be activated from the command window, the log has to saved (by the command **File > Save Log…**, then the log can be opened (like any html file); the links are now active.

It is possible to select a formatted text when creating the link, as well as an image, and a mixing of text and image. Images will not be underlined when in links, however.

Local links created will be relative to the location of the working directory of Dymola. Please note the consequences of this:

- The command log has to be saved in the working directory of Dymola, at least initially. This will be the default directory when the log is saved by the command **File > Save Log…**.
- If the command log should be moved, documents referred to should be stored in such a way that it is easy and intuitive to move them together with the log. We recommend storing the document referred to in a folder named e.g. LinkedDocuments located in the working directory of Dymola *before* referring to them by the **Link** button. By moving this folder together with the log, the links are preserved.

Please compare this with the corresponding handling of images in the command log using e.g. the **Image** button, please see below.



The **Image** button will open a browser for browsing an image. A number of formats are supported, e.g. .png, .xpm, .jpg, .tiff and .bmp. There is no scaling of the image.

The starting point of the browser is the default directory in Dymola.

When browsing for the image the file path is shown in the menu. When clicking **Open**, the image will be inserted

If a command log contains any images, a folder `DymolaLogName_files` is automatically created when the log is saved in .html format by the command **File > Save Log…**. This folder will be located in the same location as the saved log file. The images will be *copied* to the folder `DymolaLogName_files` when the log is saved.

The generated links in the .html log will be relative to the folder `DymolaLogName_files`. The result will be that it will be easy to move the complete log by moving the .html log file and this corresponding folder, without losing any links.

If the log is copied and inserted to e.g. Microsoft Word, the pictures will be inserted as well.



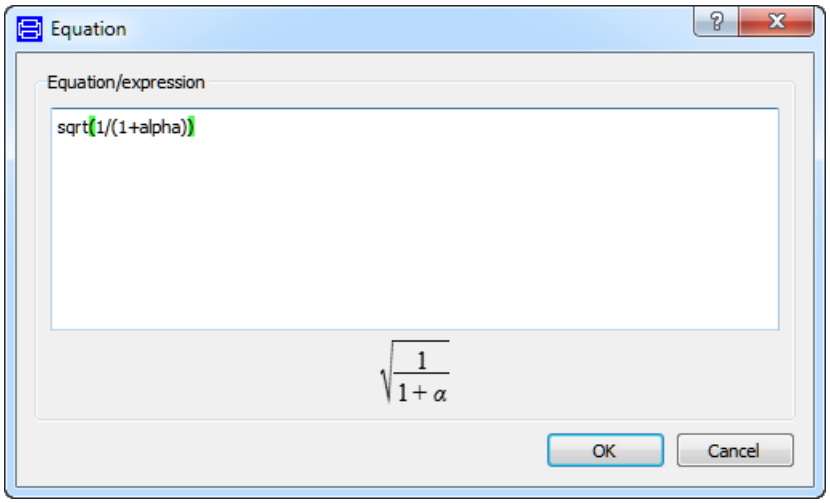
If the link is invalid (image not found) in the shown html log it will be indicated with a symbol.

Please note that the command **Tools > [Export] Image** can be used to insert a PNG image of the active window (without window borders and plot bar, if any) into the command log pane. This is also a general feature of the command log pane. See index entry "t*ools tab : [export] image*" in the end of the manual for more information about this command.



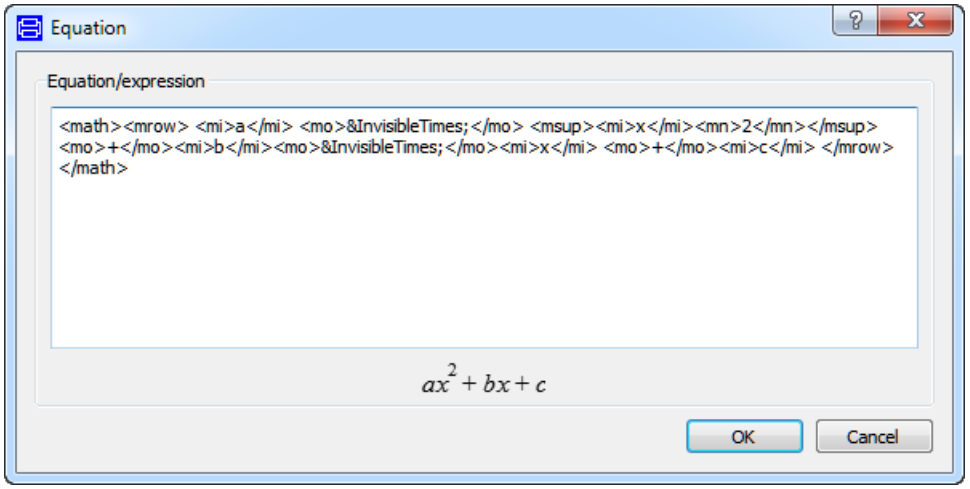
Clicking the button **Equation** opens an editor for the creation of an equation/expression.

Example of creation of a square root expression.



The result of the editing is shown below the editing pane. If an equation/expression is not concluded, the text `Incomplete equation/expression` is displayed. Clicking **OK** the equation/expression is inserted in the documentation editor where the cursor was located when pressing the **Equation** button in the first place.

Instead of using Modelica syntax, you can use MathML. An example:



Equations or expressions created this way are displayed in italics. They are in fact images; nothing inside can be selected, looking at the command log saved in html format will display these items as pictures. They can however be edited in Dymola anyway.

If such an expression or equation should be edited, select that equation/expression. The cursor will place itself after or before the equation/expression. Now click on the **Equation** button.

The editor described will be displayed, and editing of the equation/expression can be made. Clicking **OK** the edited equation/expression will replace the previously selected one.

Note that if you enter an expression in the command input line, it can also be displayed as an image. However, you cannot edit it by the **Equation** button. The reason is that this expression is a command, which should not be edited as an image.

(The images created using **Equation** are stored in a sub-folder `generated` in the folder *`DymolaLogName`*`_files` that is located in the folder the log resides in. Editing such an equation/expression will create a new image for each editing. The folder will be automatically created if not present.)

For some examples of the rendering of examples/expressions, please see section "Math rendering of formulas and equations, including Greek characters and index rendering" starting on page 772.

Please note that it is still possible to enter simple expressions/equations by typing them in directly without using the button.

More information about this menu is available in the section "Context menu: Command window – Command log pane" on page 882.



The **Mathematical Notation** button is used to activate the mathematical notation in the command log pane. By default, this notation is not activated. The button works in toggle mode. For more information about mathematical notation, including the flag corresponding to this button, see section "Math rendering of formulas and equations, including Greek characters and index rendering" starting on page 772. (The status of the button is saved between sessions.)

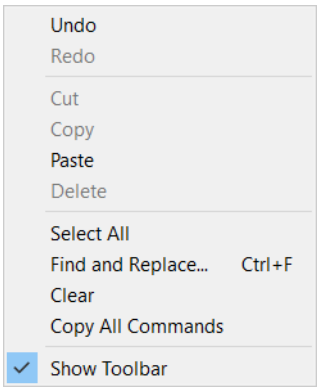
Commands

The command **File > Save Log…** can be used to save the log in various formats. Please see section "The command log file" on page 767. The command **File > Clear Log** will erase the log in the command log pane.

Context menu

A context menu for the command log pane in documentation mode is available by right-clicking in the documentation editor:

Context menu of documentation editor.



The entry **Copy All Commands** is important in scripting – since it copies commands but not results from the command log, the result can be pasted into the Modelica Text layer of a scripting function.

For more information about this menu, please see the section “Context menu: Command window – Command log pane” on page 882.

5.2.11 Working with the command history

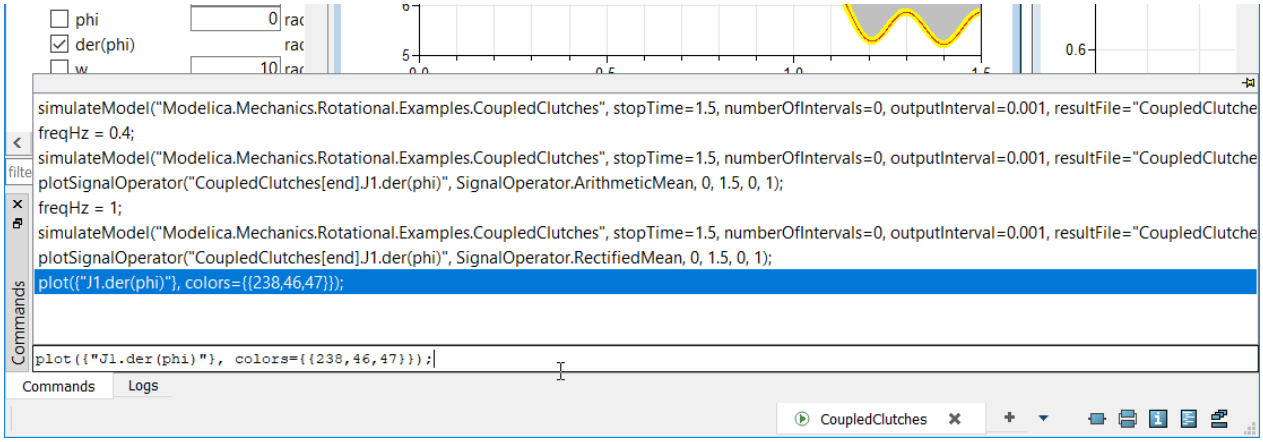
An important part of the command log is the command history. Previously it has been mentioned that you can extract the command history from the command log by right-clicking in the command log and select **Copy All Commands**. You can paste the result, for example, in a script by opening a new script in the script editor and paste in that one.

However, there is also an altenative to work more in detail with the command history, and that is to use the command history window.

The command history is available in a window in two ways, either as a frameless window docked to the command input line, or in an additonal stand-alone window that also has a toolbar and a context menu with valueable commands.

The frameless command history window

The frameless command history window appears when using arrow up from the command input line when you have a command history. An example:



In fact, the history window is this form can only be used for navigating in the commands history and execute selected command. You can use arrow up/arrow down to navigate. When reaching the end of the command history the window is closed again, using arrow up opens it again. If you click on a command, it is displayed in the command input line. If you press enter, the selected command is executed.

To just remove the frameless window without changing active command, use arrow right. To remove the frameless window and reset the active command as the last one, click **Escape**.

Important!

It is important to know that the command history in the commad history window is saved between sessions; this means that you usually have more commands in the commands history window than in the current command log. This also means that the context command **Copy All Commands** in the command log does not copy all commands in the commands history, only the commands in the current command log.

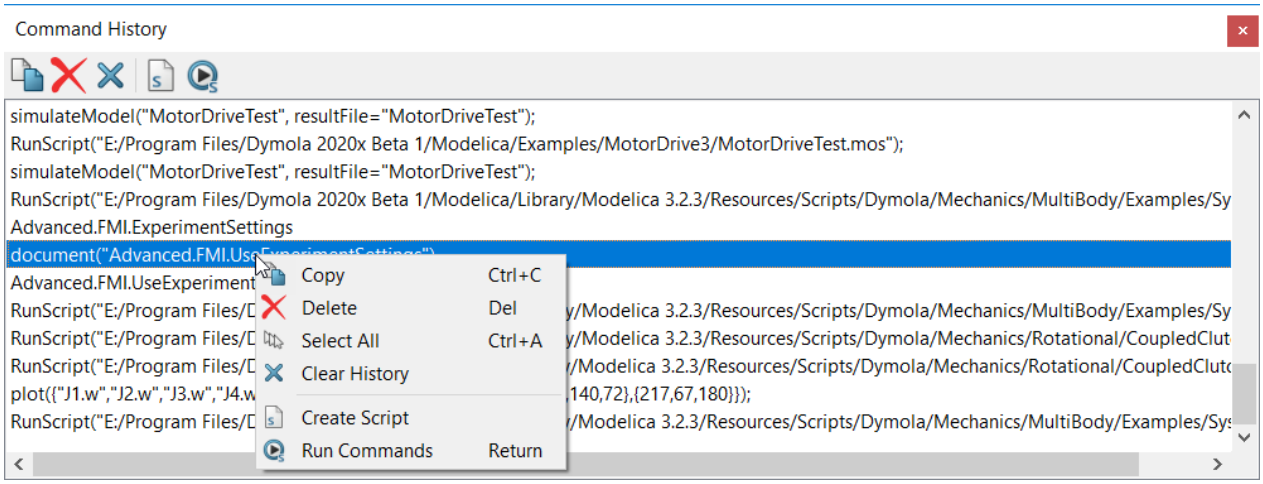
The full command history window

To be able to work with the command history fully, you can display the command history in an ordinary window with a toolbar and context commands.

You can do this in three ways:

- Click the header of the docked window seen in the figure above.
- Right-click the command input line and select **Command History**.
- When in the command input line or the command log pane, press **Ctrl+H**.

The command history window now appears; the following example is from clicking the frame of the window in the figure above (with a line also righ-clicked to illustrate the context menu):



Except having a context menu, the window also has a toolbar with the same commands, except for the **Select All** command. The commands are described in the section "Context menu: Command history window" on page 885.

Of particular interest are the commands **Create Script** and **Run Commands**, which makes it possible to create a script from selected commands by inserting them to a new script in the script editor, or to execute the selected commands.

5.2.12 Simulation settings

General

There are two possibilities to control how the simulation should be performed:

- The setup menu alternative, using the **Setup** button or the command **Simulation > Setup**. A number of tabs will be displayed to control various aspects of simulation. Please see section "Simulation > Setup" starting on page 799 for more information.
- Using commands entered in the command input line of a command window or executing a script. The script could be created using the command **Script Editor > Generate Script**. Please see section "Script Editor > Generate Script" on page 833. Another way to create the script is to use the setup menu alternative above to tick the wanted settings and look at the corresponding text in a command window – and use that information to create a script. Please also see section "Scripting" starting at page 904.

Advanced setup for simulations by flags

For advanced users, there are a number of flags that can be used to fine-tune the simulation:

- `Advanced.Simulation.StepSizeMin` The minimum step-size of integrators if no-zero, hmin. Use with care, since too large value will lead to simulation failure. Note that this flag is only valid for dense output integrators. The default value is `Advanced.Simulation.StepSizeMin=0.0`

- `Advanced.Simulation.StepSizeMax` The minmum step-size of simulators if non-zero, hmax. Use with care, since too small value will lead to simulation failure. Note that this flag is only valid for dense output integrators. The default value is `Advanced.Simulation.StepSizeMax=0.0`
- `Advanced.Simulation.StepSizeStart` The starting step-size of simulators if non-zero, h0. Note that this flag is only valid for dense output integrators. The default value is `Advanced.Simulation.StepSizeStart=0.0`
- `Advanced.Simulation.EventEpsilon` The time resolution for events, eveps. The default value is `Advanced.Simulation.EventEpsilon=1e-10`
- `Advanced.Simulation.MaxEventIterations` The maximum number of event iterations, eviter. Use with care, since too small value will lead to simulation failure. The default value is `Advanced.Simulation.MaxEventIterations=20`

Variable frequency of output points

Normally the interval length (or indirectly the number of output points) determines the frequency of results.

By setting the flag:

```
Advanced.Simulation.VariableInterval = true;
```

And in the model use

```
when time >=… then
      Dymola.Simulation.SetOutputInterval(…);
end when;
```

the frequency can be controlled from the model.

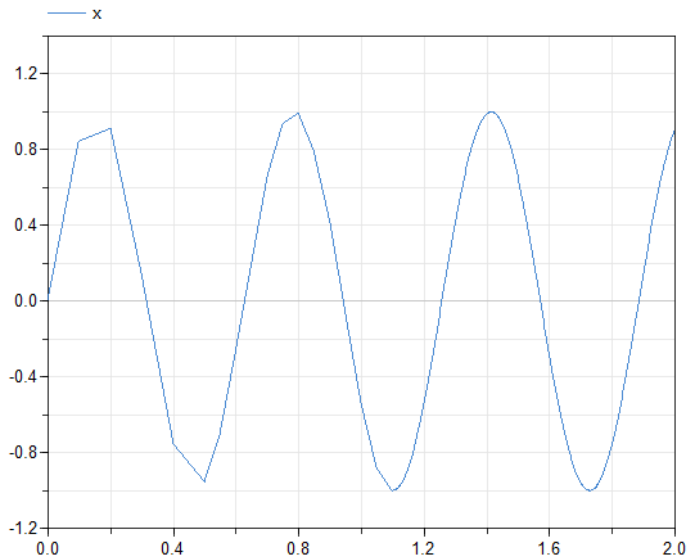
Note that the simulation setup is used until the first call of `Dymola.Simulation.SetOutputInterval(…);`

A full example, with stop time 2 and interval length set to 0.002 in the simulation setup, and then running (with the above flag set):

```
model VariableOutputInterval
   "demonstrate the use of variable output interval in a model"
   Real x;
equation
   x = sin(10*time);

   when initial() then
      Dymola.Simulation.SetOutputInterval(0.1);
   end when;
   when time >= 0.5 then
      Dymola.Simulation.SetOutputInterval(0.05);
   end when;
   when time >= 1.1 then
      Dymola.Simulation.SetOutputInterval(0.002);
   end when;
end VariableOutputInterval;
```

will give the below curve when plotted:



Notes:

- The only way to "reset" the interval length to the one in the simulation setup is to explicitly set that interval, this is done in the last `when` statement above.
- You must have a separate `when` statement for each time.

Parameter evaluation

Evaluation of parameters in Dymola is by default to evaluate structural parameters (parameters whose values are needed to run the model, e.g. dimensions of a record).

However, there are ways to control this:

- A global setting using the setup alternative above, selecting the **Translation** tab and ticking the setting **Evaluate parameters to reduce models**. This setting will have *all* parameters except top-level parameters evaluated at translation in order to generate more efficient simulation code. (The setting **Also evaluate top-level parameters** can be used to also evaluate the top-level parameters.) Please see section "Translation tab" on page 805 for more information. (This first setting is also reachable using the flag `Evaluate`; e.g. forcing evaluation by typing `Evaluate = true` in the command input line of the command window. The default value is false. The setting about top-level variables corresponds to the flag `Advanced.EvaluateAlsoTop`.)
- Specify that a certain parameter should be evaluated. This is done using the annotation `annotation(Evaluate=true)`. The annotation `annotation(Evaluate=false)` will force the parameter not to be evaluated. This setting will override the global settings above. Notes:

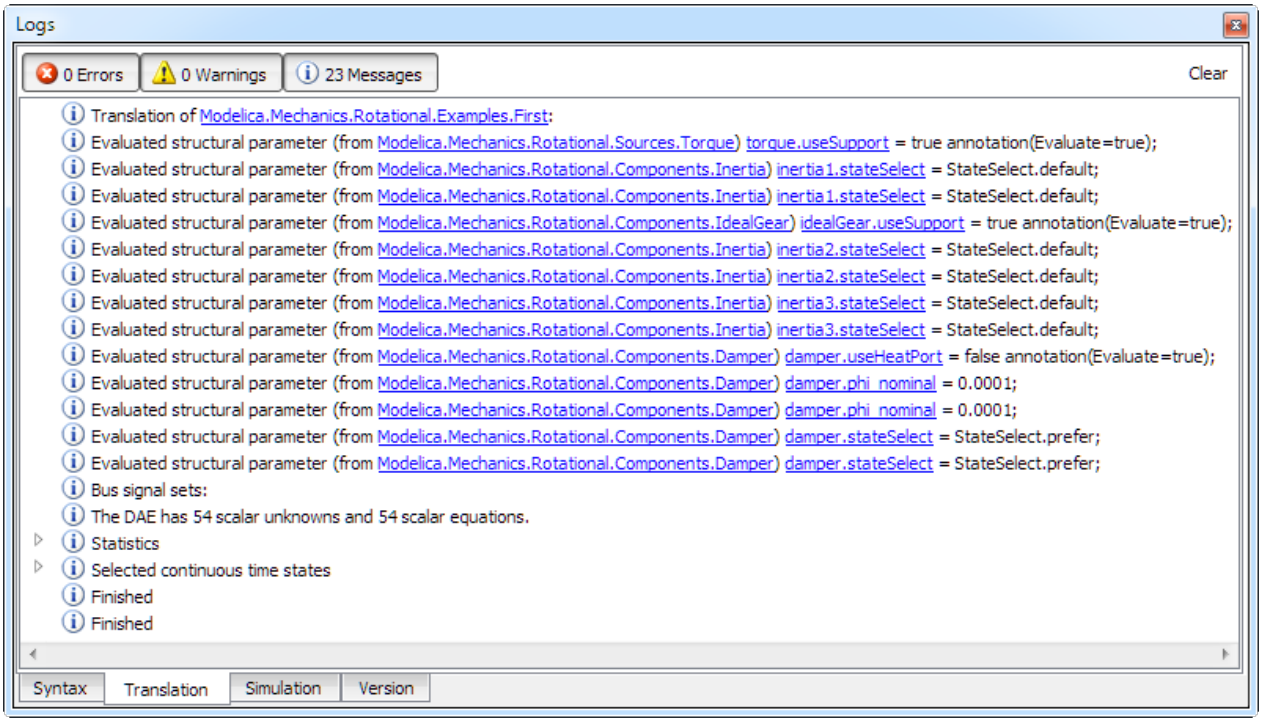
- If you use the annotation `annotation(Evaluate=true)` on a record or model component all corresponding parameters will be evaluated even if some of them individually have the annotation `annotation(Evaluate=false)`. This means that you do not have to add the annotation `annotation(Evaluate=true)` to each individual parameter of a record or model component to evaluate it. Note that this does *not* apply if you use the annotation `annotation(Evaluate=false)` on a record or model component; this annotation will not override any annotation of any corresponding individual parameter.
- If a parameter having the annotation `Evaluate=true` is bound to a parameter expression, the parameter expression will also be evaluated, unless it involves parameters with annotation `Evaluate=false`.
- Please see chapter "Developing a model", section "Advanced model editing", sub-section "Parameters, variables and constants" for more information about how to set these annotations in the variable declaration dialog.

For general information about parameter values in Modelica, please see chapter "Introduction to Modelica", section "Initialization of models", sub-section "Parameter values".

For the handling of errors and warnings of parameter values when translating the model, please see previous chapter, section "Advanced model editing", sub-section "Parameters, variables and constants".

Logging of evaluated parameters is possible in two ways.

- You can set the flag `Advanced.LogStructuredEvaluation = true`. If set, all parameters that are evaluated during the translation are logged, and the result is presented in the simulation log, displayed in the **Translation** tab of the Dymola log window. (The flag is by default `false`.) Links to the evaluated parameters are included in the result, as well as display of `annotation(Evaluate=true)` to indicate parameters that the modeler recommended to evaluate. An example of result:



- You can also log, in a file, what are the parameters in a model after translation (the parameter evauation) by setting the flag `Advanced.Translation.LogParameters = true`. If set, a text file `parameters.log` is generated in the working directory when translating a model. The list of parameters in the file will have a second column with the alternatives `fixed`, `tunable`, or `dependent`, respectively.

Storing of protected variables

A variable can be declared as protected ticking **protected** in the **Type Prefix** tab when declaring it (or using **Attributes…** in the context menu of a component to declare several variables as protected). Please see previous chapter, section “Advanced model editing”, sub-section “Parameters, variables and constants” for more information.

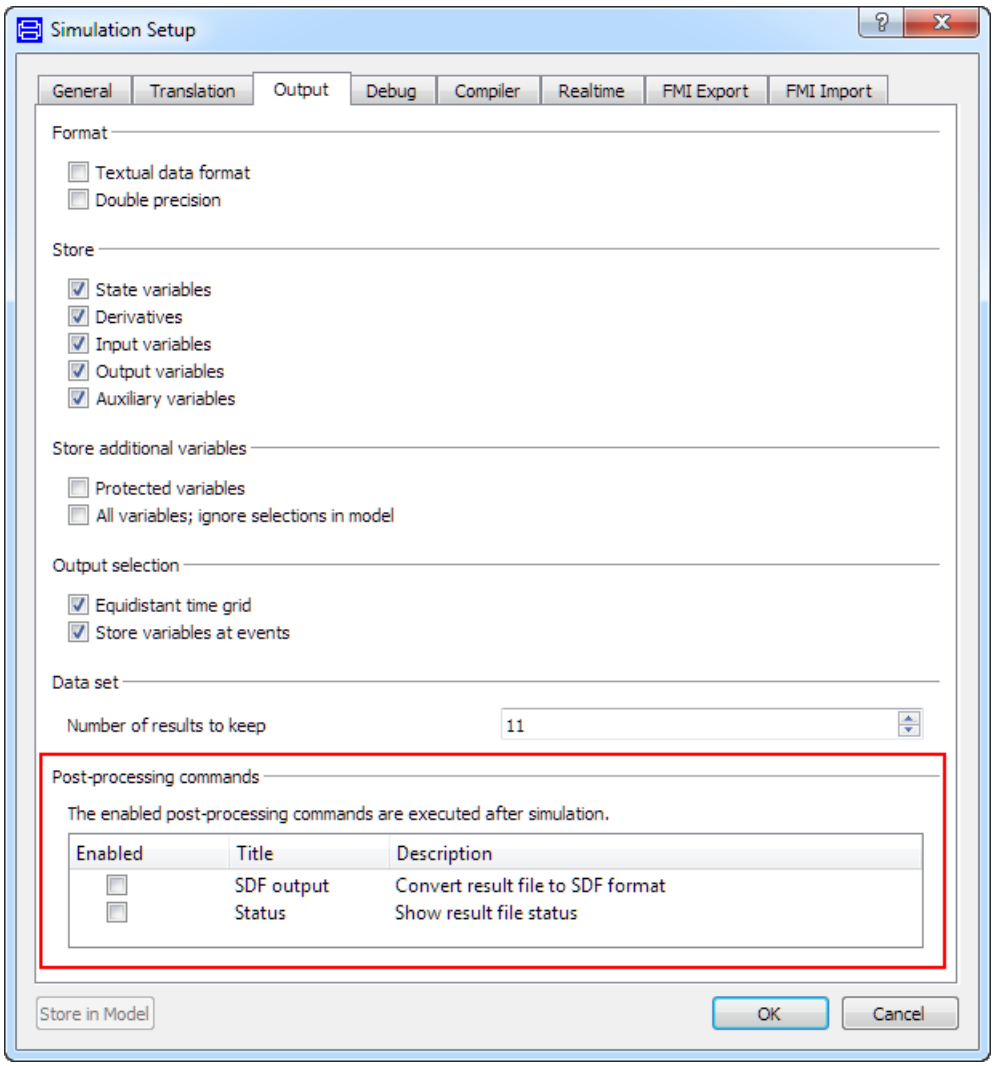
Protected variables are by default not stored during simulation. This makes it easier to find the relevant information, but in some cases detailed post-processing require access to protected variables. To store protected variables a global setting using the setup alternative above is available, by selecting the **Output** tab and ticking the setting **Protected variables**. Please see the section “Output tab” on page 809 for more information. (This setting is also reachable using the flag `Advanced.StoreProtectedVariables` e.g. activating storage by typing `Advanced.StoreProtectedVariables = true` in the command input line of the command window. The default value is false.

Please note the close relation with hidden variables. A variable can be declared as hidden by using the annotation `annotation(HideResult=true)`. Such a variable will not be stored.

Please see previous chapter, section “Advanced model editing”, sub-section “Parameters, variables and constants” for more information concerning this.

Post-processing commands

Dymola has the capability to automatically run external post-processing commands after finished simulation. One or more such commands can be selected from a list in **Simulation > Setup**, the **Output** tab, the Post-processing group:

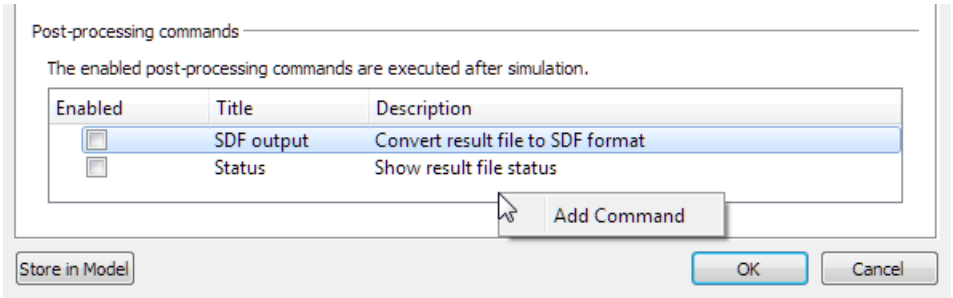


The post-processing command **SFD output** is a pre-defined command, see below. The command **Status** is user-defined in the example below.

Post-processing commands are saved between sessions.

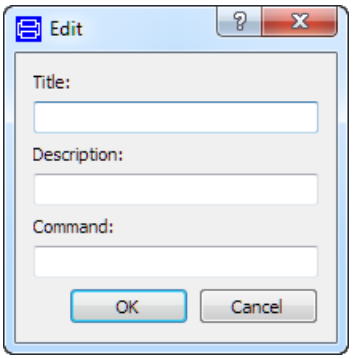
Defining post-processing commands

To define a new post-processing command, you can right-click the small area under the list of the post-processing commands. You will get then display the entry **Add Command**:



(Note that if you change the height of the window, you enlarge the area to right-click in.)

Clicking the command displays the dialog:



The text boxes correspond to a short name for identifying the command, a longer description string, and a command string. For an example of what to enter, compare with the corresponding Edit dialog for the **Status** command below.

You can also use a script function; an example is the **Status** command:

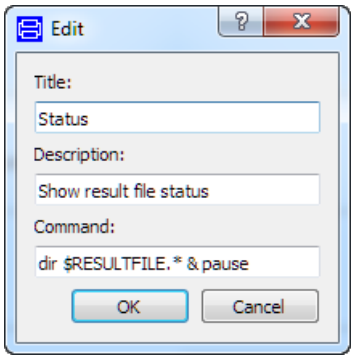
```
definePostProcessing("Status", "Show result file status",
    "Modelica.Utilities.System.command(
     \"dir %RESULTFILE%.* & pause\")", true);
```

The three first arguments are the same as in the dialog above; the last argument is a flag which says if the command should be enabled by default. The command string supports some very simple macro expansion, where `%RESULTFILE%` is expanded to the name of the simulation result file without extension (see below for an example of the expansion).

The command is executed by Dymola. To execute a command in the standard command shell of the operating system, `Modelica.Utilities.System.command()` can be used.

Editing post-processing commands

To edit a post-processing command, you can right-click the corresponding line in the Post-processing group. Doing that on the command **Status** in the figure above gives the dialog:

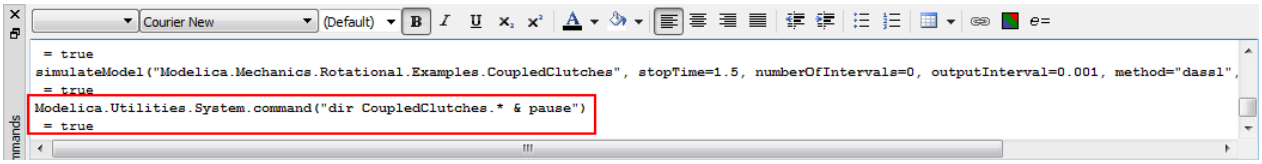


Selecting post-processing commands

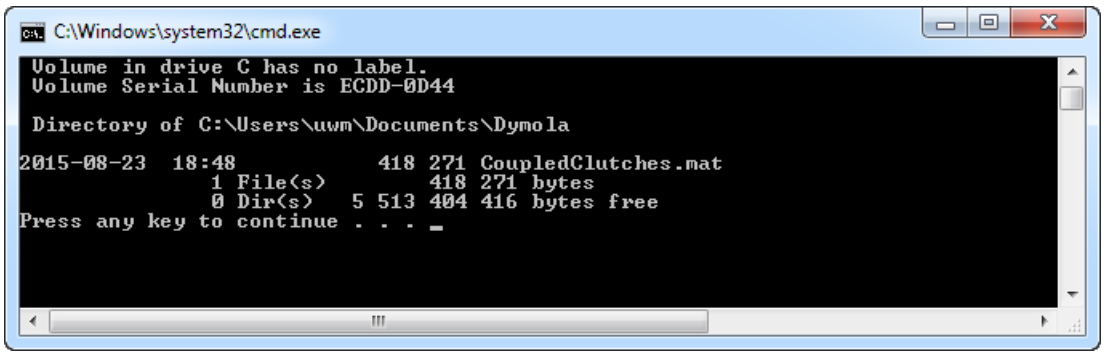
Which of the available post-processing commands are executed is selected marking the appropriate checkboxes in the Post-processing commands group, as seen in the image above. The commands will be executed in the order listed.

Execution

In the example above, we defined a "dir" command (on Windows) displaying the size of the result file. This command is automatically executed after the user runs the simulation:



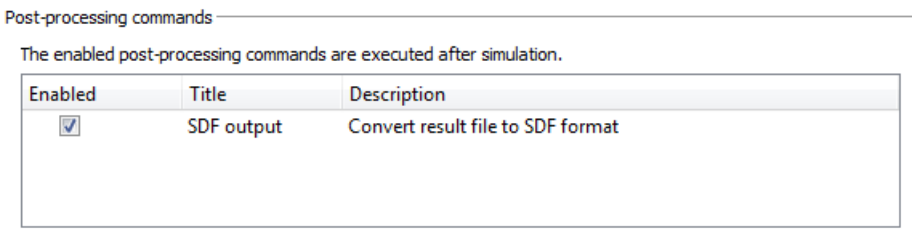
The "pause" command ensures that the command window stays visible:



Example: Support for Scientific Data Format

The Scientific Data Format (SDF) is a file format for storing table data and simulation results, using a data model based on HDF5 (https://www.hdfgroup.org/HDF5). Such files can be processed by several tools supporting HDF5.

Dymola supports copying of the simulation result file to SDF using a pre-defined post-processing command. The new file has the same name as the result file, but with extension .sdf.



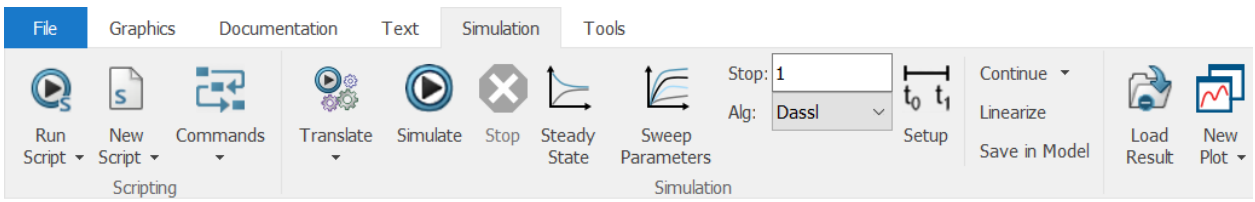
Utility functions for loading and saving SDF files in Matlab are available in the Matlab package SDF provided by the Dymola distribution. See `Program Files\Dymola 2023\Mfiles\+SDF\load.m` for documentation.

5.3 Editor command reference when working with simulation

5.3.1 Main window: File menu

For the commands in the File menu, see the previous chapter.

5.3.2 Main window: Simulation tab



The **Simulation** tab is divided above into two images since it is long.

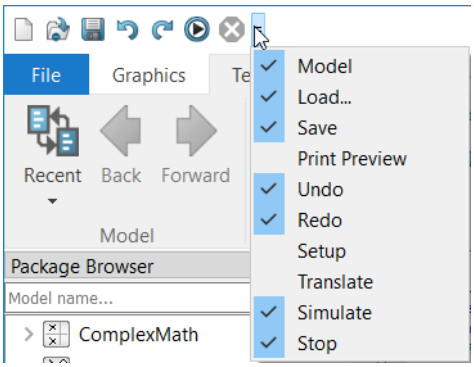
The **Simulation** tab contains commands to (the list corresponds to the tab groups):

- Handling scripts
- Simulating a model, including simulation setup and parameter sweeping
- Various tools (for example, simulation analysis, visualize and show the log)
- Handling animation

Shortcut to Simulation tab commands in the top left toolbar



The top left toolbar (quick access toolbar) by default contains buttons for simulating and stopping the simulation (the two last framed commands above). Using the arrow to the right you can configure what commands to display in the top left toolbar, for example, also add the commands for simulation setup and translation (the two first framed commands above).



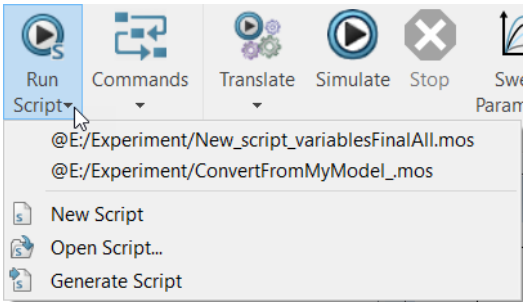
For a description of the commands, see below.

Note! For the simulation command in the quick access toolbar you have two alternatives; either you want to stay in the tab you were when giving the simulation command, or you want to switch to the **Simulation** tab when giving the **Simulation** command. You can control this by the flag `Advanced.Simulation.SwitchToSimulation`. The flag is by default `false`, meaning that you don´t switch to the **Simulation** tab.

Simulation > Run Script

Clicking directly on the **Run Script** button will open a dialog to browse for a Modelica script (.mos) file. When the script is selected, clicking the button **Open** in the dialog will execute the script.

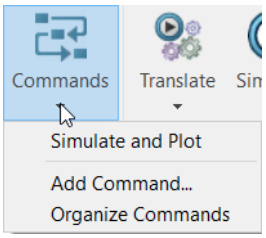
Clicking the down-arrow displays a menu:

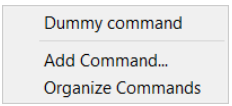


The first group shows the recently executed scripts to allow arbitrary selection.

The other alternatives are for creating a new script, opening an existing script, and generate script. Since the **Script Editor** tab will open when performing all of these commands, they are described in detail in section "Main window: Script Editor tab" starting on page 832.

Simulation > Commands





Commands to a package can be created by the user if the package is not write-protected. In the example in the side-head, the command **Dummy command** is created as an example of a user-defined command.

It is possible to include calls to scripts and opening of documents in such commands

You can also define plot window setups, including variables displayed, and animation window setups as commands.

It is possible to execute a function without displaying the function dialog. For details, see below, under the **Add Command**, the description of **Call function**.

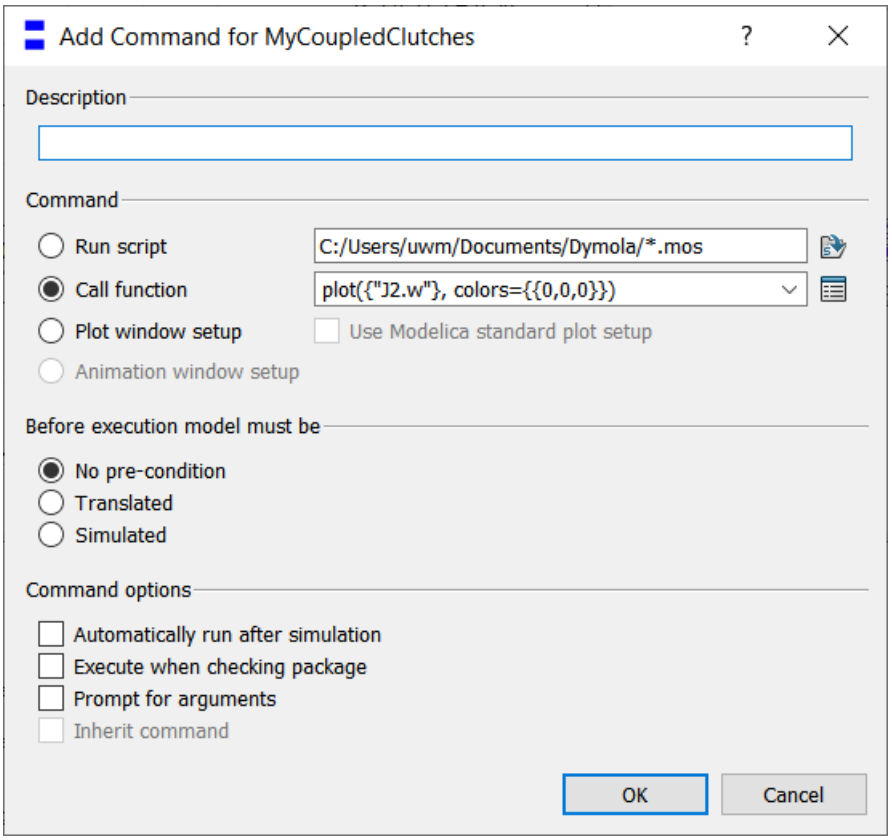
Commands in the **Commands** menu can be inheritable to models extending from it. The command must specify either that it is run after simulation (e.g. an animation setup) or that it is a function that prompts for arguments before running (e.g. a design optimization). You also have to ensure that neither the name of the model nor the replaceable contents is hard-coded into the command.

The first part of the menu displays the commands available for the current model.

After this section, two commands are available:

Simulation > Commands > Add Command

Adding a command.



This command is used to associate commands with models.

Description

Description will be the name of the command/script in the Command menu.

Executed item

Run script associate script files with the current model in the form of `file.mos`. An annotation of the following form is used:

```
annotation(__Dymola_Commands(file="file.mos" "Script to ...",
file="file.ppt" "PowerPoint presentation ..."));
```

The directory where the model was found is used in front of the file name. file.mos scripts are run everything else is opened.

Note 1: Running a script file does not perform “change directory” to its directory, thus the script file cannot run other local script files, i.e. a full path to such script is needed.

Note 2: When creating a script file using **Script Editor > Generate Script** the script file can be saved as a command ticking **Store in model as command**. Please see section "Script Editor > Generate Script" starting on page 833 for more information.

Note 3: The Modelica URI scheme 'modelica://' is supported for specifying the file location, e.g. `file="modelica://MyPackage/MyScript.mos"` if a script file MyScript is located in the top level of the top package MyPackage. Such an URI is case sensitive.

Call function associate Modelica functions with the current model in the form of `executeCall=foo()` or `editCall=bar()`. Both of these calls the given function instead of running the file, and editCall allows the caller to modify the function arguments before calling the function. By using the arrow in the end of the input field, you can access any command in the command history from the command log.

It is possible to execute a function without displaying the function dialog if the function has no input parameters, or if all input parameters have default values. This is obtained by adding the annotation `__Dymola_autoExecute=true` to the function. Notes:

- This functionality is not supported when calling functions in any other way.
- The annotation is best seen as "pressing **OK** automatically" rather than "skip launching the function call window" – the adding of current model-name to the function is still handled.
- For advanced users, note that you can override the annotation by going to the command log, right-click the function call, and select **Edit Function Call**.

Plot window setup defines the current plot window setup, including variables shown, as a new command. By activating **Use standard Modelica plot setup**, you can store the plot window setup as a standard `Figure` item in the `Documentation` annotation of the model.

Note that you can also use the command **Script Editor > Generate Script**, selecting **Plot setup** and entering a name in **Store in model as command** to create a command - however without the option to adapt to the proposed Modelica standard. Another difference is that with that solution, you also create a corresponding script file. In addition, you have an option to read from different result files. For more about the command **Script Editor > Generate Script**, see section "Script Editor > Generate Script" starting on page 833.

Animation window setup defines the current animation window setup as a new command. Note that you can also use the command **Script Editor > Generate Script**, selecting **Animation setup** and entering a name in **Store in model as command** to do the same thing. The difference is that with that solution, you also create a corresponding script file. For more about the command **Script Editor > Generate Script**, see section "Script Editor > Generate Script" starting on page 833.

Demands on the state of the model

No pre-condition The command/script can be executed independent on what is the state of the model.

Translated The model must be translated before the command/script can be executed. If the model is not translated when the command/script is selected, the model will automatically first be translated, then the command/script will be executed.

Simulated The model must be simulated before the command/script can be executed. If the model is not simulated when the command/script is selected, the model will automatically first be simulated, then the command/script will be executed.

How the command should be executed or used

Automatically run after simulation means that the command will automatically run after simulation, typically useful for, for example, a plot window setup command, to immediately have a usable plot displayed.

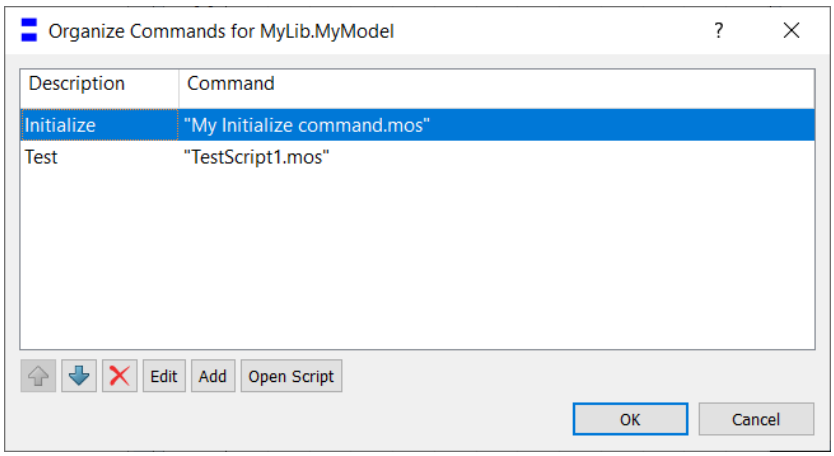
Execute when checking package specifies that the commands should be part of check. Please see indext entry "*check : model –with simulation*" for more information on how to perform such a check.

Prompt for arguments. If this checkbox is checked, the function starts by popping up a dialog where the input values for the function can be modified.

Inherit command allows extended classes to inherit the command. This requires that the function/script does not explicitly use the name of the model.

Simulation > Commands > Organize Commands

Organizing commands.



You can sort, delete, edit and add commands by the five first buttons in the window. The **Edit** button calls the dialog in the previous figure for the selected command. You can also double-click a command to enter this dialog.

The last button, **Open Script**, opens the script editor for the selected command, if it is a .mos script. This makes it easy to change the script without having to find it in the folder structure.

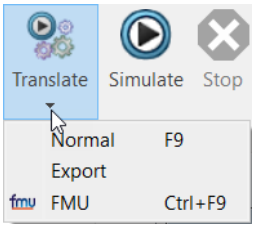
Keys can be used for some actions as well:

- Move up: **Ctrl+Up**
- Move down: **Ctrl+Down**
- Edit: **Space**
- Add: **Plus**

Simulation > Translate



Clicking on the **Translate** button translates the model according to the default below. Clicking on the down-arrow displays the alternatives:



Simulation > Translate > Normal

Translates the active model to simulation code. Error messages and warnings are shown in the log window. After successful translation, the class is ready for simulation.

(Note that you can add the button  in the top left toolbar of the Dymola window – that button is a shortcut for this command. See “Shortcut to Simulation tab commands in the top left toolbar” on page 791 for details.)

There are some settings to control what messages should be displayed when translating. Please see the Translaton tab section of the command “Simulation > Setup” starting on page 799.

The “normal” translation is default when clicking the **Translate** button in the **Simulation** tab.

Note that it is possible to find unused parameters when translating, if setting the flag `Advanced.Check.WarnAboutUnreferenced=true`. (By default, the flag is `false`.)

Note also that there is a flag `Advanced.SourceCodeExportNormal` that can be set to true to obtain the same features for normal translation as for the next command. For more information, see the chapter “Simulation Environments”, section “Code and Model Export”. You can also use the index entry for this flag in the index in this manual.

Simulation > Translate > Export

Translates the active model to code executable on any platform without a Dymola license at the target system. The command corresponds to the built-in function `translateModelExport`.

The “export” simulation must be selected by using the drop-down menu of the **Simulation > Translate** command. See image above.

This command demands license. For more information, please see the chapter “Simulation Environments”, section “Code and Model Export”.

Simulation > Translate > FMU

Translates the active model to an FMU, with the export settings specified in the FMI Export tab of the simulation setup, reached by the command **Simulation > Setup…**. See section "FMI Export tab" on page 822 for information of settings available and other information.

Simulation > Simulate



Simulates the model for one simulation period according to the specified simulation setup. If the active model has not been translated or an editing operation has invalidated the previous translation, then the model is automatically translated before simulation.

(Note that by default the button  in the top left toolbar of the Dymola window is a shortcut for this command.)

Simulation > Stop



Interrupts the execution of a translation, command script or simulation. An active stop button indicates that one of these operations is in progress.

(Note that by default the button  in the top left toolbar of the Dymola window is a shortcut for this command.)

Simulations normally run to completion, but in some cases it is necessary to stop the simulation – either because the setup was incorrect (e.g. stop time 1e10 instead of 10) or because the simulation is progressing too slowly.

Simulations are normally stopped in a nice way in order to ensure that the user gets a complete result file including diagnostics (see "Diagnostics for stuck simulation" starting on page 1002). However, in extreme examples a model might be stuck in an infinite loop.

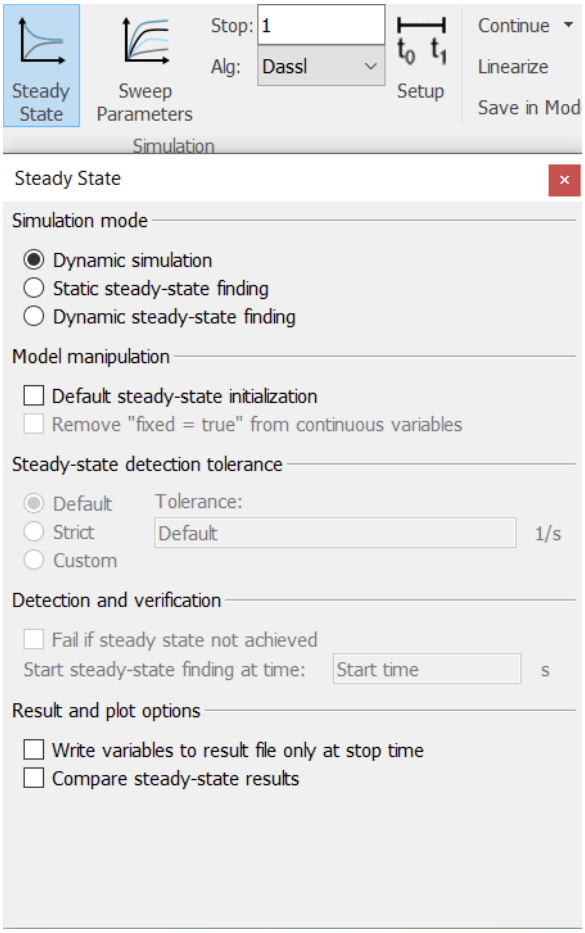
After waiting for a half a minute, the simulation process (dymosim) will be terminated. The waiting period is used to ensure that the dymosim process is terminated in a nice way if possible, even for larger examples.

Note that this command is also available in the top left toolbar of the Dymola window. See Shortcut to Simulation tab commands in the top left toolbar" on page 791.

Simulation > Steady State



Opens the Steady State dialog:



For more information about this dialog and steady state alternatives, see section “Steady state solver interface” starting on page 1029.

Simulation > Sweep Parameters



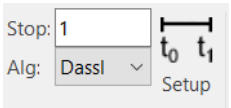
Opens the dialog for sweeping parameters. For more information about sweeping parameters by this command, see the chapter “Model Experimentation”, section “Sweeping parameters using new GUI”.

Simulation > Setup



Setup opens a dialogue for specifying experiment name, simulation interval, integration algorithm etc.

Note that the stop time and algorithm can be changed directly (to the left of the **Setup** command), wihout entering the simulation setup:



(Note also that you can add the button  in the top left toolbar of the Dymola window – that button is a shortcut for this command. See “Shortcut to Simulation tab commands in the top left toolbar” on page 791 for details.)

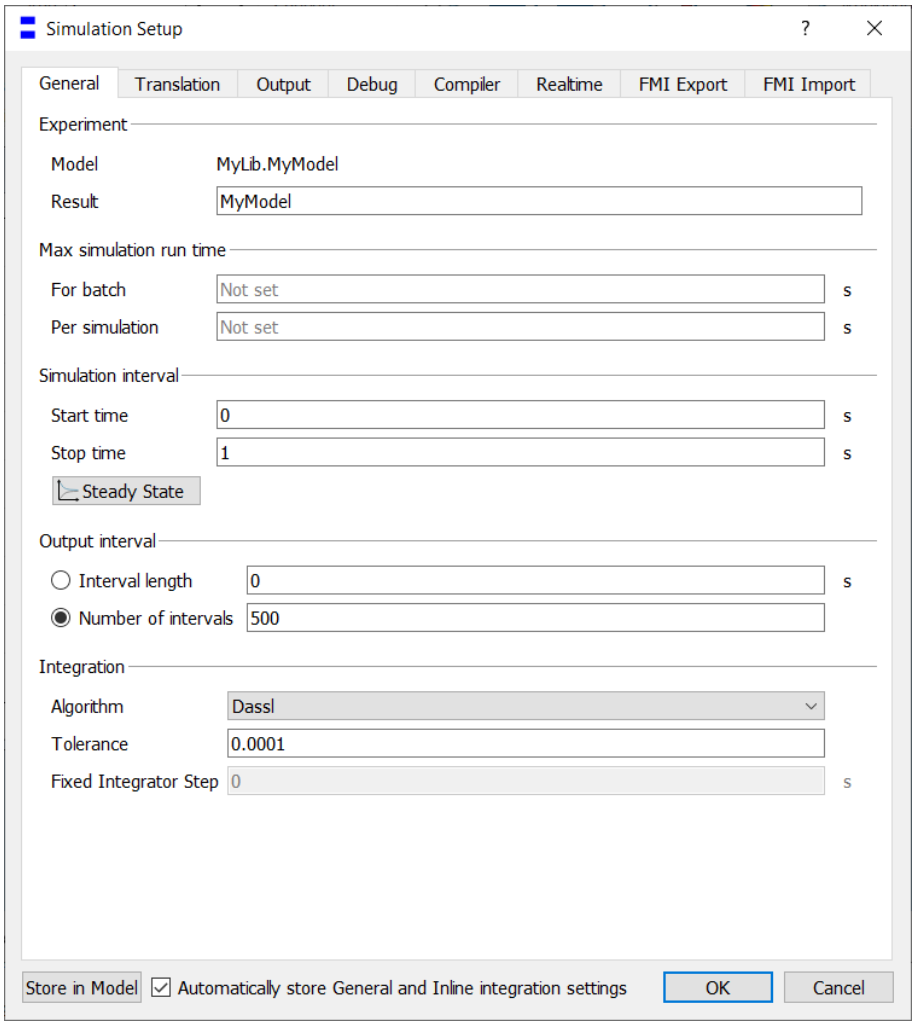
Selected settings from the simulation setup menu can also be stored in the current model by clicking the button **Store in Model**. This only applies to the **General**, **Translation**, **Output**, and (for one setting), the **Debug** tab. When the model is later selected, these settings appear in the simulation setup dialog and can be changed before simulation.

By default, changes in the settings from the **General** tab and the inline settings group from the **Realtime** tab are always stored to the model when clicking **OK** in the menu. See these tabs below for more information.

A number of settings are stored between sessions. In the images of the different tabs below, such settings are hightlighted in green.

General tab

General tab of Simulation > Setup



The **General** tab includes:

The Experiment group.

An **Experiment** group where **Model** specifies the name of the active simulation model (the experiment). If the name is in boldface, you have selected this model as the active simulation model. If the name is in boldface, you can reset the selection by clicking a button after the text box. **Result** is the name of the result file. By default, the name of the active simulation model is used, but you can change this name.

The Max simulation run time group.

A **Max simulation run time** group to specify max run time for all simulations or individual simulations.

You can use **For batch** to set Dymola to stop all simulations after a given positive wall-clock time. The default setting Not set corresponds to the flag

`Advanced.Simulation.MaxRunTime = 0.0`, meaning that there is no limitation on the run time. Notes:

- The wall-clock time is measured, not the CPU time.
- The run time unit is independent of the simulation time unit.
- The option works for both single simulations and batch runs.
- Because the max time is checked by Dymola, the feature works even if the simulation is stuck in code that is not produced by Dymola, such as code of an imported FMU or external C code.
- Changing the value of this timer does not require re-translation of the model.

If you use the **Per simulation** option, each simulation gets its own timer. Each simulation starts its own timer when it starts the simulation and stops itself when this timer runs out. This option may be useful if a few of the simulations in a parameter sweep takes very long time and eventually fails. Then these can be automatically stopped early, without affecting the other simulations. Notes:

- The above notes are valid also for **Per simulation**, except that changing of any timer value *do* require re-translation of the model.
- The option is embedded in the simulator, including an exported FMU, and is therefore independent of Dymola running the simulation.

The Simulation interval group.

A **Simulation interval** group where **Start time** and **Stop time** for the simulation can be specified. For the **Simulation time unit** button after the input lines, see below. Note that the stop time can be changed without going into the simulation setup, see above.

By clicking the **Steady State** button, you open the Steady State solver interface dialog. For information about this dialog and the selections possible, see section "Steady state solver interface" starting on page 1029.

The Output interval group.

An **Output interval** group to specify how often results shall be stored. It can be specified in terms of **Interval length** or **Number of Intervals** for the simulation. By default, the results are also stored at discrete events. For the **Simulation time unit** button after the interval length input line, see below.

Note that it is possible to override the output interval of the simulation setup, and to have variable frequency of output points. See section "Variable frequency of output points" on page 783.

For variable step size numerical algorithms, the output interval length normally does not match the step sizes used for simulating the model. These step sizes are instead chosen by the numerical algorithm during simulation. The solution is then interpolated to retrieve the variable values at the output points. See section "Selecting the integration algorithm" starting at page 895 for details. Note that the algorithm can be changed without going into the simulation setup, see above.

Generally, the user´s choice of output interval does not affect the algorithms step size sequence. However, there are a few exceptions:

- To choose the initial step size a variable step size algorithm needs to estimate the scale of the problem. A common strategy is to use the output interval as a reference for the temporal scale.
- The output interval may affect the quality of start guesses used to solve algebraic loops. For details, see section "Initial guesses for nonlinear equations in the model during simulation" starting at page 997.

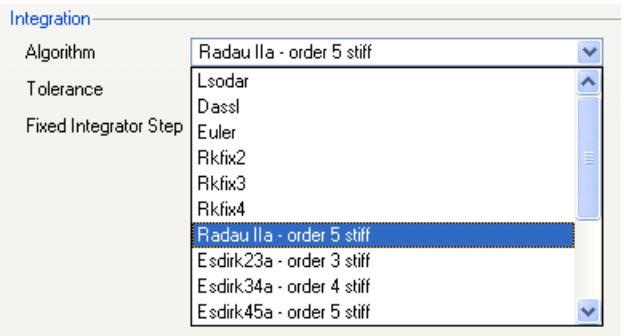
The Integration group.

An **Integration** group that specifies **Algorithm** to be used to solve the differential equations and **Tolerance** specifies required accuracy. **Fixed integrator step** is specified for fixed step integrators such as Euler. For the **Simulation time unit** button after the fixed integrator step input line, see below.

A number of the algorithms/methods are "traditional" (Euler, DASSL etc.) while a number of algorithms/methods are newer. The main user benefit with the newer ones is better restart after events, the addition of non-stiff integrators with variable step size, and solvers better suited for poorly damped stiff systems (i.e. with poles close to the imaginary axis).

The new algorithms are included in the integrator setup making it straightforward to switch between the traditional dymosim integrators and the new ones (marked with order and the algorithms applicable to stiff systems are marked with "stiff" – the first one is "Radau …").

Some algorithms.



Changing between the new solvers and the normal solvers will cause a recompilation (but not a complete retranslation). Linearizing a model will use the dymosim solvers and might thus cause a recompilation. For a user this implies that one should preferably select the integration algorithm before pressing **Translate** (or start by pressing **Simulate**).

Notes:

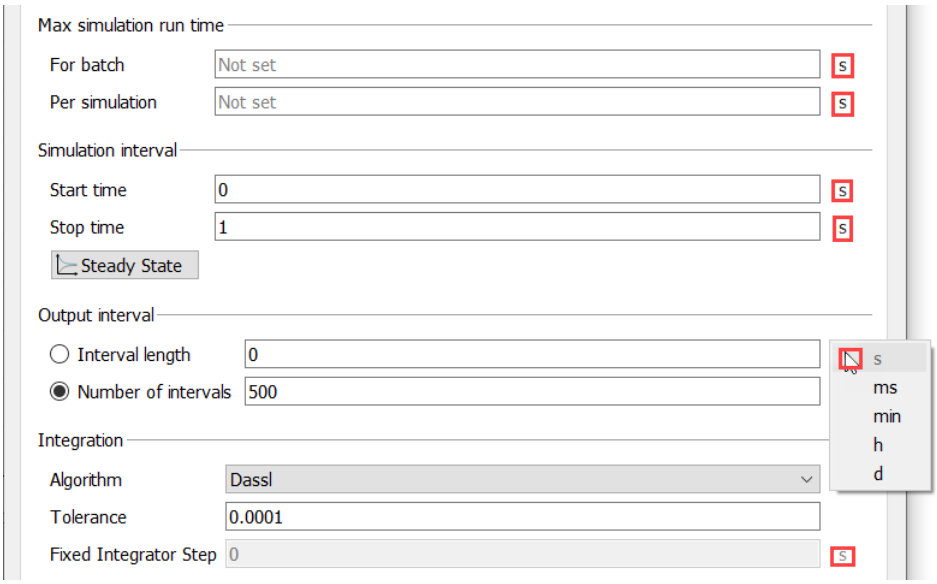
- DAE (Differential-Algebraic Equations) mode is supported for the Dassl, Radau, Esdirk*, and Sdirk34hw solvers. By default, this mode is not activated.
- Sparse solvers are supported for the Lsodar, Dassl, Cvode, Radau, Esdirk*, and Sdirk34hw solvers. If the model is suitable for sparse solvers, a message will appear in the translation log about that, and how you activate it.

More information about the algorithms, including the DAE mode and the sparse solver, is given in the section "Selecting the integration algorithm" starting on page 895.

Note that there are, for advanced users, some additional flags that are related to the step size and event handling, see section "Advanced setup for simulations by flags" on page 782.

The Simulation time unit button.

The **Simulation time unit** button can be used to select the simulation time unit for max run time, start time, stop time, interval length, and fixed integrator step by clicking this button after the input line of any of these values, and then selecting what time unit to use. The present alternatives are:

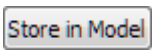


The time units are read from the file `displayUnit.mos`. The simulation time units are only display units, internally seconds are still used.

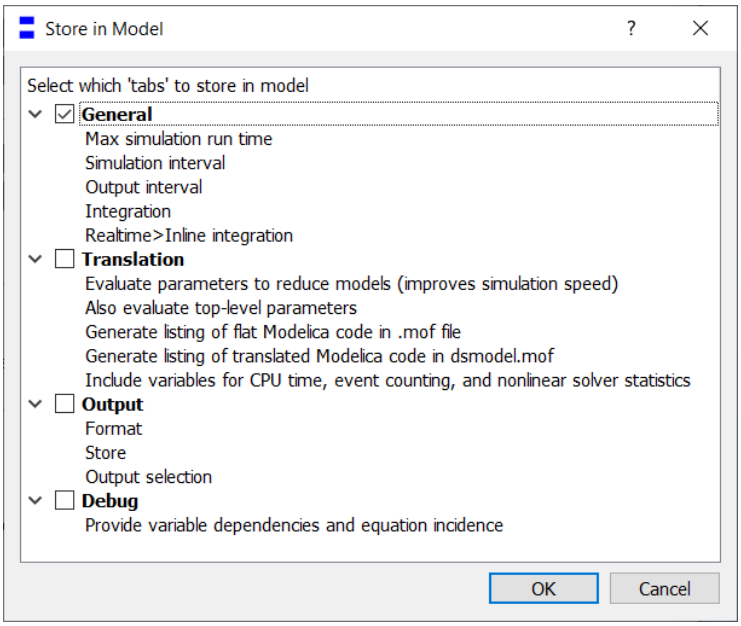
If a new simulation time unit is selected, this selection applies for all (start time, stop time, interval length, and fixed integrator step). The values are changed to match the new unit.

Note that the simulation time unit also changes the time unit used in the active plot, and are used also when new plots are created. See section “Changing time unit on x-axis” 711 for details.

The simulation time unit can also be changed from the animation toolbar. See section “Simulation time unit” on page 832.



The information from this tab can also be stored in the current model by clicking the button **Store in Model** and selecting the relevant tab(s) to store settnings from. When the model is later selected, these settings appear in the simulation setup dialog and can be changed before simulation. Clicking **Store in Model** also displays the settings that are saved for each tab if selected:



Note that if Euler has been selected as algorithm, some settings from the **Realtime** tab are also saved if the **General** tab is saved:

- Inline integration method
- Runge Kutta order
- Optimize code for step size.

By default, the **General** tab is selected to be stored in this menu.

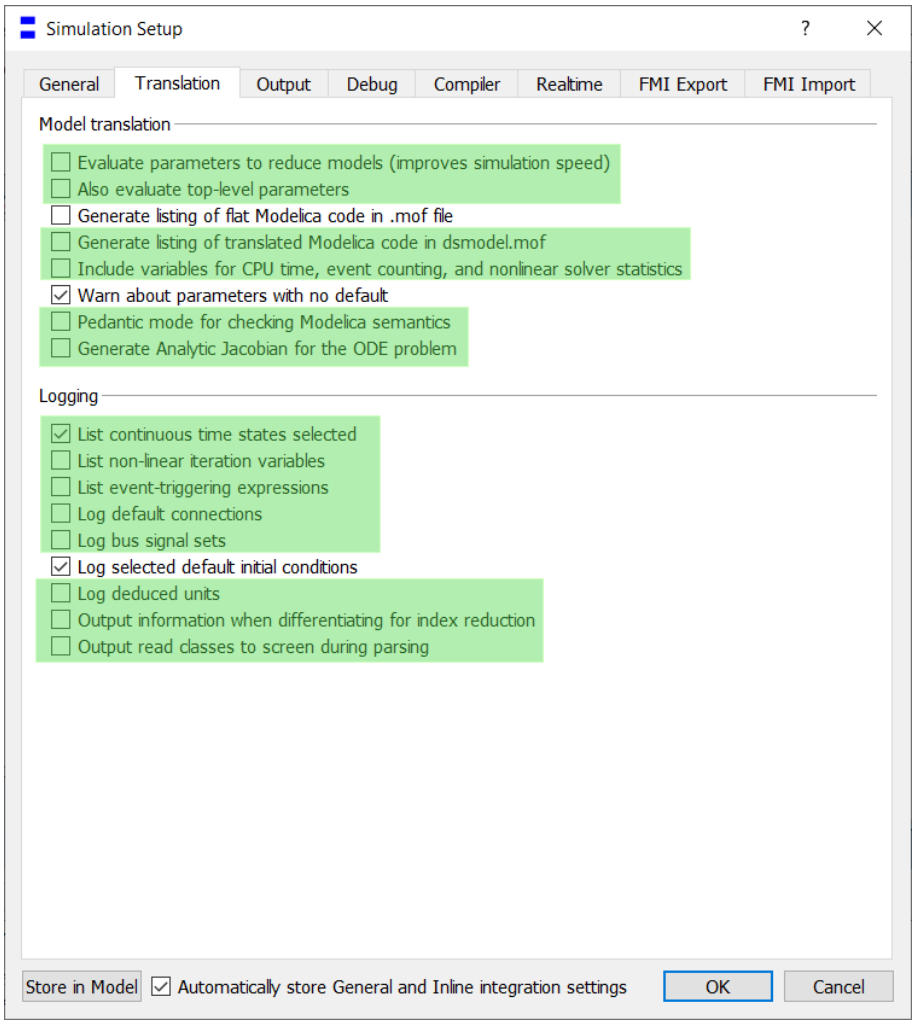
The above information is also by default always stored in the model because of the general setting **Automatically store General and Inline integration settings**. In practice this means that you don´t need to use the **Store in model** button if you *only* want to save these settings; they are automatically saved to the model when you click **OK**. Notes:

- If you for some reason untick **Automatically store General and Inline integration settings**, note that this change is not saved between sessions.
- Changes of simulation settings by scripting are not saved by this feature.

(No settings from this tab are saved between sessions.)

Translation tab

Translation tab of Simulation > Setup...



Evaluate parameters to reduce models (improves simulation speed) is a global setting to have all parameters except top-level parameters evaluated at translation in order to generate more efficient simulation code. These parameters cannot be set interactively before a simulation run.

Top-level parameter records are handled in the same way as simple top-level parameters (they are not evaluated when you specify that parameters should be evaluated).

However, you can use the second setting **Also evaluate top-level parameters** to also evaluate the top-level parameters.

For more information about evaluation of parameters (e.g. the possibility to set evaluation selectively for certain parameters and to log evaluated parameters) see section "Parameter evaluation" on page 784.

Generate listing of flat Modelica code in .mof file to output a listing of all variable declarations and equations.

Generate listing of translated Modelica code in dsmodel.mof. The result of translating a Modelica model can be listed in a Modelica like representation. The listing is stored in the file dsmodel.mof and is meant to be a more readable version of dsmodel.c. This can be used to e.g. investigate algebraic loops and for other debugging purposes. Please see the section "Output of manipulated equations in Modelica format" on page 987 for more information about this.

Include variables for CPU time, event counting, and nonlinear solver statistics introduces six extra variables that can be used e.g for plotting:

- CPU time, with the accumulated CPU time during simulation. The slope of this signal is an indirect measure of the computational complexity of the simulation.
- EventCounter, which is automatically incremented for each event point. Note that the EventCounter counts the event iterations, which makes it possible to identify where you have many iterations, while the events listed in the simulation log are the actual events. It is perfectly normal if the EventCounter shows a higher value than the sum of the events in the simulation log; you can have more than one iteration for each actual event.
- Nonlinear solution statistics summary variables. These four variables (Calls, Residuals, Iterations, and Jacobians) are created for each nonlinear system in the model. They contain the same information as the one given in the simulation log by activating the setting **Nonlinear solution summary** in the **Debug** tab of the simulation setup, but now it is possible to plot the diagnostics over time to see where the most effort is spent. For more information, see section "Nonlinear solution summary" on page 979.

The setting is by default not activated. The setting corresponds to the flag `Advanced.Translation.OutputCPUtime = false`.

Warn about parameters with no default lists all free parameters not given an explicit value in the model.

Pedantic mode for checking Modelica semantics. Translation (compilation) of Modelica models may generate warning messages if the model does not fully comply with the semantics of the Modelica Language Specification. In most cases, these warnings are non-critical and the model will still be possible to simulate. (The relaxed checking enabled by default makes porting of older models easier.) In order to simplify development of fully compliant Modelica models, switching to a pedantic compiler mode will treat all warnings as errors. This forces the model developer to correct the model to avoid semantic errors. Note that you can perform pedantic check without setting this flag, by using the command (from the **Graphics** tab or the **Text** tab) **Check > Pedantic**. See index entry "*check : model – pedantic*" in the index in the end of this manual.

Generate Analytic Jacobian for the ODE problem If enabled, analytic Jacobians are used during simulation, and also for the initialization. For more information, see section "Using analytic ODE Jacobians" starting on page 1010.

List continuous time states selected lists the continuous time states selected.

List non-linear iteration variables lists the variables appearing in the nonlinear systems of equations, including both iteration variables, initial equations and time-varying variables that the equations depend on. Note that if you activate this setting, the non-linear variables will also be saved in the file `dsmodelIterationSelect.mof` in the working directory when translating the model. This enables looking back on them later.

List event-triggering expressions lists possible event-triggering expressions. This setting is by default not activated, it corresponds to the flag `Advanced.Translation.Log.EventExpressions = false`. For more information, see section "Event logging" on page 985.

Log default connections outputs diagnostics when unconnected connectors receive a default value according to the Modelica semantics. This may help in finding an incorrectly composed model.

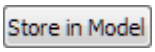
Log bus signal sets outputs information about connection of bus signals expandable, for tracing of bus signals. Please see section "Bus signal tracing" on page 606.

Log selected default initial conditions reports the result of the automatic selection of default initial conditions. If initial conditions are missing, Dymola makes automatic default selection of initial conditions. The approach is to select continuous time states with inactive start values and make their start values active by virtually turning their fixed to true to get a structurally well posed initialization problem.

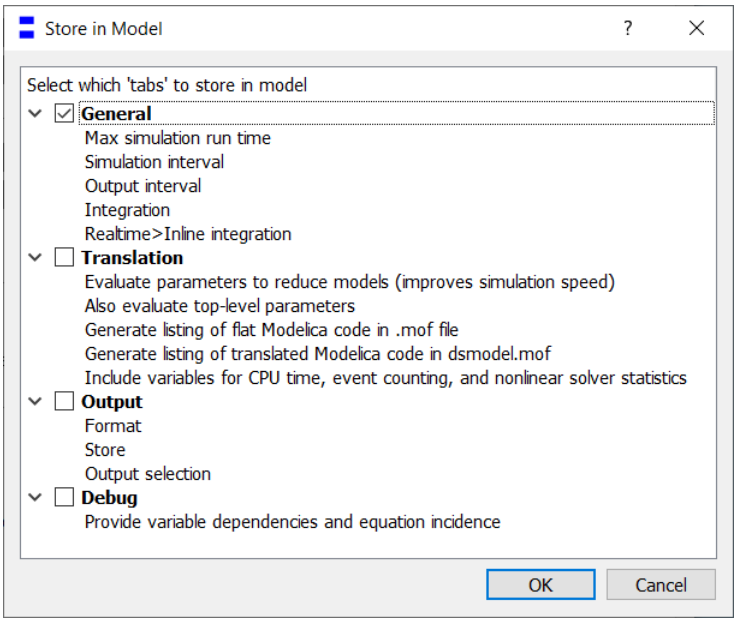
Log deduced units outputs deduced units, see previous chapter, section "Checking the model", sub-section "Unit checking and unit deduction in Dymola".

Output information when differentiating for index reduction reports about which equations that are differentiated.

Output read classes to screen during parsing reports on the read classes during parsing.



Some settings from this tab can also be stored in the current model by clicking the button **Store in Model** and selecting the relevant tab(s) to store settings from. When the model is later selected, these settings appear in the simulation setup dialog and can be changed before simulation. Clicking **Store in Model** also displays the settings that are saved for each tab if selected:



If **Translation** is selected in the dialog above, when clicking **OK** the listed settings from the **Translation** tab will be stored in the model, in the annotation `annotation(__Dymola_experimentFlags())`.

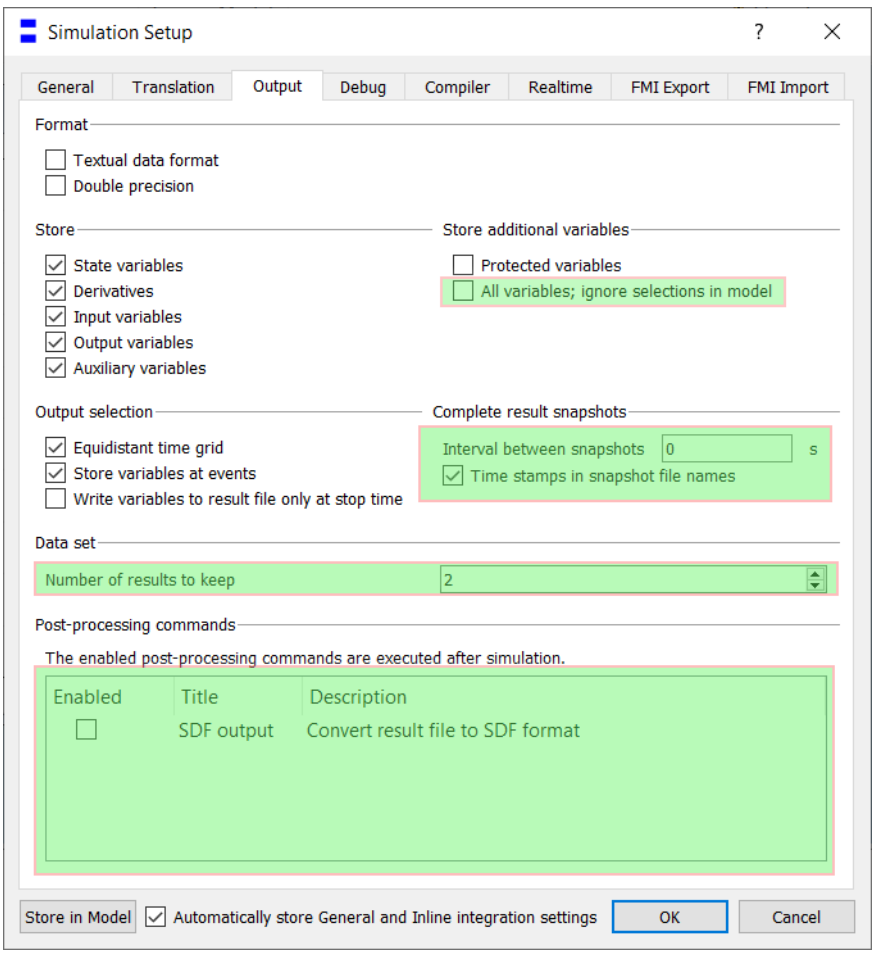
Note that you can insert additional flags by typing them inside this annotation. They are applied when translating the model, and automatically restored afterwards. This means that the values of such flags cannot be seen outside the annotation, that is, as an example, if you enter the flag `Advanced.Translation.Log.ExternalResources=true` in the annotation, you still see the value of the flag as `false` if you, for example, type `Advanced.Translation.Log.ExternalResources` in the command input line of the command window and press **Enter**.

For information about the general setting **Automatically store General and Inline integration settings**, see the **General** tab above.

The settings highlighted in green are saved between sessions.

Output tab

Output tab of Simulation > Setup...



Textual data format stores result in textual format instead of binary.

Double precision stores results in double precision instead of single precision.

The **Store** group defines which categories of variables to store. Notes:

- For **Input variables** and **Output variables**, only *public top-level* input and output variables can be stored, respectively.
- See also next group; that one is related to the **Store** group, and the last note applies also to the **Store** group.

The **Store additional variables** group specifies additional storage settings to the **Store** group. **Note** that these settings in default state *decrease* the number of variables being saved. If both settings are ticked, variables according to the **Store** group settings are being saved. Unticking any of them decrease the number of variables saved; e.g. **Protected variables** being unticked,

a filter is applied on the variables to be saved according to the **Store** group settings, excluding protected variables from being saved.

- Concerning **Protected variables**, please also see section "Storing of protected variables" on page 786. Please note that variables declared as hidden will not be stored.
- If **All variables; ignore selections in model** is unchecked (default), only variables present in selections (given the **Store** settings), are stored in the result file. If no variable selections exist for a model, the setting has no effect on the number of variables being saved.

Note! The **Store** group and the **Store additional variables** group are dimmed when **Provide variable dependencies and equation incidence** is ticked in the **Translation** tab, the dimmed settnings displays the status when **Provide variable dependencies and equation incidence** was ticked. When this variable is ticked however, *all categories of values are stored, independently what is displayed*. Unticking this variable, the status of what categories were saved is restored and the groups are no more dimmed.

In the **Output selection** group:

- **Equidistant time grid** stores result equidistantly as given by simulation setup.
- **Store variables at events** stores variables before and after events.
- **Write variables to result file only at stop time** can be used to minimize the size of the result file if the model is large and only the result file at stop time is needed. Notes:
    - The output points, as defined by the **General** tab of the simulation setup are still used internally by the integrator
    - This option can be seen as a complement to variable selections.

In the **Complete result snapshots** group:

- **Interval between snapshots** controls the interval between printing snapshots of the simulation result file "dsfinal.txt". The default value is 0.0, which means no snapshots. (The flag `Advanced.Simulation.ResultSnapshotInterval = 0.0` corresponds to the default setting.)
- **Time stamps in snapshot file names**: By default, a time stamp is added to the snapshot file name, e. g. "dsfinal_0.1.txt". By deactivating this setting, no time stamps are added. (The flag `Advanced.Simulation.ResultSnapshotTimeInFileName = true` corresponds to the default setting.)

A message will be printed in the simulation log whenever a snapshot is created, for example

```
…. "dsfinal_0.4.txt" creating (snapshot at time t = 0.4)
```

For more information about using snapshots, see section "Saving periodic snapshots during simulation" starting on page 985.

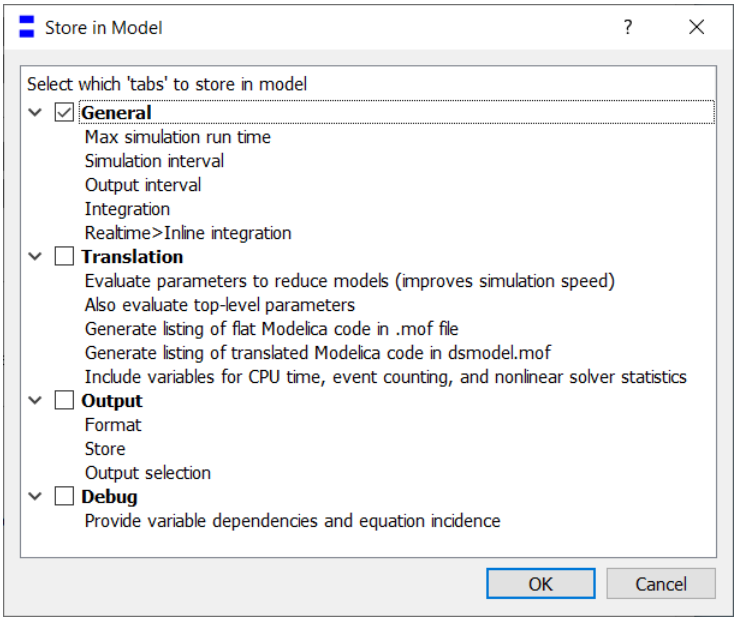
In the **Data set** group, **Number of results to keep** specifies the number of result files kept in the variable browser (default 2). Manually loaded results are not included, nor do result files being selected to keep (user-defined or automatic). Please see section "Selecting which

simulation result files to keep when performing a new simulation" on page 675 for more information about this.

The **Post-processing commands** group is described in section "Post-processing commands" starting on page 787.

Store in Model

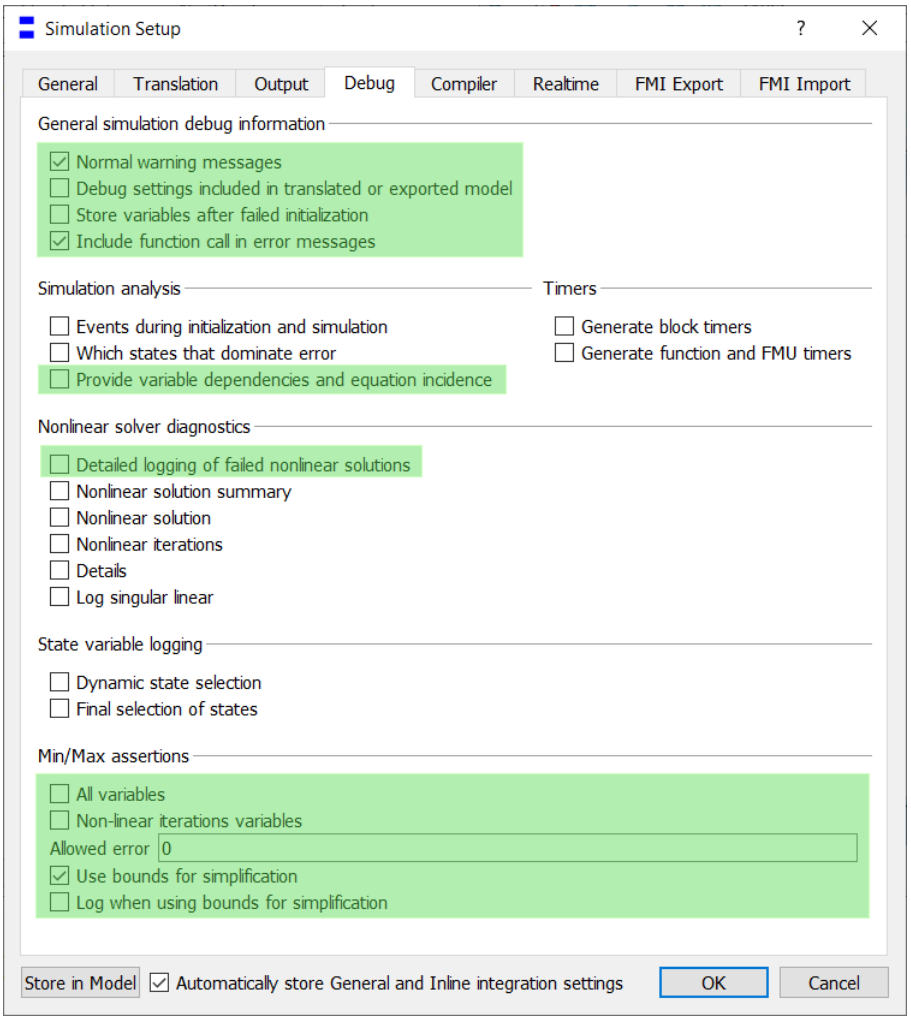
Selected information from this tab can also be stored in the current model by clicking the button **Store in Model** and selecting relevant tab(s) to store settings from. When the model is later selected, these settings appear in the simulation setup dialog and can be changed before simulation. Clicking **Store in Model** also displays the settings that are saved for each tab if selected:



For information about the general setting **Automatically store General and Inline integration settings**, see the **General** tab above.

The settings highlighted in green are saved between sessions.

Debug tab



This tab makes it possible to track down problems during simulation, e.g. chattering, problems with non-linear systems of equations and state-selection. How to use some of these settings are described in more detail in section “Debugging models” starting on page 974.

Normally these settings do not require a new translation, but does on the other hand only take effect when running inside Dymola.

The resulting text is displayed in a separate window. Please note that resulting text is not displayed in the Command log in the Command window. This means that if the resulting text should be saved, it must be done separately.

General simulation debug information group

Note. All settings in this group are saved between sessions.

Normal warning messages If you want to disable all non-fatal warnings and errors.

Debug settings included in translated or exported model Useful if you want to debug the model when not running inside Dymola, for example if the model is exported as an FMU.

Store variables after failed initialization When enabled, the current values of all variables are stored at initialization failure, which means that they can be available in the package browser which in turn makes it possible to plot them. Note that the stored values are not a consistent solution. For usage of this flag, see "Debugging failed initialization" starting on page 1027.

Include function call in error messages – when enabled, errors in functions will report the call with variable names – making it easier to fine the troublesome spot in the mode.

Simulation analysis group

Events during initialization and simulation Log events during the initial phase and during the simulation. This feature is useful for finding errors in the event logic including chattering. By default, this setting is not activated. This setting corresponds to the flags

- `Advanced.Debug.LogEvents=false`
- `Advanced.Debug.LogEventsInitialization=false`

By changing the value of these flags, it is possible to, for example, only log events during simulation, and not during initialization.

To fine-tune the loggig of events during the simulation (not the initialization) even more, you can use the following flags to disable logging of time events, state events, or step events:

- Set `Advanced.Simulation.Debug.LogTimeEvents=false` to disable logging of time events during the simulation.
- Set `Advanced.Simulation.Debug.LogStateEvents=false` to disable logging of state events during the simulation.
- Set `Advanced.Simulation.Debug.LogStepEvents=false` to disable logging of state events during the simulation.

All flags are by default `true`. You must set the flags before simulating. The choise affects both the simulation log and event log in the Simulation Analysis feature. Changing any of these flags forces re-translation of the model.

An example of a use-case is simulations with frequent sample events resulting in large logs. Disabling time event logging makes it easier to find the state events.

As told, the events during initialization are not filtered with any of the tree last flags above, to filter out events during initialization you can use the flag `Advanced.Debug.LogEventsInitialization=false` mentioned earlier. (The idea with the three last flags is to exclude based on what triggers the event during simulation; if the triggering event is a time, state, or step event. As initialization is none of these three categories, events during initialization cannot be filtered by these three flags.)

Which states that dominate error - if the simulation is slow due to tolerance requirements for some state variable this can help to investigate which variables are the critical ones. Setting

this option also enables analyzing numerical integration; see section "Analyzing numeric integration" starting on page 649.

Provide variable dependencies and equation incidence To be able to plot dependencies for a variable, and to plot equation incidence, this setting must be ticked. The setting corresponds to the flag `Advanced.GenerateVariableDependencies`, and is by default unticked (false). Note that this setting also automatically activates storing of all types of variables, corresponding to the settings in the two groups **Store** and **Store additional variables**, in the **Output** tab (see next section), *independently* wheter those settings are seen as ticked in that tab or not. To indicate this, these two groups will be dimmed when **Provide variable dependencies and equation incidence** is ticked, the dimmed settnings displays the status when **Provide variable dependencies and equation incidence** was ticked. Unticking this variable, that status is restored and the groups are no more dimmed.

Timers

Generate block timers This setting activates generation of block timers. It corresponds to the flag `Advanced.GenerateBlockTimers`. For more information, see section "Profiling" starting on page 1017.

Generate function and FMU timers This setting activates generation of function and FMU timers. It corresponds to the flag `Advanced.GenerateFunctionTimers`. For more information, see section "Profiling" starting on page 1017.

Nonlinear solver diagnostics

These settings can be used for finding problems with non-linear system of equations, e.g. ill-posed problems, or missing noEvent (see the chapter "Advanced Modelica Support", section "Using noEvent"). In general, except for the first and last setting, the idea is to start enabling the setting **Nonlinear solution summary**, if this one does not give sufficient information, also enable the next setting, and so on. For more information, see section "Controlling the amount of diagnostics for nonlinear solvers in the simulation log" on page 977.

Detailed logging of failed nonlinear solutions controls the amount of information given in the simulation log about failed nonlinear solutions.

Nonlinear solution summary – at the end of the simulation print a summary of nonlinear solver statistics in the simulation log. Note the option to plot the summary variables. See the above reference.

Nonlinear solutions – log solutions of non-linear systems.

Nonlinear iterations – log iteration of non-linear systems.

Details – log iteration of non-linear systems in detail.

Log singular linear – log if linear systems are consistently over-determined.

State variable logging

Dynamic state selection logs the changes to the selection of states. The format of the logged changes is similar to the format of making a fixed state selection when editing a model. (For more information about state selection please see the chapter "Advanced Modelica Support", section "Means to control the selection of states".)

Final selection of states logs the state variables used at end. The format of the logged changes is similar to the format of making a state selection when editing a model.

Min/Max assertions group

For an example of usage of the first three settings, please see the section "Bound checking for variables" on page 1000. For an example of usage of the two last settings, and related settings not available in the GUI, please see section "Simplifying if-then-else expressions by using min/max of variables" on page 1009.

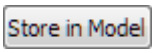
All variables dynamically checks that variables are within the min and max bounds when checked.

Non-linear iteration variables checks that non-linear iteration variables are within the min and max bounds when checked.

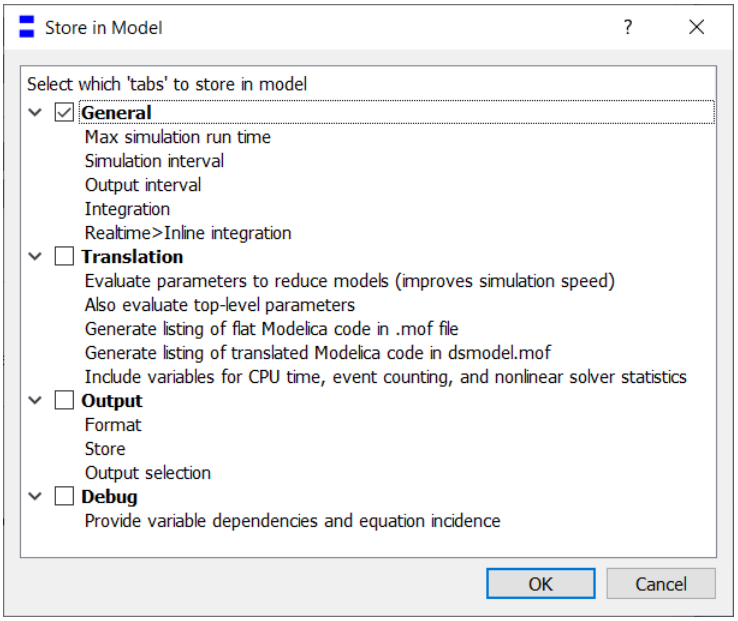
Allowed error Maximum allowed error when checking min/max values, e.g. 1e-6.

Use bounds for simplification Use the min/max of variables to simplify if-then-else expressions.

Log when using bounds for simplification Log the simplifications of if-then-else expressions.



Selected information from this tab (one setting) can also be stored in the current model by clicking the button **Store in Model** and selecting relevant tab(s) to store settings from. When the model is later selected, these settings appear in the simulation setup dialog and can be changed before simulation. Clicking **Store in Model** also displays the settings that are saved for each tab if selected:

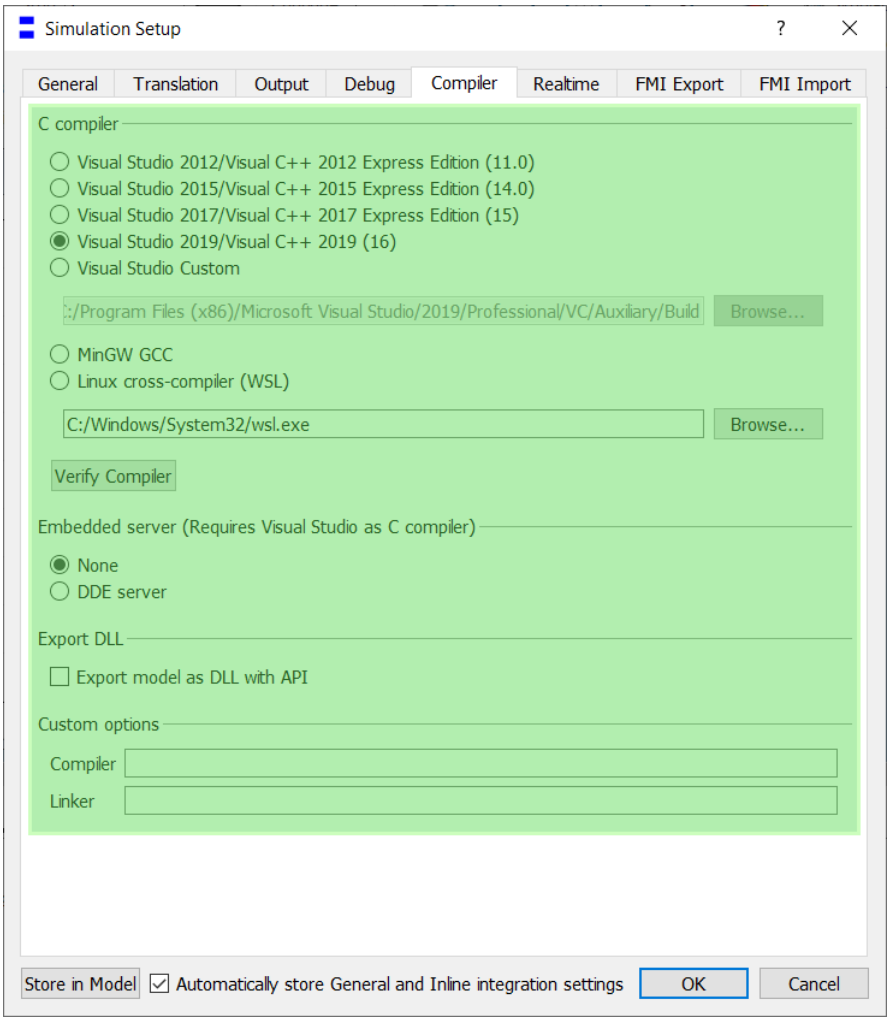


For information about the general setting **Automatically store General and Inline integration settings**, see the **General** tab above.

The settings highlighted in green are saved between sessions.

Compiler tab of Simulation > Setup...

Compiler tab



Allows the user to change the compiler used by Dymola to compile simulation code into an executable program that performs the simulation.

Note that you can use the built-in function `GetDymolaCompiler` to retrieve the current compiler settings (except the custom options ones in the dialog). See section “GetDymolaCompiler” on page 941. There is also a corresponding built-in function for specifying the settings, `SetDymolaCompiler`.

C compiler group

Dymola uses a C compiler to compile the generated C code into a simulation program. The C compiler is not distributed with Dymola; the user has to install a supported C compiler. For more information for Dymola on Windows, please see the chapter “Appendix – Installation”,

section “Installation on Windows”, sub-section “Installing a C compiler”. Note that if you select **MinGW GCC** as compiler, the menu is expanded to display more selections.

To get a small improvement of the simulation performance, you can activate the global optimization in the compiler, by setting the flag

```
Advanced.Define.GlobalOptimizations = 2;
```

before generating code. (The default value of the flag is 0.)

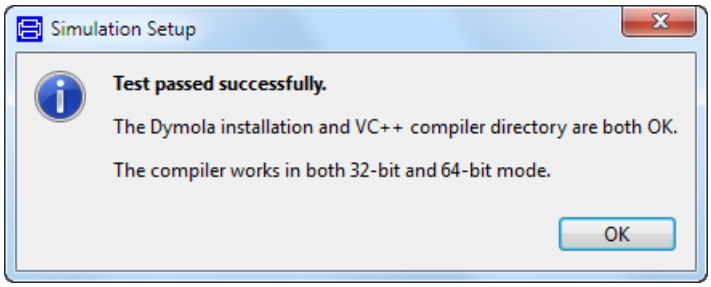
This flag works the same for all Visual Studio compilers. Note that although the simulation performance is somewhat improved setting the flag, the compilation of the code might take substantially longer time for large models. The setting corresponds to the compiler command `/Og`.

Concerning consequences of not writeable directories etc please see the chapter “Appendix – Installation”, section “Installation on Windows”, sub-section “Installing the Dymola software”; “Location of directory”.

On Linux systems a gcc compiler is used, also the clang compiler can be used. The compilation is controlled by the shell script `dymola/insert/dsbuild.sh`. For details, please see the chapter “Appendix – Installation”, section “Installation on Linux”.

Verify compiler

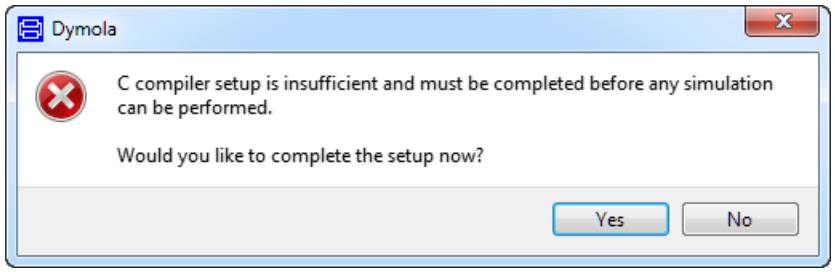
Several different kinds of problems may occasionally arise when compiling the generated C code produced by Dymola. Pressing the **Verify compiler** button performs tests to ensure that the selected compiler is available and can compile a small test model, for both 32-bit and 64-bit mode. If this is the case, the following message appears:



See the chapter “Appendix – Installation”, section “Troubleshooting” for more information.

Compiler check during Dymola startup

During the startup of Dymola, the compiler setting is checked. If insufficient, the following message appears:



If you click **Yes**, the menu for the compiler setup is opened to let you complete the setup.

Embedded server

If real-time simulation should be used under Windows, **DDE server** must be checked. Checking this alternative means that Dymosim will be compiled to provide a DDE server.

For more information about the DDE server, please see the chapter "Simulation Environments".

Export DLL group

If the user has Binary Model Export (or Source Code Generation) option, ticking **Export model as DLL with API** makes it possible to generate a dynamic link library (dymosim.dll) from a model. For more information, please see the chapter "Simulation Environments". Note that this feature will in some later release be replaced by the use of FMI.

Custom options group

Here advanced users can specify options for the compiler and/or linker. They correspond to the string flags `Advanced.LinkerOptions` and `Advanced.CompilerOptions`, respectively. Note that these settings are not handled by the built-in functions `GetDymolaCompiler` and `SetDymolaCompiler`, you have to use the flags.

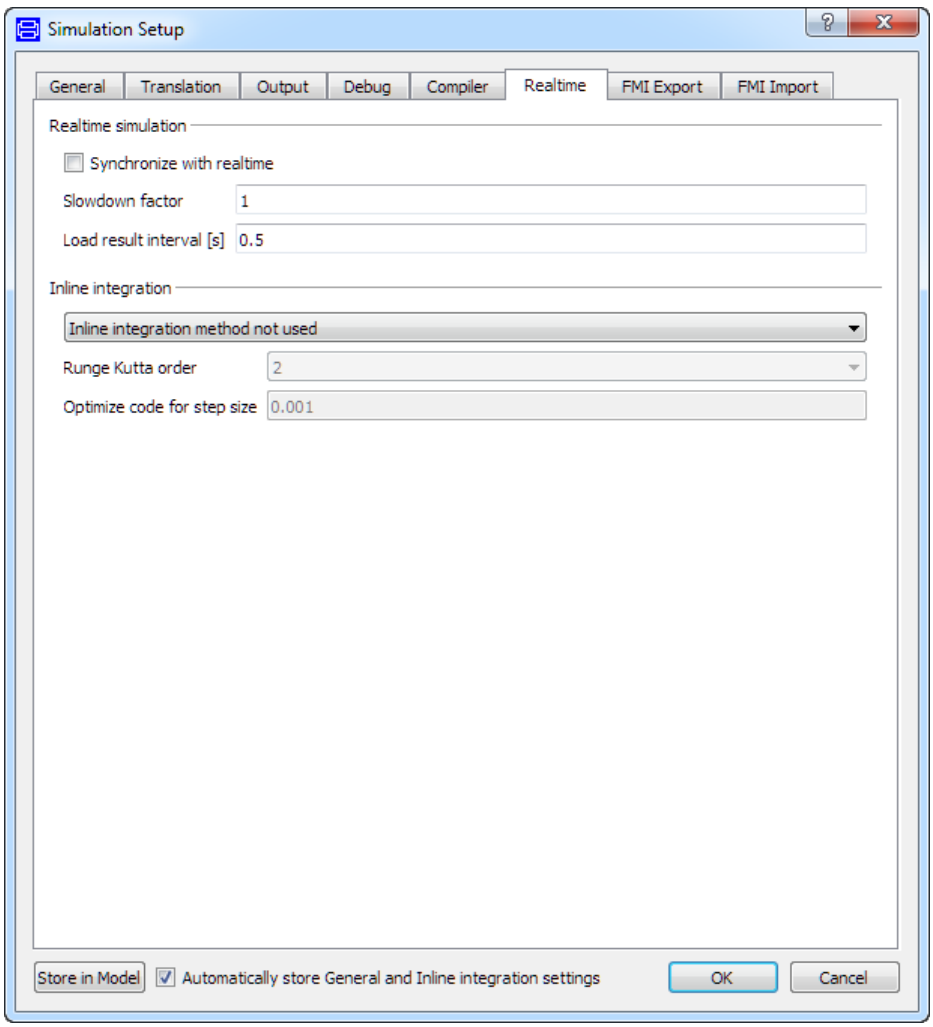
(The **Store in Model** button only applies to the tabs **General**, **Translation**, **Output**, and **Debug**.)

For information about the general setting **Automatically store General and Inline integration settings**, see the **General** tab above.

The settings highlighted in green are saved between sessions, for this tab all settings are saved between sessions.

Realtime tab

Realtime tab of Simulation > Setup...



Activating items in this tab requires the real-time simulation option of Dymola.

Synchronize with realtime This enables (soft) real-time simulation with frequency of about 20Hz running under Windows in Dymola, if the compiler selection is **DDE server**. (For selection of compiler, please see previous section.)

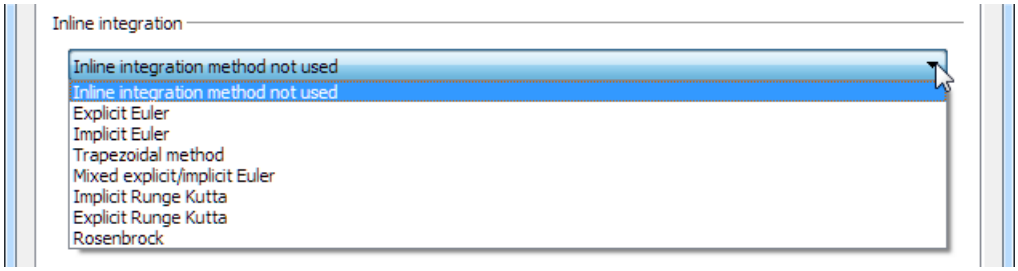
For hardware-in-the-loop simulation see the chapter “Simulation Environments”, section “Real-time simulation”.

Slowdown factor makes it possible to simulate in scaled real-time.

Load result interval [s] - how often the result file should be loaded and written. Online plots are updated this often. **Note** that the default value 0.5 is suitable in most cases. If you set a lower time, this may slow down the speed of the simulation.

Inline integration Select inline integration algorithm/method. The implicit algorithms are important for real-time integration of stiff systems (models with time-scales that are very different). For the higher order algorithms, it is possible to select order, and for all implicit algorithms the code is optimized for one step-size, which should be given here. This step-size should be the same as the fixed-step size when running in Dymola and the same as the real-time step-size when running on other platforms.

Please note that the last statement means that if you select any inline integration algorithm, you must in the **General** tab select Euler and the same step length as the inline algorithm. (If e.g. Trapezoidal inline method is selected, and **Optimize code for step size** is selected as 0.002 you have to in the **General** tab select **Algorithm** Euler and **Fixed Integrator Step** 0.002.) You will get an error message if improper algorithm is selected. If the step length is not explicitly entered, Dymola will calculate a step length from the chosen **Interval length** or **Simulation interval / Number of intervals**, whichever relevant.



For the three last methods, the order can be set between 1 and 4. (Setting order 1 for Implicit/Explicit Runge Kutta actually means using Implicit/Explicit Euler.)

More information about this can be obtained in the section “Inline integration” starting at page 1021.

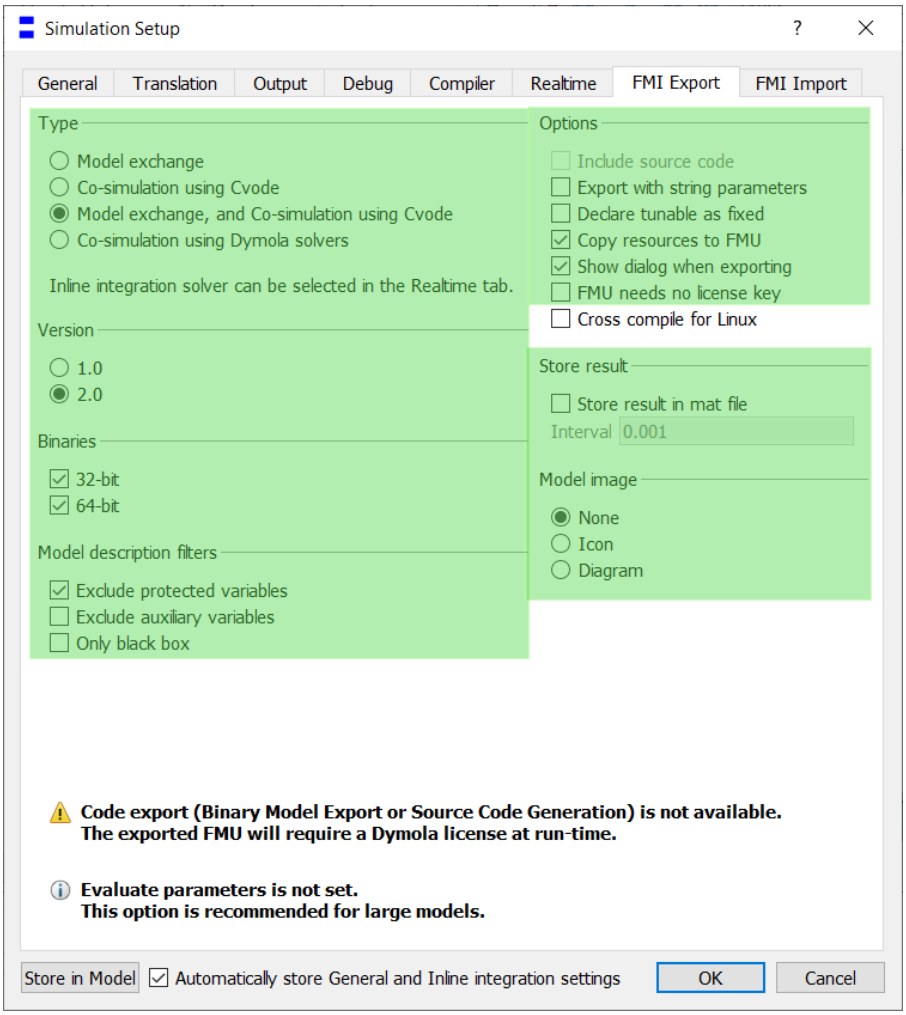
(The **Store in Model** button only applies to the tabs **General**, **Translation**, **Output**, and **Debug**.)

For information about the general setting **Automatically store General and Inline integration settings**, see the **General** tab above.

No settings in this tab are saved between sessions.

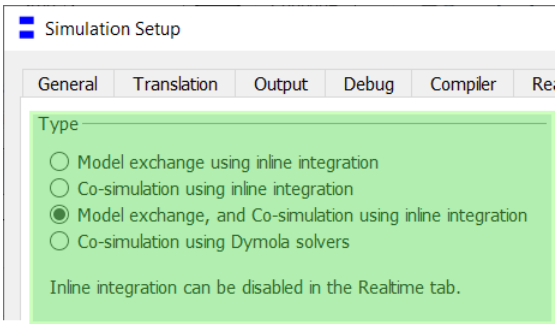
FMI Export tab

FMI Export tab of Simulation > Setup... – no inline integration

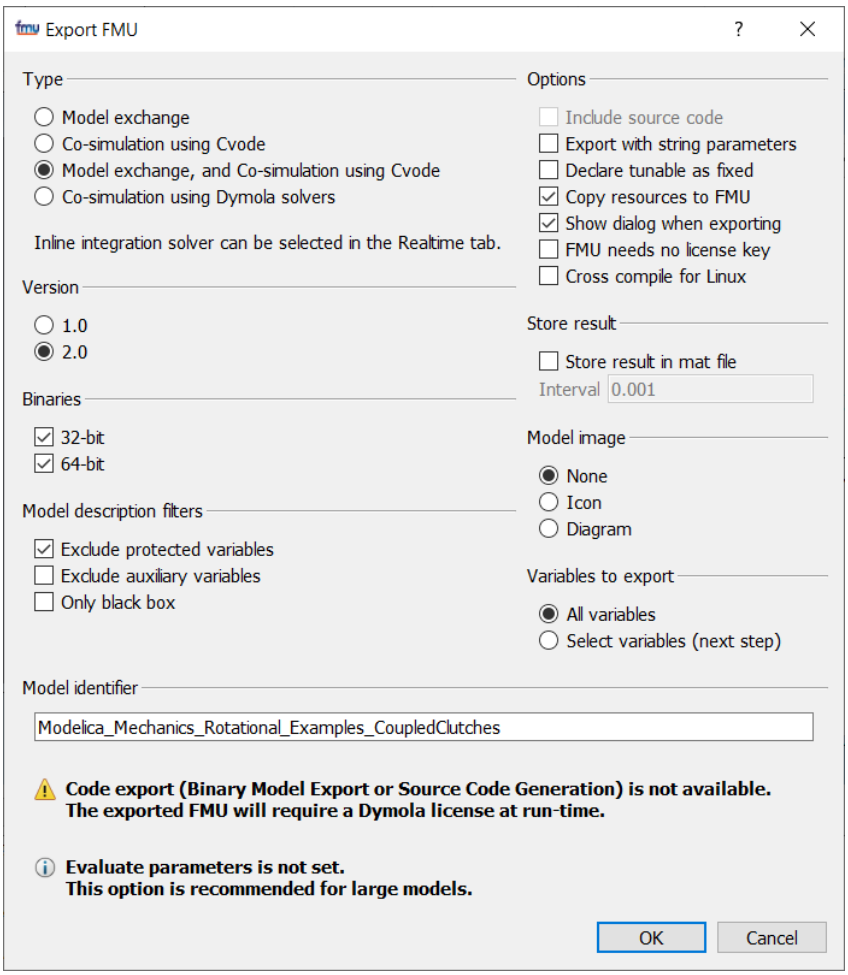


The above image is displayed when inline integration is not selected. If inline integration is selected, the **Type** group is changed to:

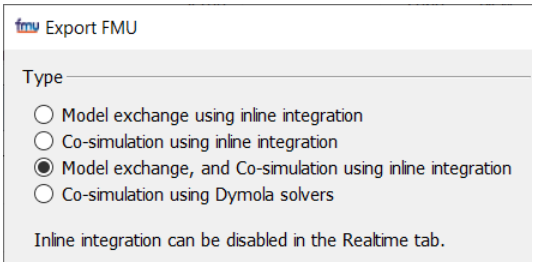
FMI Export tab of Simulation > Setup... - inline integration



The settings of this tab will also appear when exporting an FMU by the command **Simulation > Translate > FMU**.



The above image is displayed when inline integration is not selected. If inline integration is selected, the **Type** group is changed to:



For more information, including flags for scripting, please see the chapter "FMI, eFMI, and SSP Support in Dymola".

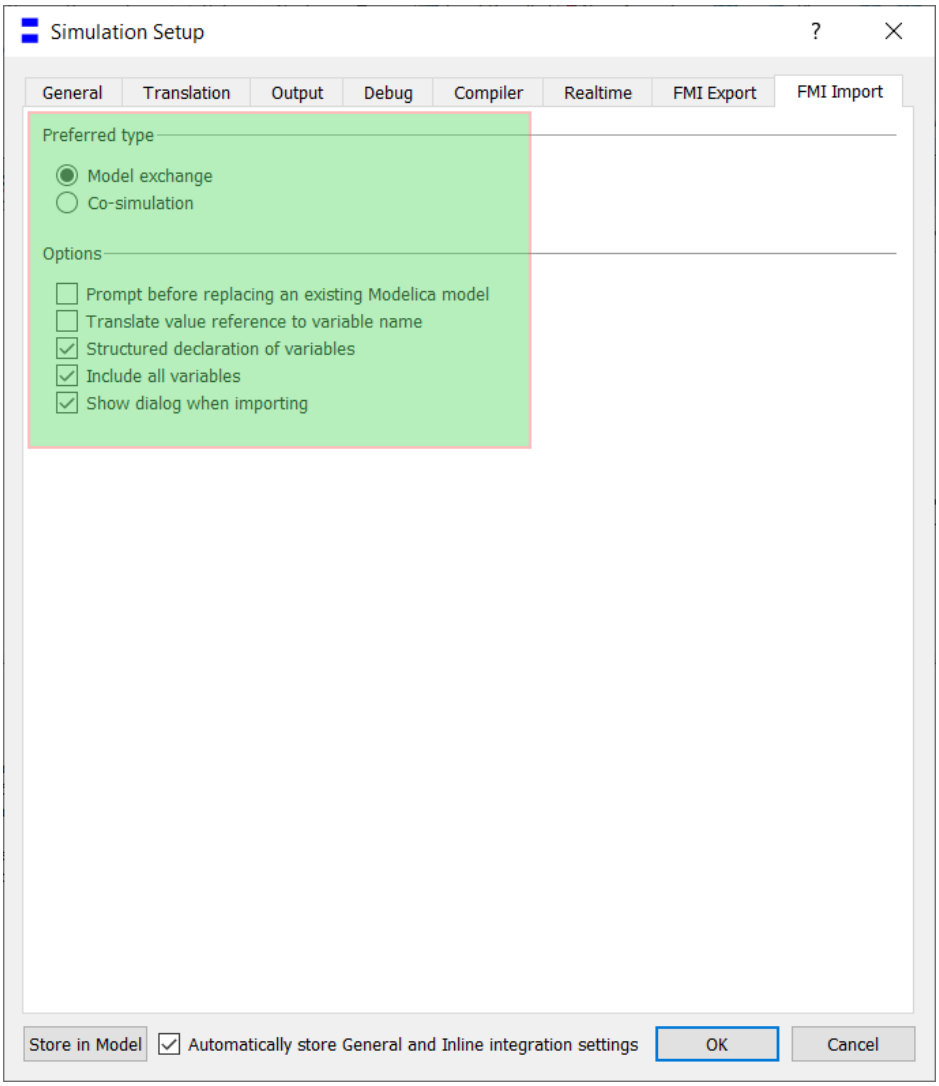
(The **Store in Model** button only applies to the tabs **General**, **Translation**, **Output**, and **Debug**.)

For information about the general setting **Automatically store General and Inline integration settings**, see the **General** tab above.

The settings highlighted in green are saved between sessions, for this tab all settings are saved between sessions.

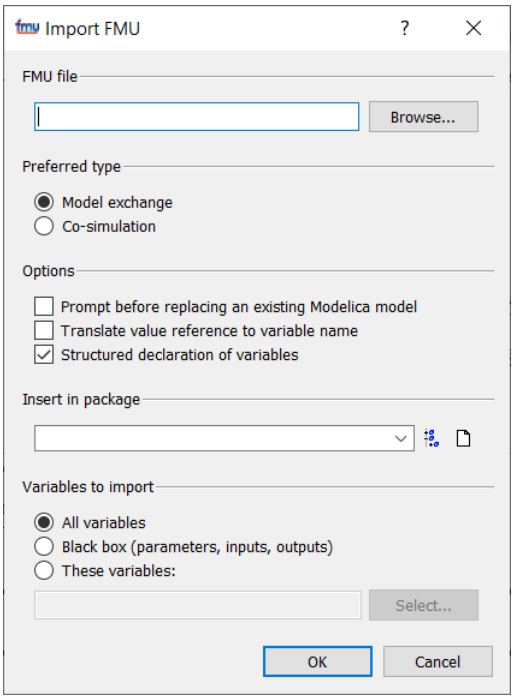
FMI Import tab

FMI Import tab of Simulation > Setup...



For more information about FMI Import in general, and the settings below (including flags for scripting), please see the chapter "FMI, eFMI, and SSP Support in Dymola".

The settings of this tab will also appear in a dialog when using the command **File > Open > Import FMU...** or when dragging an `.fmu` file into the Dymola main window:



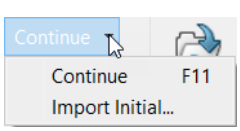
There are some differences in this dialog compared to the tab above. For more information, including flags for scripting, please see the chapter "FMI, eFMI, and SSP Support in Dymola".

(The **Store in Model** button only applies to the tabs **General**, **Translation**, **Output**, and **Debug**.)

For information about the general setting **Automatically store General and Inline integration settings**, see the **General** tab above.

The settings highlighted in green are saved between sessions, for this tab all settings are saved between sessions.

Simulation > Continue



The two alternatives in this menu are:

Simulation > Continue > Continue

Continues the simulation for one simulation period (start time to stop time) from the previous stop time.

Please note the issue concerning models using C-functions to opening external files using the Modelica operator `when initial()` that is described in the next command.

Simulation > Continue > Import Initial…

Loads initial values usually from the end of another simulation (corresponding to **Simulation > Continue > Continue**, see that command) or from the middle of a simulation. The simulation can then continue from this point using **Simulate** (optionally after modifying parameters). In other words, this command allows **Continue** from an arbitrary point in a result file (from the same model).

When using this command, the user is prompted for input file and, depending on what file is selected, start time.

Please note that the variables that should be used must be saved before using the command. This is done automatically if *all* alternatives are checked in the **Store** group in the Output tab in the simulation setup menu that will be displayed using the command **Simulation > Setup**. Note that result snapshots can be used, see section "Saving periodic snapshots during simulation" starting on page 985.

Please note an issue concerning models using C-functions to opening external files using the Modelica operator `when initial()`, i.e. functions other than the table-function used in Modelica that are used in the same way. These will not be run during any **Continue** command. It is recommended to use the predefined Modelica class `ExternalObject` instead.

This command corresponds to the functions `importInitial` and `importInitialResult` (depending on what type of input file is selected). Please see these commands in section "Simulator API" starting on page 939.

Please also note the alternative available by the command **Save Start Values in Model** selecting the option **Store values in new model**. See section "Save start values in model" starting on page 625 for more information about this alternative.

Simulation > Linearize

The command **Simulation > Linearize** translates the model (if not done) and calculates a linearized model at the initial values.

The linearized model is stored in the Dymola working directory in Matlab format as the file `dslin.mat`.

Content of dslin.mat

The file dslin.mat contains four variables: ABCD, Aclass, nx and xuyName.

- ABCD is one matrix that contains the matrices A, B, C and D in the format [A,B;C,D], corresponding to the linearized system

```
der(x)=A*x+B*u
y=C*x+D*u
```

- Aclass contains textual information of no relevance for the vast majority of users.
- nx specifies the number of rows/columns of the A matrix. Knowing the format of the ABCD matrix is (with numbers of rows/columns added outside the matrix in this drawing):

	nx	nu
nx	A	B
ny	C	D

  the user can easily divide ABCD into A, B, C and D if the matrix is not too big.
- xuyName specifies the variables of the linearized system above in the order x, u and y variables.

Handling of dslin.mat

The content of the dslin.mat file can be read in Dymola by using the `readMATmatrix` function of the DataFiles package. (The DataFiles package can be accessed by typing the command **import DataFiles** in the command input line of the command window. To read e.g. the ABCD matrix, the function call `DataFiles.readMATmatrix("dslin.mat","ABCD")` can be used.)

If the LinearSystem option is available in Dymola, the function `Modelica_LinearSystems.StateSpace.fromFile()` can be used to display the information in the A, B, C and D matrices, respectively, as state-space representation. This can in turn be used by other LinearSystems functions.

If the dslin.mat file should be used in Matlab, it can be loaded using the command `load('filepath\dslin.mat')` or the m-file `tloadlin()`. (Using the m-file, e.g. the ABCD matrix can be loaded using the command `[A,B,C,D]=tloadlin()` assuming the name of the file dslin.mat has not been changed.)

Creating the B, C and D matrices

The A matrix is always created. To create the other matrices, u must be declared as an input and y as an output. This might be done any of the following ways:

- To create an output, a sensor could be added and a graphical connection can be drawn from the output of that sensor. While drawing, right-click and select **Create Connector**. Inputs could be created in a similar way.
- Suitable input/output connectors from Modelica.Blocks.Interfaces can be used, SISO etc.
- Inputs and outputs can be declared as

```
input Real u;
output Real y;
```

Linearizing around a specific time point

For inputs an offset can be manually added (by sending the input and a constant to an Add block) to make it possible to linearize around input values other than 0.

Another way is to simulate to the desired time-point, then use the built-in function `importInitial` and then the built-in function `linearizeModel`. For information about `importInitial`, please see "importInitial" on page 943, for information about `linearizeModel` please see below.

Corresponding function

The **Simulation > Linearize** command corresponds to the built-in function `linearizeModel`. Please see section "linearizeModel" on page 945.

Simulation > Save in Model

Opens a menu for saving of start values in the current model or in an extended model. For more information, see section "Save start values in model" starting on page 625.

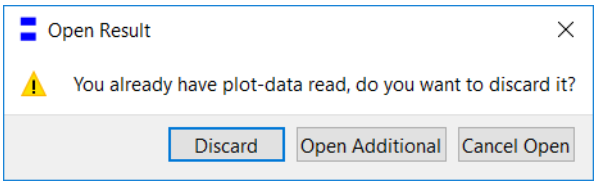
Simulation > Load Result

Reads the result file from a simulation made with Dymola to allow variables to be plotted and in some cases animated.

The result file is by default automatically opened after simulation, and if it contains animation data an animation window is also opened automatically.

You can open several result files at the same time with this command by multiselecting them using **Ctrl** or **Shift** in the resulting dialog. The result files you open this way will be displayed sorted in alphabetical order in the variable browser.

If you alredy have an open result when applying the command, you will have the following alternatives:



If you select **Open Additional**, you will have the same menu as if you did not have any open result from the start.

Note that you can also open multiple result files by dragging them into Dymola main window. To open one file, you can also double-click it. To use double-clicking, `.mat` files must be associated to Dymola.

Simulation > New Plot [Window]





Clicking directly on the **New Plot** button performs the first alternative above, that is, opens a new plot window with one diagram.

Clicking the down-arrow, you will create a new plot window with a diagram configuration as marked; in the above case, you will create a plot window with four diagrams. (These commands are also available from the **Plot: Options** tab.)

Simulation > New Table [Window]



Creates a new active table window which is initially empty. This command can be given from the **Plot: Options** tab as well.

Simulation > New Animation [Window]



Creates a new animation window. This command is also available in the **Animation** tab.

Simulation > Visualize 3D



Creates a new visualizer window for 3D models. This command is also available in the **Animation** tab.

Simulation > Simulation Analysis



Opens a menu for simulation analysis. Four tabs are available, to work with:

- Plot dependencies, see section “Plot dependencies” starting on page 633.
- Equation incidence, see section “Equation incidence graphs” starting on page 640.
- Analysing numeric integration, see section “Analyzing numeric integration” starting on page 649
- Event logging, see section “Event logging user interface” starting on page 654.

Simulation > Visualize



Visualizes initial configuration of 3D models.

Simulation > Show Log



Show Log opens the log window and shows a log for the last simulate command. More about this window can be read in the section “Log window” on page 603.

Simulation > Play



Starts an animation. The animation runs from the current frame until the last frame. If the animation is set up to run continuously (see “Animation > Animation [Setup]” starting on page 861) the animation automatically restarts with the first frame.

Simulation > Pause



Stops a running animation. This command has no effect if no animation is running.

Simulation > Rewind



Rewinds the animation to the first frame. A running animation is stopped.

Simulation > Reverse



Moves the animation backward at high speed.

Simulation > Forward



Moves the animation forward at high speed.

Simulation > Backstep



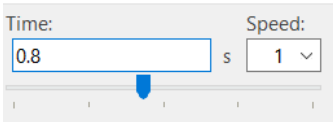
Displays the previous frame of the animation. If the animation is at the first frame, this command steps to the last frame. A running animation is stopped.

Simulation > Step Forward



Displays the next frame of the animation. If the animation is at the last frame, this command steps to the first frame. A running animation is stopped.

Simulation > [Animation] Time and Speed [Toolbar]

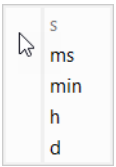


The animation time and speed toolbar consists of:

Time

The time box shows the progress of the simulation in time. You can change the value to display another time.

Simulation time unit

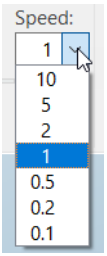


You can click the unit after the time, this is also a button to change the simulation time unit. Clicking it may give the result shown in the figure in the margin. For more information about the simulation time unit, see section "General tab" on page 800. (The simulation time unit can be changed using this tab of the simulation setup as well.)

Time slider

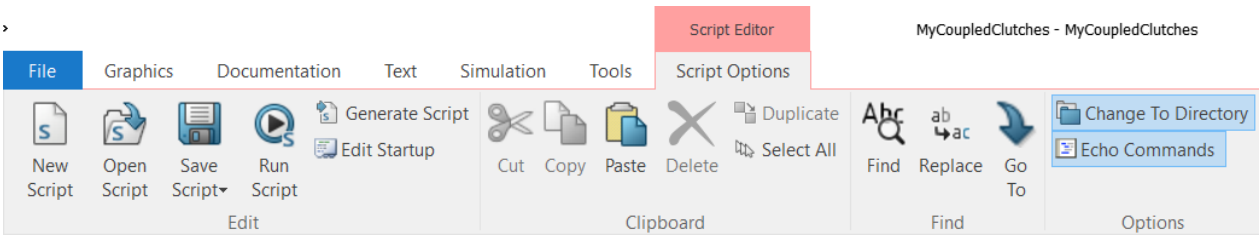
The time slider under the time shows the simulation time. You can drag it to change the time.

Speed



The speed enables setting different speed. A higher value means that the time runs faster in the animation.

5.3.3 Main window: Script Editor tab



The **Script Editor** tab is divided above into two images since it is long.

The **Script Editor** tab opens when a script is activated, by, for example, selecting any scripting command in the **Simulation** tab. See "Simulation > Run Script" on page 791.

The **Script Editor** tab contains commands to (the list corresponds to the tab groups):

- Creating, opening, saving, generating, and running scripts
- Cliboard commands (cut, copy, paste etc.)
- Find commands, including "go to"
- Various options (change directory and echoing commands)

Script Editor > New Script



Opens the script editor with a new Modelica script (.mos) file with the name `Unnamed` in Dymola script editor. If more than one unsaved new script is opened, the following script files will have the names `Unnamed 2`, `Unnamed 3` etc.

The command can be given directly from the **Simulation** tab as well, see "Simulation > Run Script" on page 791.

The scripting facilities are described more in detail in “Scripting” starting on page 904.

Script Editor > Open Script



The command opens a dialog to browse for an existing Modelica script (.mos) file. When selecting such a file and selecting **Open** in the dialog, this script file is opened in Dymola script editor.

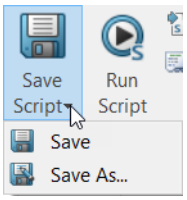
The command can be given directly from the Simulation tab as well, see “Simulation > Run Script” on page 791.

The scripting facilities are described more in detail in “Scripting” starting on page 904.

Script Editor > Save Script

Clicking directly on the Save Script button perform the first alternative below.

Clicking the arrow-down displays the alternatives:



Note: The scripting facilities are described more in detail in “Scripting” starting on page 904.

Script Editor > Save



The command opens a dialog to browse for a location to save the Modelica script (.mos) file. When the locaton is selected, clicking the button **Save** in the dialog will save the script file.

Script Editor > Save As…



Saves the active script to a new file.

Script Editor > Run Script

The command opens a dialog to browse for a Modelica script (.mos) file. When the script is selected, clicking the button **Open** in the dialog will execute the script.

The command can be given directly from the Simulation tab as well, see “Simulation > Run Script” on page 791.

The scripting facilities are described more in detail in “Scripting” starting on page 904.

Script Editor > Generate Script



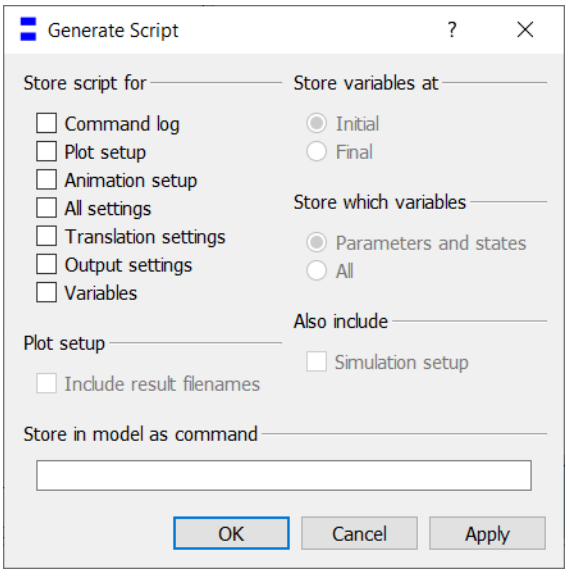
Saves various settings in a script file. The command is also available in the **Simulation** tab, as **Simulation > New Script > Generate Script…**.

When selected, the following window is shown:

For more information about creating script files using this command, please see section "Creating a Modelica script file" starting on page 918.

The command corresponds to the built-in function `saveSettings` (except storing the command log and store the script as a command in the model). Please see section "saveSettings" on page 953 for more information (and section "savelog" on page 953 for storing of the command log, and section "createPlot" starting on page 955 for an example of storing of plot setup).

Generate script file alternatives.



The Store script for group

The **Store script for** group specifies what should be stored in the script. The alternatives **Command log**, **Animation setup**, **All Settings**, and **Variables** need no further explanation.

Plot setup.

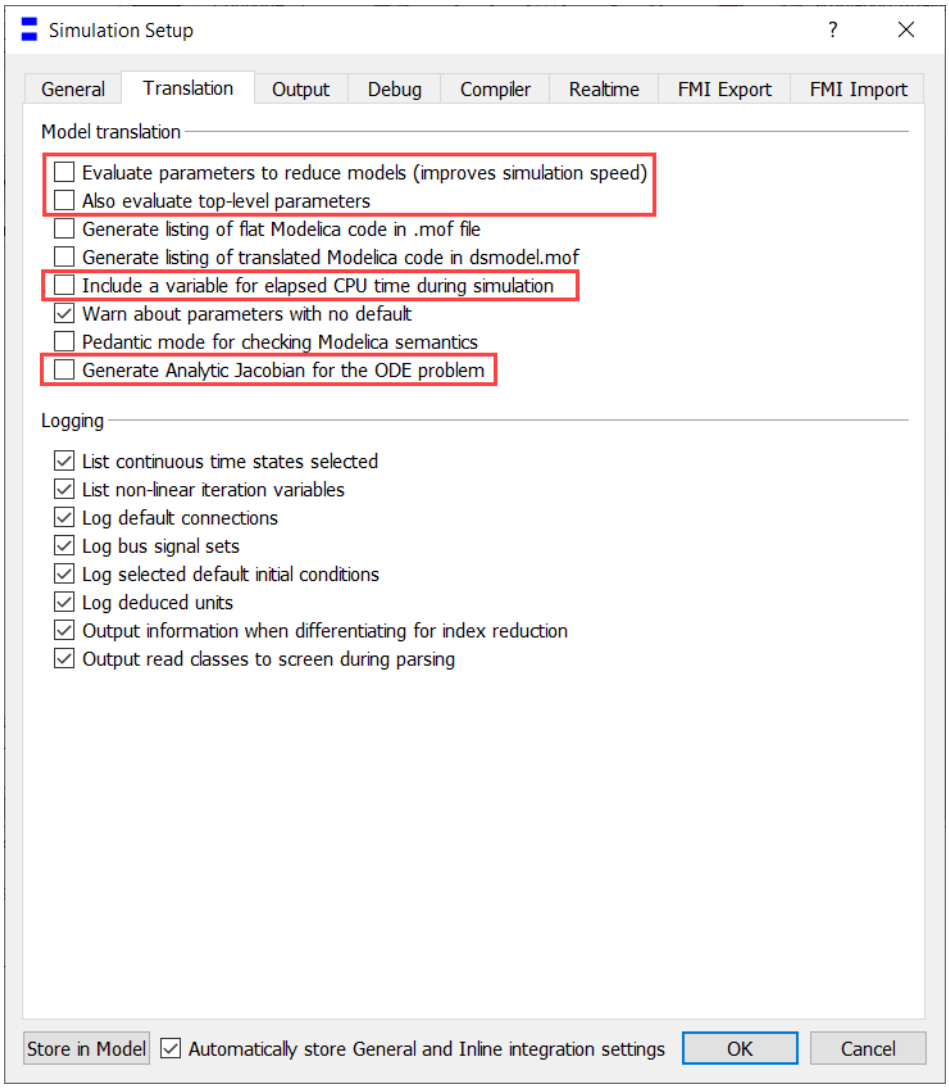
Concerning the **Plot setup** alternative, note:

- Part of the plot setup is also automatically saved (however except what variables are shown) Please see the command **Tools > Options**, the **Settings** tab, in previous chapter for more information.
- You can use the built-in function `saveSettings` as an alternative to **Script Editor > Generate Script** to create the plot setup script (see above).
- You can also use the command **Commands > Add Commands**, selecting **Plot window setup**, to create a command in the model that can be used to retrieve the saved plot windows settings. You can also select to automatically run this command after the simulation by selecting **Automatically run after simulating**. See the section "Simulation > Commands > Add Command" starting on page 793 for more information about this command. **Important!** There is no eqivivalent in this command to the setting **Include result filenames** in the "Plot setup group" below; the **Add Commands** alternative does not support reading different result files.

Translation settings.

Activating **Translation settings** will include a number of flags corresponding to settings in the **Model translation** group in the **Translation** tab of the simulation setup, reached by the command **Simulation > Setup**, the **Translation** tab.

To be specific, the flags corresponding to the framed settings below are included:



Note that there are some differences when comparing this feature with storing the translation settings in the model using the button **Store in Model** (in the lower left corner in the figure above):

- Using **Store in Model**, the settings **Generate listing of flat Modelica code in .mof file** and **Generate listing of translated Modelica code in dsmodel.mof** are also included.

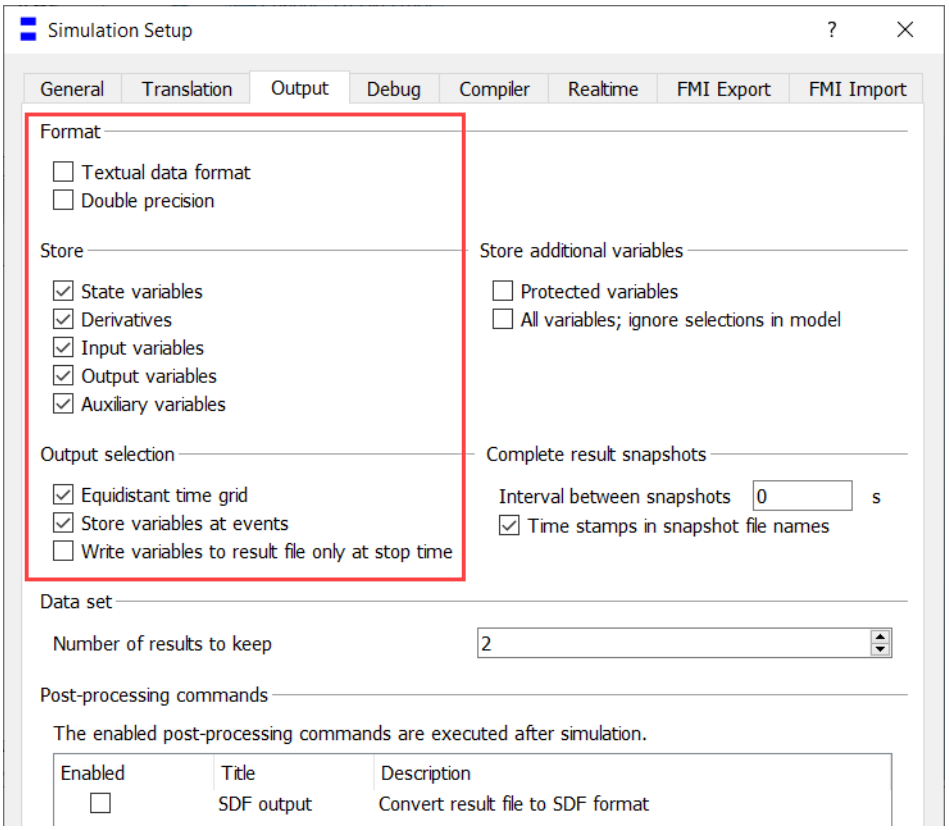
The reason they are not included in **Translation settings** is that they do not influence the generated C code.

- Using **Store in Model**, the setting **Generate Analytic Jacobian for the ODE problem** is not included. It is included in **Translation settings**.

Note also that **Translation settings** is also included in the option **All Settings**; if you activate **All Settings**, the **Translation settings** is grayed out.

Output settings.

Activating **Output settings** will include the built-in function `experimentSetupOutput` in the script; this built-in function includes the settings in the groups **Format**, **Store**, and **Output selection** in the **Translation** tab of the simulation setup, reached by the command **Simulation > Setup**, the **Translation** tab – the settings framed:



Note that this setting is not included in the setting **All settings**.

The Plot setup group

The **Plot setup** group setting **Include result filenames** specifies if result filenames should be included when the plot setup is stored with the corresponding command in the previous group. This is needed if you want to generate a script from a plot that contains curves from different result files.

The Store variables at group

The **Store variables at** group decide whether specified initial values or final values after simulation should be stored.

The Store which variables group

The **Store which variables** group decides whether all variables or only parameters and initial values of states should be saved.

The Also include group

The **Also include** group gives possibility to also include simulation setup, that is, start time, stop time and integrator. Note that there are, for advanced users, some additional flags that are related to the simulation setup, see section "Advanced setup for simulations by flags" on page 782.

The Store in model as command group

It is possible to specify that the generated script shall be callable from the Commands menu associated with the current model using **Store in model as command**. The name that should be used as command should be specified.

When the command is executed, the script is opened in the **Script Editor**, to make it easy to add a description, and also to acknowledge that the command did work. If you don´t want to add anything, you can just close the **Script Editor**.

Script Editor > Edit startup



The command is used to create and edit a customized setup file. For the handling of this file, see index entry "*startup file: customized startup file*" in this manual, for the scripting facilities details, see "Scripting" starting on page 904.

Script Editor > Cut



Cuts the selected text and move it to clipboard.

Script Editor > Copy



Copies the selected text to clipboard.

Script Editor > Paste



Pastes the copied text from clipboard.

Script Editor > Delete



Deletes the selected text.

Script Editor > Duplicate



Duplicates selected items.

Script Editor > Select All



Selects all text in the script editor.

Script Editor > Find



Finds text in the editor. For more information, see the command Find and Replace in the section “Context menu: Script editor” starting on page 885. *Please note* however that there is a specific setting for the command window if all found occurrences of whole words should be highlighted. It has the same name as the general setting, **Highlight all matches**, but is located in the **Command Window** group of the **Text Editor** tab of the command **Tools > Options**. (The general setting, which is not applicable for the command window, is located in the **Modelica text editor** group.) The setting is by default not activated, for performance reasons – the command log might be quite big. For more information, see index entry “*tools : options*” in the index in this manual.

Script Editor > Replace

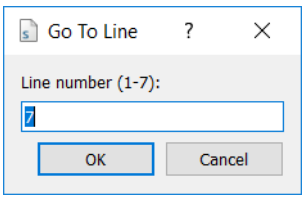


Finds and replaces text in the editor. See previous command.

Script Editor > Go To



Sets the cursor at specified line and scrolls window if necessary. The number of the current line is shown as default.



Script Editor > Change to Directory

If this command is active, the working directory is changed to the directory where a script is located, before it is executed. The button works in toggle mode, and is by default activated (this is indicated by the button being displayed as depressed).

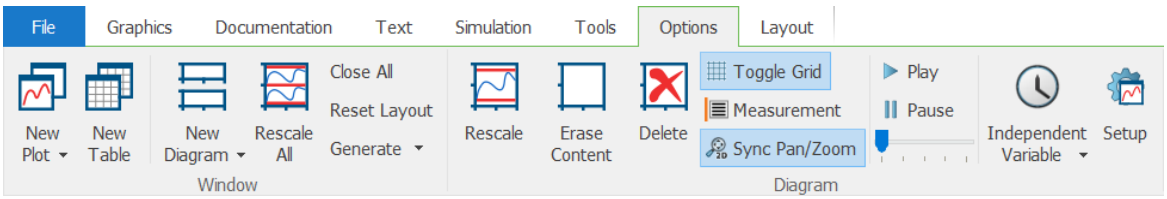
The scripting facilities are described more in detail in “Scripting” starting on page 904.

Script Editor > Echo Commands

If this command is active, the scripting commands when running a script are echoed. The button works in toggle mode and is by default activated (this is indicated by the button being displayed as depressed).

The scripting facilities are described more in detail in “Scripting” starting on page 904.

5.3.4 Main window: Plot tab: Options subtab



The **Plot: Options** subtab is divided above into two images since it is long.

The **Plot: Options** subtab contains commands to (the list corresponds to the tab groups):

- Handling plot windows
- Handling plot diagrams
- Working with signal operators, plot expressions and text
- Changing plot appearance
- Changing display unit, show component or negate curve

When plotting, there must be an active plot window. If the active plot window has multiple diagrams, one of the diagrams is active and the plot commands are directed to this diagram.

Plot: Options > New Plot [Window]





Clicking directly on the **New Plot** button performs the first alternative above, that is, opens a new plot window with one diagram.

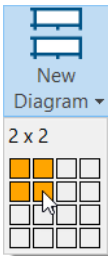
Clicking the down-arrow, you will create a new plot window with a diagram configuration as marked; in the above case, you will create a plot window with four diagrams. (These commands are also available from the **Simulation** tab.)

Plot: Options > New Table [Window]



Creates a new active table window which is initially empty. This command can be given from the **Simulation** tab as well.

Plot: Options > New Diagram





Clicking directly on the **New Diagram** creates a new active diagram in the active plot window. By default, the horizontal range of the new diagram is the *present* horizontal range of the previous active diagram (zooming included). This behavior is controlled by the option **Copy horizontal range for new sublplots** in the plot setup. See “Options tab” on page 854 for details.

Clicking the down-arrow, you will add diagrams in the present plot window to form a diagram configuration as marked; in the above case, you will have four diagrams. Given you start from only one diagram, that one will be the the top left one.

Plot: Options > Rescale All



Rescales all diagrams (reset them to initial zooming) in the active plot window

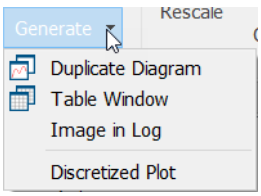
Plot: Options > Close All

Closes all plot windows. This command corresponds to the built-in function `removePlots(false)`.

Plot: Options > Reset [Window] Layout

Rescales all diagrams in the active plot window and distribute them evenly in the window.

Plot: Options > Generate



The alternatives in this menu are:

Plot: Options > Generate > Duplicate Diagram

Copies the selected diagram, with content, to a new plot window.

Plot: Options > Generate > Table Window

Copies the curves in the selected diagram to a new table window.

Plot: Options > Generate > Image in Log

Inserts the plot and corresponding command in the command window. Please see section “Insert plot and its corresponding command in command window” on page 753 for more information.

Plot: Options > Generate > Discretized Plot

You can generate a discretized plot from variables in a plot window. For more information, see section “Plotting a 1D discretization” starting on page 631.

Plot: Options > Rescale [Diagram]



Rescales the active diagram so that the full range of all plotted variables is shown. If the diagram has not been zoomed in, rescaling is performed automatically when new variables are selected for plotting.

Plot: Options > Erase Content



Erases all curves (and text objects, if any) in the active diagram. If automatic rescaling is enabled, the diagram is rescaled to the default scale.

Plot: Options > Delete [Diagram]



Deletes the currently active diagram.

Plot: Options > Toggle Grid



Enables grid lines in the diagram if they are not visible, otherwise disables grid lines. By default, grid lines are shown.

Plot: Options > Measurment Cursor



Displays the measurement cursor with the corresponding data box. The button works in toggle mode. See section “Using measurement cursor for plots” on page 715 for more information.

Plot: Options > Sync Pan/Zoom



Controls if horizontal panning/scrolling and zooming should be synchronized for all diagrams in the active plot window or not. By default, the horizontal panning/scrolling and zooming is synchronized. The command button controls the following commands:

- Horizontal zoom: **Shift+Drag** (with mouse), see “Horizontal (time axis) zooming” on page 707.

- Zoom to curve: **Alt+Drag** (with mouse), see “Zooming with maximum scaling in y direction” on page 705.
- Horizontal panning/scrolling: **Shift+Mouse wheel**, see “Moving in the plot window” on page 703.

The command is individual for each plot window. The command works in toggle mode.

Plot: Options > Play



Starts an animation. The animation runs from the current frame until the last frame. If the animation is set up to run continuously (see “Animation > Animation [Setup]” starting on page 861) the animation automatically restarts with the first frame. The idea is that you should be able to handle an animation without leaving the plot.

Plot: Options > Pause



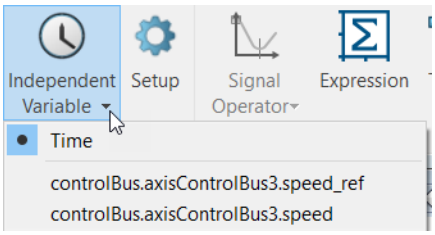
Stops a running animation. This command has no effect if no animation is running.

Plot: Options > Time slider

The time slider shows the animation time for the animation and the measurement cursor. You can drag it to change the time. Note that the time input field and the speed setup is only available in the **Simulation** tab.

Plot: Options > Independent Variable

Clicking on **Independent Variable** enables selection of what should be the value of the horizontal axis in the plot window. Default is usually Time. An example:



For more information, see section “Selecting independent variable” on page 629.

Plot: Options > Setup



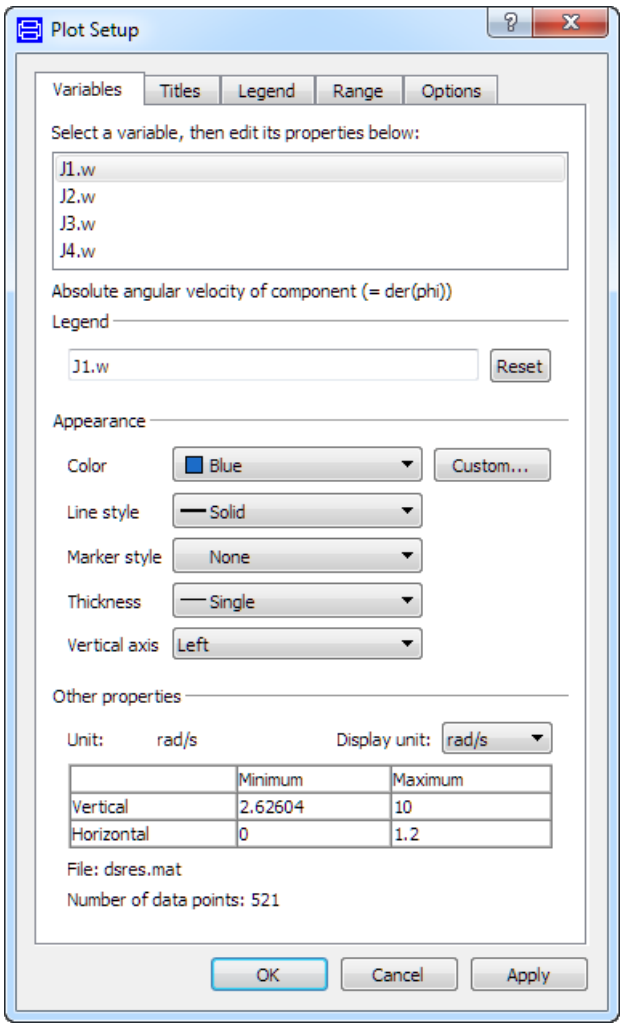
The settings of a plot window, including displayed variables, can be saved by two commands:

- The command **Simulation > Commands > Add Commands**, selecting **Plot window setup**. You can have this command to be executed automatically after simulation by selecting **Automatically run after simulation**. For information about this command, see section “Simulation > Commands > Add Command” starting on page 793.
- The command **Script Editor > Generate Script**, ticking **Plot setup**. See section “Script Editor > Generate Script” on page 833.

Note that there are some differences between these commands, if you have specifc requirements, please compare them in detail.

Variables tab

Variables tab of the plot window setup.



Selecting a variable in the top pane displays the corresponding information in the rest of the window. Once a variable is selected, it is also possible to navigate by pressing the Up and Down arrow keys to select another variable. If a comment for the variable exists, it is shown below the list box.

Note the possiblility of selecting several variables by keeping **Ctrl** pressed while selecting. This can be used to for example associate a number of curves with the secondary vertical axis.

The **Legend** group contains the current description of the selected variable. The legend may be written using extended character set (UNICODE). Edit the text and press `Apply` to change

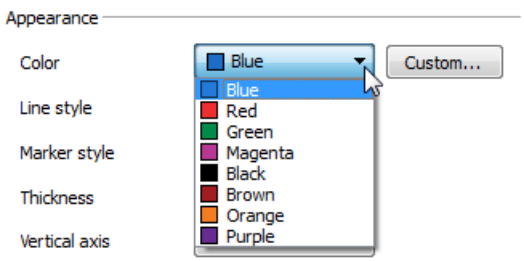
the legend. If the text is empty, no legend is shown for this variable. Pressing the **Reset** button fills the edit field with the default legend. Note that you also can double-click the legend in the plot window to edit it.

The **Appearance** group enables changing the color, line/marker style, thickness, and axis of the selected variable. Note that the items in this group are also available from the context menu for each curve/legend in the plot window.

Color

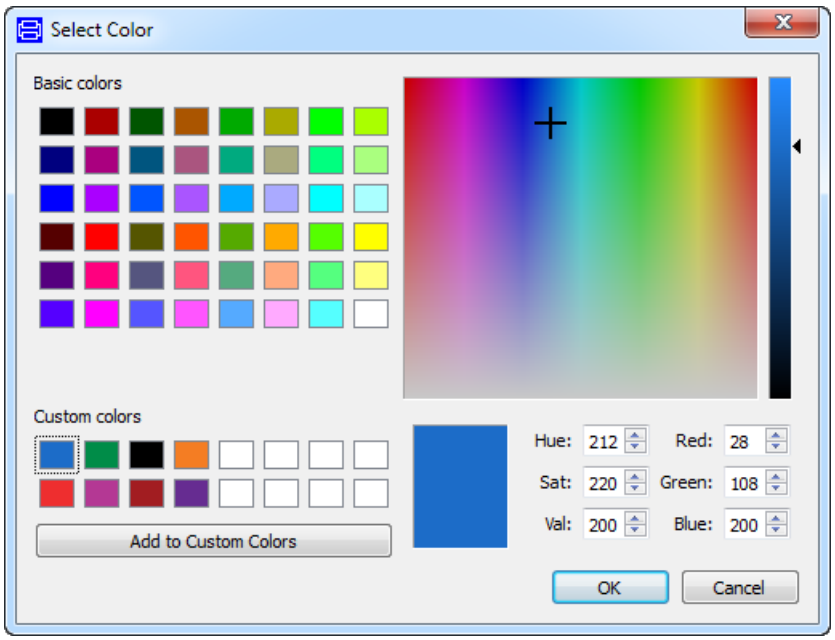
The default drop-down alternatives are:

Color of a variable.



Custom colors can be set by clicking the **Custom...** button. A color palette is displayed:

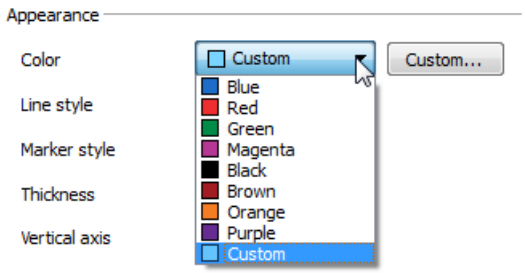
The color palette.



The handling of the palette is described in the chapter “Developing a model”, section “Graphical object menus: Line and fill style”.

Selecting a white square in the **Custom colors** area, and then selecting a color in the right part of this palette and then clicking **OK** will display this selection as a new custom color last in the drop-down list, e.g.:

Custom color of a variable.

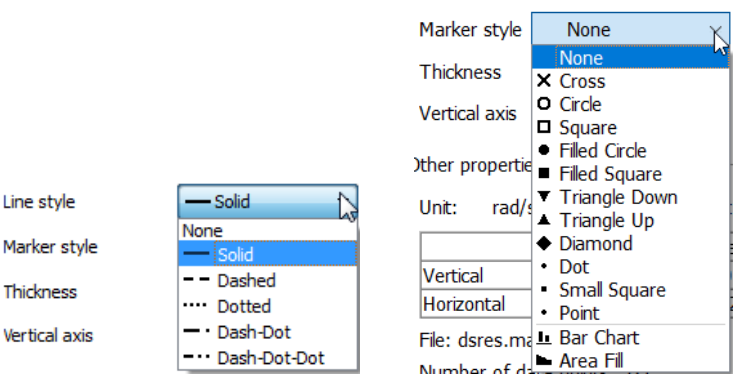


Only one custom color per variable is displayed. Note that any color selection in the palette will define this color as custom color for the variable; it does not have to correspond to a custom color in the palette.

Line style and marker style

The drop-down alternatives for line style and marker style are:

Line style and marker style of a variable.

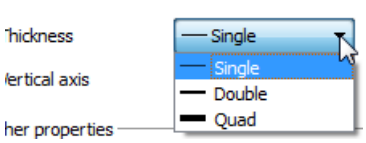


The line style alternatives are the same as for lines when drawing primitives.

Thickness

The drop-down alternatives for thickness are:

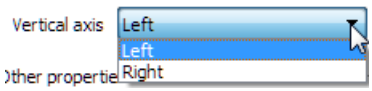
Thickness of a variable.



The thickness alternatives are the same as for lines when drawing primitives.

Vertical axis selection

The drop-down alternatives for vertical axis (y-axis) are:

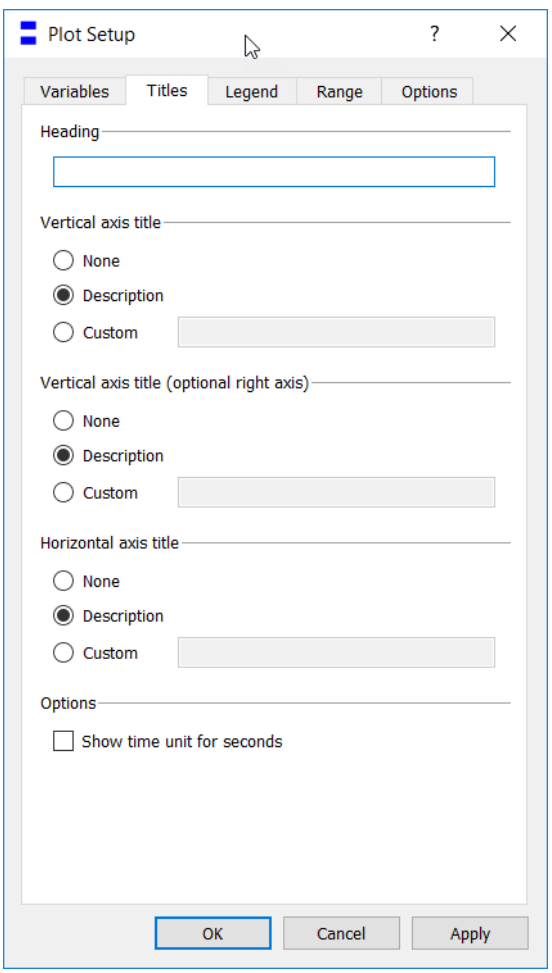


Handling of multiple vertical axes (y-axes) is described in the section “Muliple y-axis in plot window” on page 720.

The **Other properties** group shows the name of the result file, the number of data points, as well as minimum and maximum values of the variable. Plotted values can be interrogated in the diagram, see “Dynamic tooltips for signals/curves” on page 700.

Titles tab

Titles tab of the plot window setup.



All text in this tab may be written using extended character set (UNICODE).

The user may enter a heading to be displayed above the active diagram. An empty heading is not shown; the space is used for the diagram instead.

Vertical axis title specifies the title for the vertical axis of the diagram.

- **None** No title is shown by the vertical axis.
- **Description** The title is extracted from the descriptions and units of plotted variables.
- **Custom** The title is specified by the user in the input field.

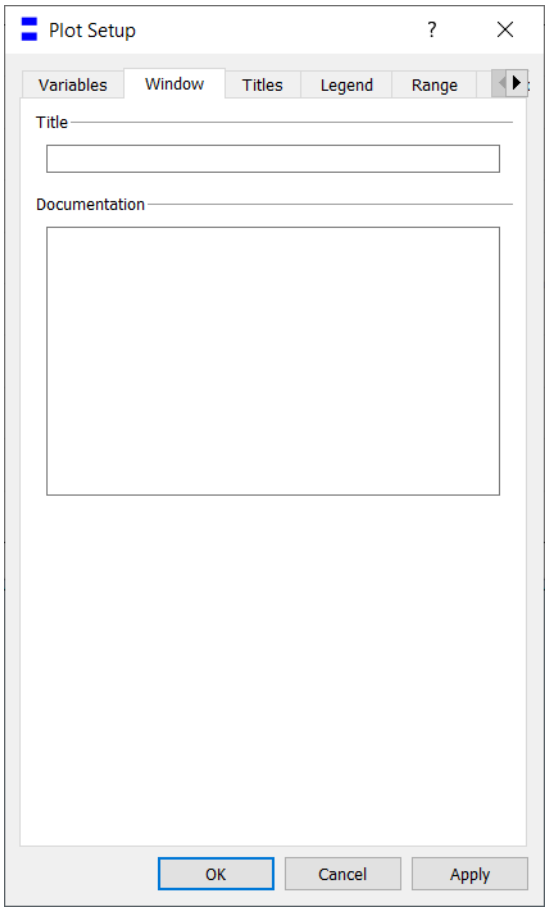
Vertical axis title (optional right axis) specifies the title for the secondary vertical axis to the right, if such axis exists. The alternatives are similar to the ones above.

Horizontal axis title specifies the title for the horizontal axis of the diagram. The alternatives are similar to the ones above.

Options currently only contains one option **Show time unit for seconds**. This option is by default not activated, but if activated, and the horizontal axis title is specified as **Description**, the horizontal axis title will be `Time [s]` if the time unit of that axis is seconds. If the option is not activated, this axis title will not be shown in such case.

Window tab

Winod tab of the plot window setup.



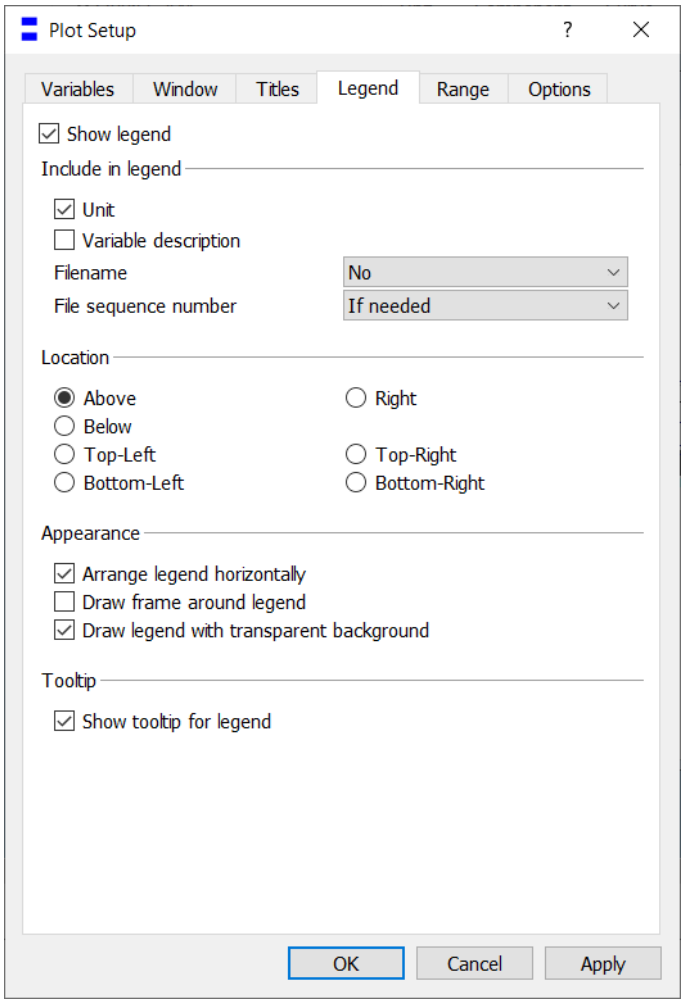
- The **Window** tab can be used to add title and documentation to the plot window. For more information, and an example, please see section “Printing the result.

Note that the argument `fileName` in the built-in function also supports URL´s, in addition to file paths.

Adding titles and documentation to plot windows” on page 748.

Legend tab

Legend tab of the plot window setup.

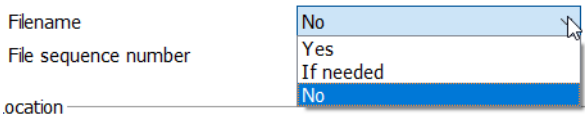


The legend tab displays names and other properties about plotted variables. The first setting **Show legend** is the default. If unchecked, no legend is displayed.

The **Include in legend** group specifies the content in the legend:

- **Units** of variables in the vertical axis title or in the legend. The unit is shown in the axis title (if all variables have the same unit) or in the legend (if there are multiple units on one axis).
- **Variable description** adds the description of the variable, as specified in the model, to the vertical axis title or the legend. The variable description is shown along the axis, if it is the same for all variables.

- **Filename** specifies if the file name (without extension .mat) should be added in the legend, as `// file name`. The following alternatives are possible:



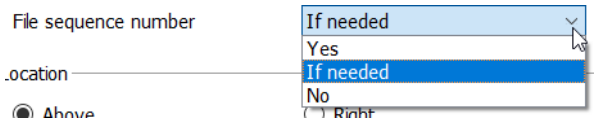
  Yes means always display file name in legend. This selection corresponds to the flag `Advanced.Plot.Legend.AlwaysFilename = true`.

  If needed means to add the file name if the plot contains variables from more than one result file. This selection corresponds to the flag `Advanced.Plot.Legend.Filename = true`.

  No means never display file name in the legend. This is the default and corresponds to the flag `Advanced.Plot.Legend.Filename = false`.

  (If the flag `Advanced.Plot.Legend.AlwaysFilename` is `true`, it overruns the flag `Advanced.Plot.Legend.Filename`, independent of the value of that flag. If however the flag `Advanced.Plot.Legend.AlwaysFilename` is `false`, the value of the flag `Advanced.Plot.Legend.Filename` specifies how the file name is treated in the legend.)

- **File sequence number** specifies if the file sequence number shoud be added in the legend of the variable, as `// sequence number`. The following alternatives are possible:



  Yes means always display file name in legend. This selection corresponds to the flag `Advanced.Plot.Legend.AlwaysSequenceNumber = true`.

  If needed means to add the file name if the plot contains variables from more than one simulation. This selection if the default and corresponds to the flag `Advanced.Plot.Legend.SequenceNumber = true`.

  No means never display file sequence number in the legend. This corresponds to the flag `Advanced.Plot.Legend.SequenceNumber = false`.

  (If the flag `Advanced.Plot.Legend.AlwaysSequenceNumber` is `true`, it overruns the flag `Advanced.Plot.Legend.SequenceNumber`, independent of the value of that flag. If however the flag `Advanced.Plot.Legend.AlwaysSequenceNumber` is `false`, the value of the flag `Advanced.Plot.Legend.SequenceNumber` specifies how the file name is treated in the legend.)

The **Location** group settings specifies the location of the legend; the location is either outside of the diagram (above, below or to the right of the diagram), or close to one of the corners inside the diagram. See section “Plot window” on page 591 for examples of diagrams with different locations of the legend.

The **Appearance** group specifies the following:

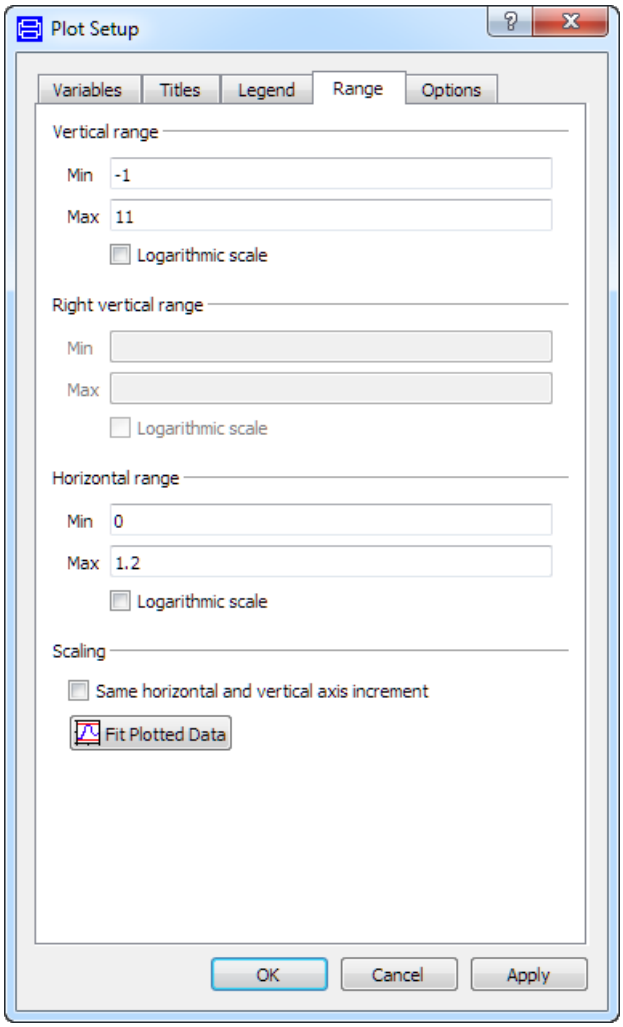
- **Arrange legend horizontally** lists the variables from left to right in the legend (the default). If unchecked, the variables are listed vertically in the legend.
- **Draw frame around legend** when outside of diagram. Inside the diagram the frame is always drawn.
- **Draw legend with transparent background** (when inside of diagram) is default, in order to not hide any curve. The default opacity is 0.7; it can be changed to e.g. 0.6 by setting `Advanced.Legend.Opacity=0.6`. Opacity 0 means no opacity, opacity 1.0 means that the legend background will be fully opaque.

A tooltip is available for the legend.

- **Show tooltip for legend** is default; if unchecked no tooltip is shown; however, the highlighting of the corresponding legend is still present.

Range tab

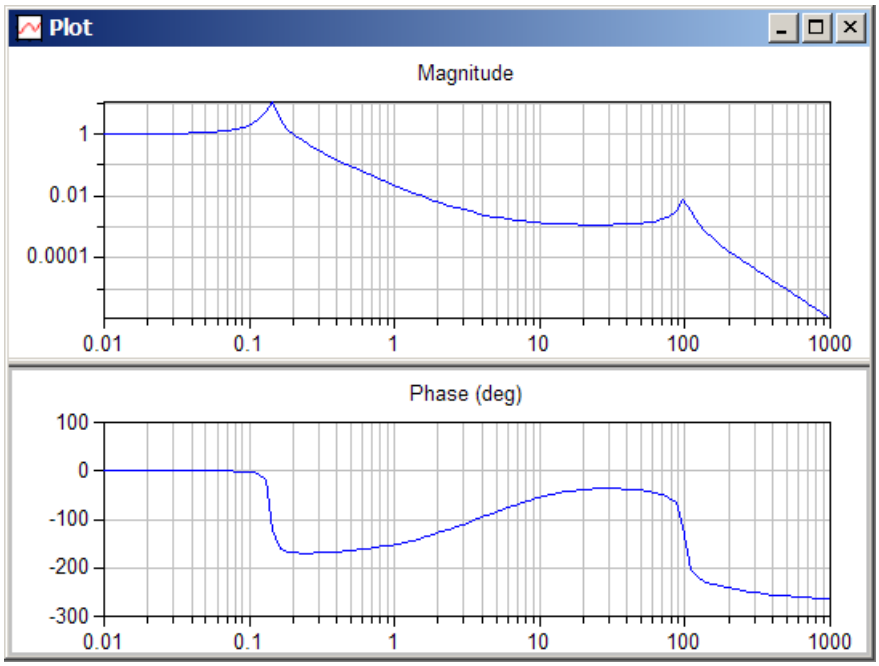
Range tab of the plot window setup.



The Range tab allows the user to define the minimum and maximum ranges of the axes and to select logarithmic scale instead of linear scale. Other settings are:

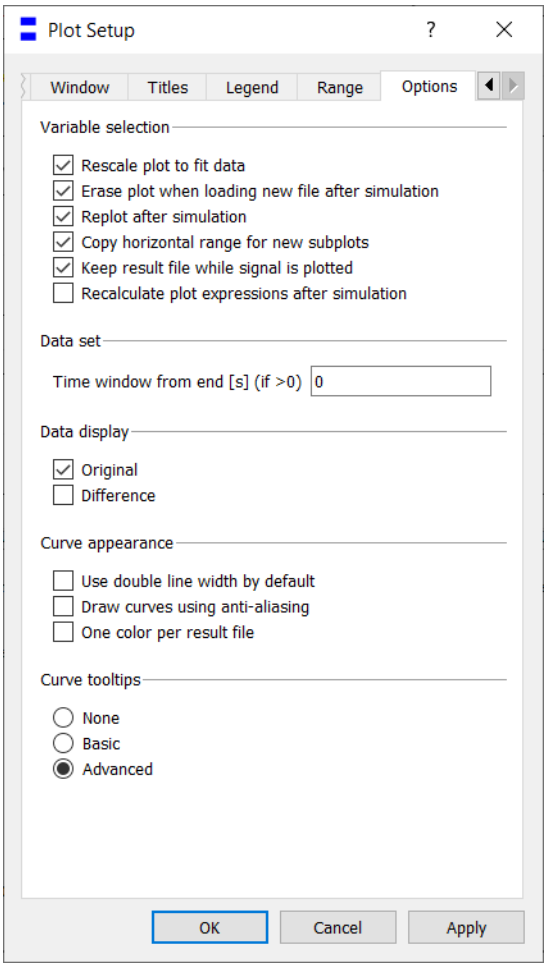
- **Same horizontal and vertical axis increment** may be used when the relative scale of vertical and horizontal axes is important.
- **Fit Plotted Data** rescales the diagram to fit plotted variables. This is same as pressing the Rescale button and then reads the default range.

Logarithmic scale used in Bode plot.



Options tab

Options tab of the plot window setup.

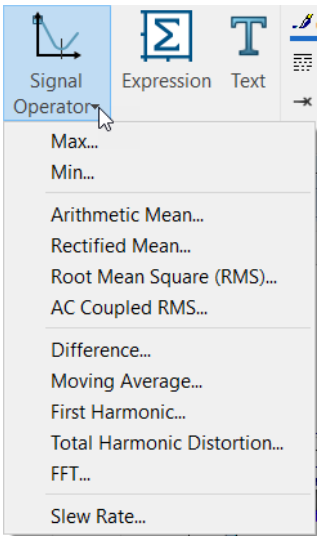


The Options tab sets up options controlling plotted variables and how variables are plotted.

- **Rescale plot to fit data** when zoomed out. When diagram is zoomed in, automatically rescaling is disabled.
- **Erase plot when loading new file after simulation** to have automatic erase of the diagram when new results are loaded (and thus when a simulation is run). Note that this setting is only valid for the current window, there is a flag `Advanced.DefaultAutoErase` that controls this behavior globally for the session. The flag is by default `true`. (However, plot windows created by the built-in function `createPlot` are not influenced by this flag.)
- **Automatically replot after simulation**; if not selected variables must be manually plotted after a simulation.

- **Copy horizontal range for new subplots**. By default, the horizontal range of a new subplot is copied from the *present* horizontal range of the previously active diagram (zooming included). Unchecking this option means that the horizontal range of a new subplot will be the *default* horizontal range of the previously active diagram, that is, any zooming of the range will not be copied. The default behavior corresponds to the flag `Advanced.AutoScaleSubPlot=true`.
- **Keep result file when signal is plotted** – By default a simulation result file that contains a plotted variable is kept from being closed when a new simulation result file is created. However, this might not be wanted when, for example, you work with manually sweeping parameters. For more information about keeping files this way, see "Keeping result files for which variables have been selected (automatic selection)" on page 675. The default status corresponds to the flag `Advanced.Plot.AutoPinResults=true`. The setting is saved between sessions.
- **Recalculate plot expressions after simulation** – Activating this setting, plot expression curves are automatically recalculated after simulation. By default, this setting is not activated. This corresponds to the flag `Advanced.Plot.AutoUpdatePlotExpressions = false`. The setting is saved between sessions.
- **Data set: Time window from end [s] (if > 0)** defines the size of the time window during online plotting. For more information, see "Applying time sliding window" on page 711.
- Data display; two options are available:
    - **Original** – show the original curves
    - **Difference** – show the difference between the curves
- Curve appearance; three options are available:
    - **Use double line width by default**
    - **Draw curves using anit-aliasing**
    - **One color per result file** – curves from different result files gets different colors. For more information, see "Assign different colors to curves from different simulation result files" on page 721.
- **Curve tooltips**: one of the following can be selected:
    - **None** Tooltips are not displayed. This corresponds to the flag `Advanced.ShowPlotTooltip=false`.
    - **Basic** Tooltips contain the variable name, value and time.
    - **Advanced** [default] Toolips contain the variable name, value, time local min, local max and slope.

Plot: Options > Signal Operator

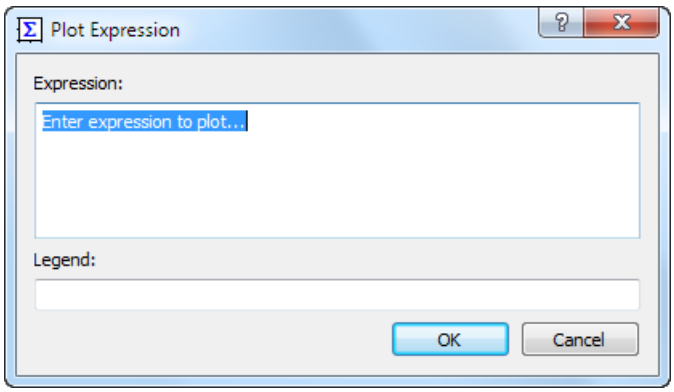


This command makes it possible to add signal operators to the curve. Please see section "Displaying signal operators" starting on page 722 for information about this.

Plot: Options > Expression



Displays a dialog for creating an expression that will be plotted if the expresson is valid.



Please see section "Plotting general expressions" starting on page 736 for more information.

Plot: Options > Text



Inserts a text object in the active diagram. The text object is treated like text objects in the diagram layer. Please see section "Insert and edit text objects in the plot" on page 751 for more information.

Plot: Options > [Line] Color



See the corresponding command for the plot setup, section “Color” on page 844.

Plot: Options > Line Style



See the corresponding command for the plot setup, section “Line style and marker style” on page 845.

Plot: Options > Marker Style



See the corresponding command for the plot setup, section “Line style and marker style” on page 845.

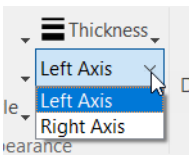
Plot: Options > Thickness



See the corresponding command for the plot setup, section “Thickness” on page 845.

Plot: Options > Left/Right [Vertical Axis Selection]

The drop-down alternatives for selecting of vertical axis (y-axis) for the selected curve are:

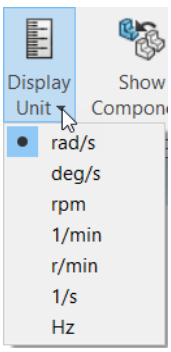


The command can be used to associate the selected curve to a secondary vertical axis (y-axis) to the right, by selecting the subentry **Right Axis**. By default, the **Left Axis** is selected:

If **Right** is selected, and such an axis is not present, it will be created. Like previous features, this feature is also available in the plot setup menu. See section “Variables tab” on page 843 (useful when several curves should be associated with a vertical axis to the right).

Handling of multiple vertical axes (y-axes) is described in the section “Muliple y-axis in plot window” on page 720.

Plot: Options > Display Unit



This command makes it possible to change the display unit of signals. See section “Changing the displayed unit of signals” on page 712 for more information.

Plot: Options > Show Component



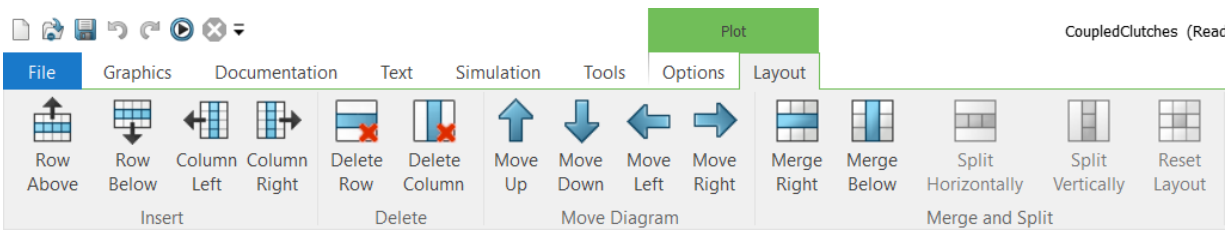
Highlights the component that contains the variable corresponding to the selected curve in the Model View window.

Plot: Options > Negated Curve



Displays the selected curve negated. A minus sign will appear in the legend of the curve. The command is also valid for plot tables. For more information, see “Display curves negated” on page 700.

5.3.5 Main window: Plot tab: Layout subtab



The **Plot: Layout** subtab contains commands for editing the layout of a Plot window, working with rows and columns of a 2D plot layout.

Plot: Layout > Row Above

Inserts a new row of diagrams above the active diagram.

Plot: Layout > Row Below

Inserts a new row of diagams below the active diagram.

Plot: Layout > Column Left

Inserts a new column of diagrams to the left of the active plot.

Plot: Layout > Column Right

Inserts a new column of diagrams to the right of the active diagram.

Plot: Layout > Delete Row

Deletes the diagram row of the active diagram, if more than one row is present.

Plot: Layout > Delete Column

Deletes the diagram column of the active diagram, if more than one column is present.

Plot: Layout > Move Up

Moves the active diagram one step up in the diagram order in the active diagram column. You can use **Ctrl+Up** as well.

Plot: Layout > Move Down

Moves the active diagram one step down in the diagram order in the active diagram column. You can use **Ctrl+Down** as well.

Plot: Layout > Move Left

Moves the active diagram one step to the left in the diagram row. You can use **Ctrl+Left** as well.

Plot: Layout > Move Right

Moves the active diagram one step to the right in the diagram row. You can use **Ctrl+Right** as well.

Plot: Layout > Merge Right

Merges the active diagram with the diagram to the right. Note that only the diagram cells are merged, not the content. The active diagram cell takes over the space and deletes the content in the overtaken diagram cell.

Plot: Layout > Merge Below

Merges the active diagram with the diagram below. Note that only the diagram cells are merged, not the content. The active diagram cell takes over the space and deletes the content in the overtaken diagram cell.

Plot: Layout > Split Horizontally

Splits a horizontally merged diagram.

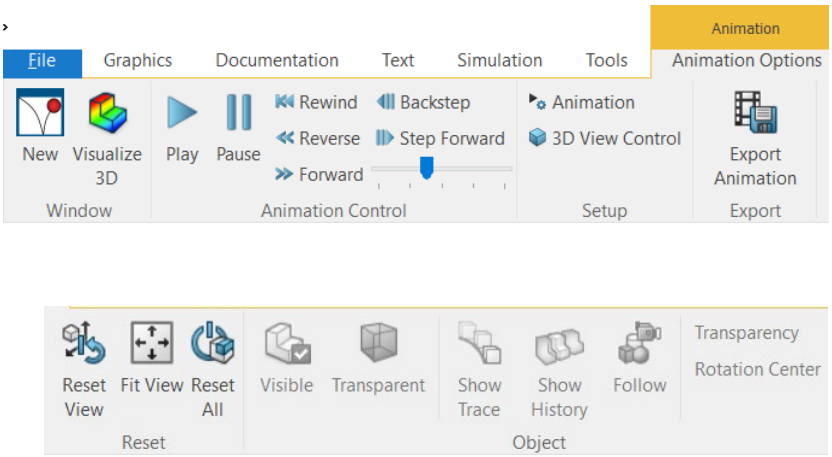
Plot: Layout > Split Vertically

Splits a vertically merged diagram.

Plot: Layout > Reset Layout

Resets the merging of a merged diagram.

5.3.6 Main window: Animation tab



The **Animation** tab is divided above into two images since it is long.

The **Animation** tab contains commands for animation of models, that is, to (the list corresponds to the tab groups):

- Opening new animation window and new vizualiser window
- Setup of animation and vizualisation
- Animation control
- Exporting the animation
- Reset commands
- Object commands, for example, to show trace, history, and follow an object

Animation > New [Animation Window]



Creates a new animation window. This command is also available in the **Simulation** tab.

Animation > Visualize 3D [Window]



Creates a new visualizer window for 3D models. This command is also available in the **Simulation** tab.

Animation > Play



Starts an animation. The animation runs from the current frame until the last frame. If the animation is set up to run continuously (see “Animation > Animation [Setup” below) the animation automatically restarts with the first frame.

Animation > Pause



Stops a running animation. This command has no effect if no animation is running.

Animation > Rewind



Rewinds the animation to the first frame. A running animation is stopped.

Animation > Reverse



Moves the animation backward at high speed.

Animation > Forward



Moves the animation forward at high speed.

Animation > Backstep



Displays the previous frame of the animation. If the animation is at the first frame, this command steps to the last frame. A running animation is stopped.

Animation > Step Forward



Displays the next frame of the animation. If the animation is at the last frame, this command steps to the first frame. A running animation is stopped.

Animation > Time slider

The time slider shows the animation time for the animation and the measurement cursor. You can drag it to change the time. Note that the time input field and the speed setup is only available in the **Simulation** tab.

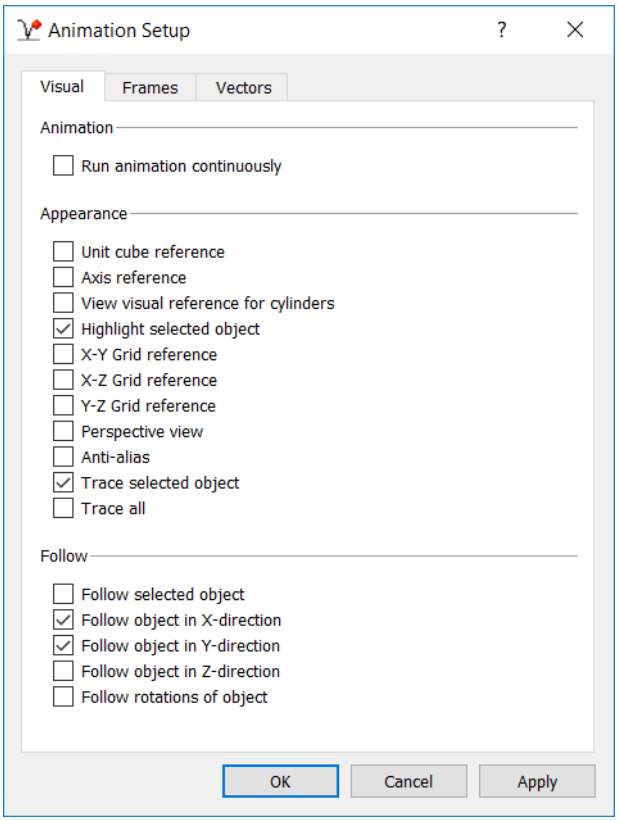
Animation > Animation [Setup]



Changes several options that control animation and viewing of animated objects.

Visual Tab

The animation setup dialog.



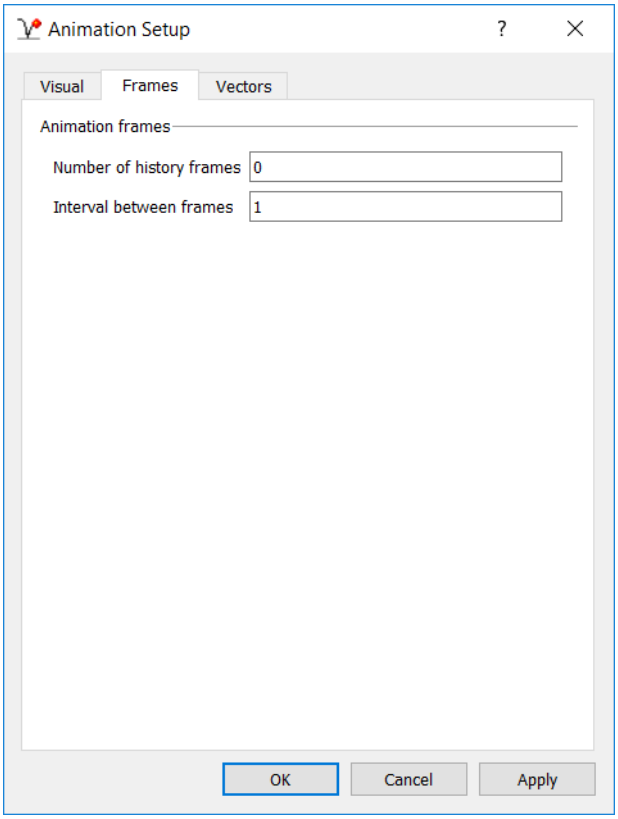
The Visual tab contains:

- **Run animation continuously**, which toggles whether the animation is run continuously or not. If the animation is set to be continuously running, the animation automatically restarts at the beginning instead of stopping at the end. The default mode is off.
- **Unit cube reference** to display a unit cube with corners at {0,0,0} and {1,1,1} for reference.
- **Axis reference** to display axes, where the x-axis is red, the y-axis is green, and the z-axis is blue. Remember xyz = rgb.
- **View visual reference for cylinders** to make cylinders to be drawn with a black reference line on the side in order to make it easier to see if the cylinders are rotating.
- **Highlight selected** object colors the selected object red.
- **Grid reference** to display reference grids in X-Y, X-Z or Y-Z plane.
- **Perspective view** to toggle between perspective view and orthographic projection. The default is orthographic projection.

- **Anti-alias** to obtain smoother edges. However, it may make the animation much slower.
- **Trace selected object** to trace the selected object, i.e. to show the path of its origin.
- **Trace all** to trace all objects, i.e. to show the path of their origin.
- **Follow selected object** enables the view to dynamically change to follow the animated object. This can be done conditionally in different directions by enabling/disabling
- **Follow object in X, Y,** or **Z-direction** or **Follow rotations of object**. The Follow feature is useful for animating mechanisms that move in the global coordinate system, such as, vehicles.

Frames tab

Animation frames tab.

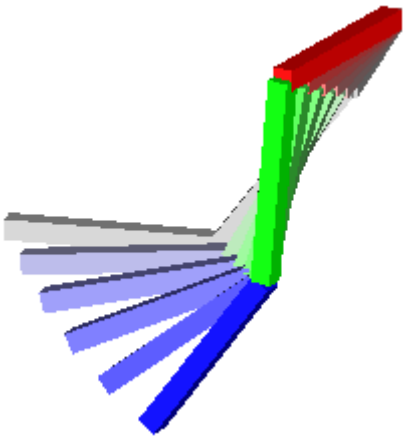


The Frames tab sets a number of attributes related to history frames. History frames gradually fade to the grey color of the background.

- **Number of history** frames introduces motion blur showing several frames. Zero will turn off all history frames. A large number of history frames may slow down the animation considerably. If the number of history frames times the interval between history frames is greater than the total numbers of frames, all frames are displayed and there is no performance penalty.

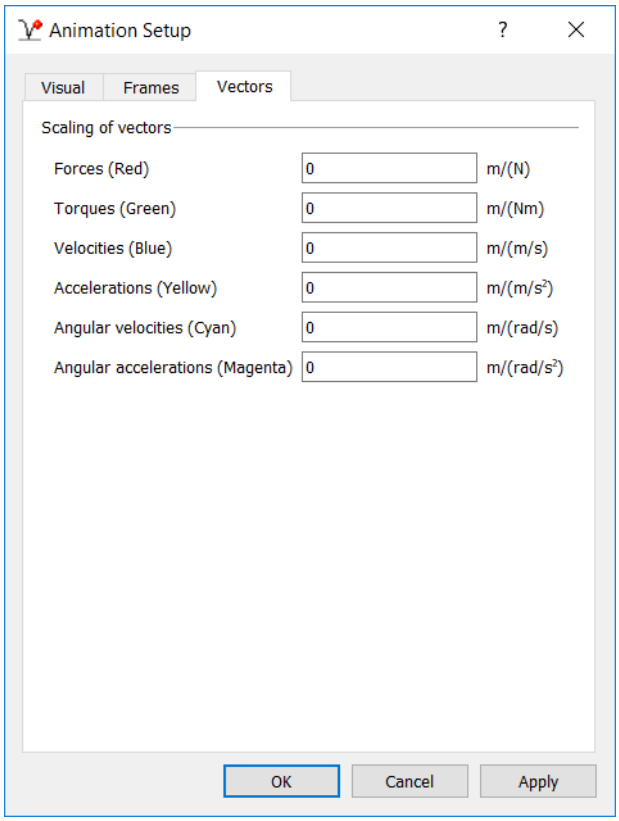
- **Interval between history frames**. With the default frame interval (1), frames *1, 2, ..., n* older than the leading edge frame are shown. With a frame interval of *m,* frames *m, 2m, ..., 2mn* are shown. It is sometimes necessary to increase the frame interval to get a suitable visual separation between the history frames.

History frames for the Furuta pendulum.



Vector tab

Vector scaling setup.



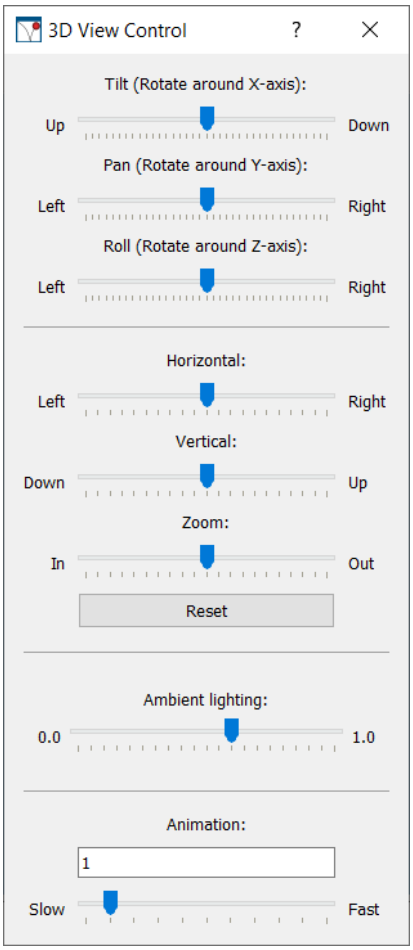
Vector tab is for controlling the scaling of vector visualizers (found in Program Files\Dymola 2023\Modelica\VisualVector.mo). The units indicate that if you for example set the scale factor for forces to 1e-3 a force of 100N will be represented by an arrow 0.1m (=1e-3 m/N*100N) long in scale of the diagram.

Animation > 3D View Control



3D view control.

This command creates a window with controls for 3D viewing.



The setting **Ambient lighting** is available if you use extended lighting, or if the model has been created by the built-in function `animationLightning`. Extended lighting is available from Dymola 2021x, and by default activated. You can however select to use the lighting from previous versions by setting the flag `Advanced.Animation.ExtendedLighting = false`. The flag is by default true.

Animation > Export Animation



Saves the current animation window in any of the following formats:

- AVI (Audio-Video Interleaved), .
- animated GIF.

- VRML format (.wrl).
- X3D as pure XML (.x3d).
- X3D as XHTML (HTML/JS) (.xhtml). When exporting to this file format, an external library, X3DOM, is used.

(This command is also available as **Tools > [Export] Animation**.)

Animation > Reset View



Resets all manipulation of the view (panning, rotation and zooming), however not any object selection (visibility, trace, history, follow, transparency, rotation center).

Animation > Fit View



Fits the animated objects into the animation window. This feature can be useful to find again items that have disappeared outside the window while animating.

Animation > Reset All



Resets any selection made for the object using any of the below commands (for visibility, transparency, trace) but not toggle history or follow.

By selecting an object in the animation window (the object will be marked with red color) and then right-click, the following context menu can be used:

Animation > Visible



Visible is activated by default. Deactivating it makes the object invisible and impossible to select.

Animation > Transparent



Makes the object transparent.

Animation > Show Trace



Toggles showing the trace of the object, that is, the path of the object can be displayed during (and after) simulation. Please observe that the trace of an object can always be seen by selecting it after a simulation, independent whether **Show Trace** has been selected or not.

Animation > Show History



Toggles whether the history frames for the selected object will be shown. To have any effect, a number of history frames must be set using the command **Animation > Animation [Setup]**, the **Frames** tab. Please see section “Frames tab” on page 863.

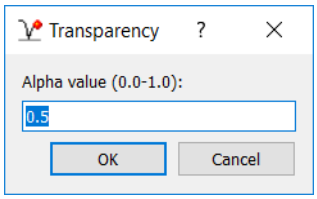
Animation > Follow



The command will to a certain extent (decided by the **Visual** tab in the **Animation > Animation [Setup]** command) fix the selected object resulting other objects having to move relative the fixed object. Please also see section “Visual Tab” on page 862.

Animation > Transparency

This command displays the following menu for setting the transparency:

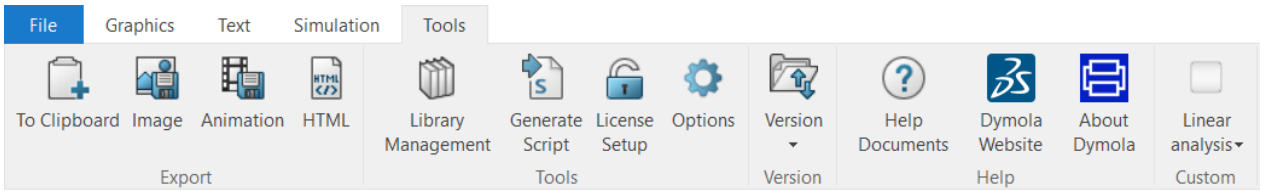


An alpha value of 1.0 means opaque, a value of 0 means invisible. An invisible object is not possible to select.

Animation > Rotation Center

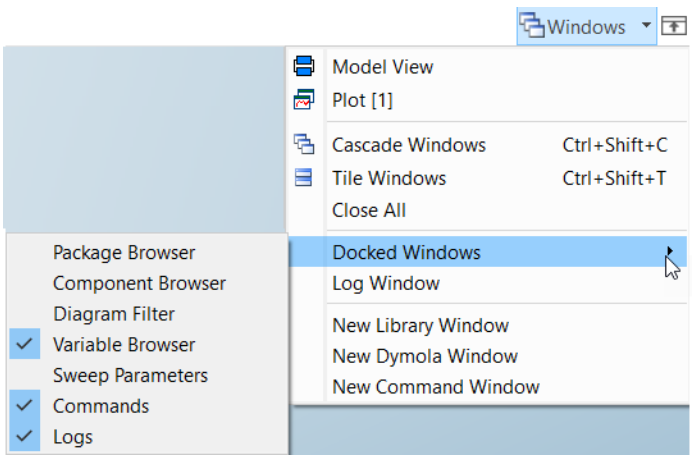
Sets the rotation center to the selected component.

5.3.7 Main window: Tools tab



Please see corresponding section in the chapter “Developing a model”.

5.3.8 Main window: Window menu

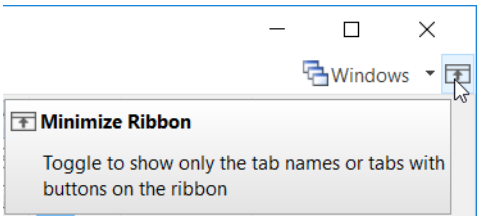


Please see corresponding section in the chapter “Developing a model”.

Clicking directly on the **Window** command will toggle between all present sub-windows; what window should be active. Clicking the arrow next to the command displays a menu where the first part contains all sub-windows available (plot, animation, diagram layer, vizualiser). The alternatives for e.g. plot windows are based on the plot heading (if specified) or the names of the plotted variables. Selecting a window from this list will make it active. This feature is valuable when working e.g. with a number of plot windows. (You can also use shortcuts, **Ctrl+Tab** activates the subwindow one step down in the list relative the currently active subwindow, **Ctrl+Shift+Tab** activates the subwindow one step up in the list relative the currently active subwindow.)

5.3.9 Main window: Toggle button for ribbon information level

At the upper left corner of the Dymola window, the **Minimize Ribbon** button can be used to control what is displayed in the ribbons.



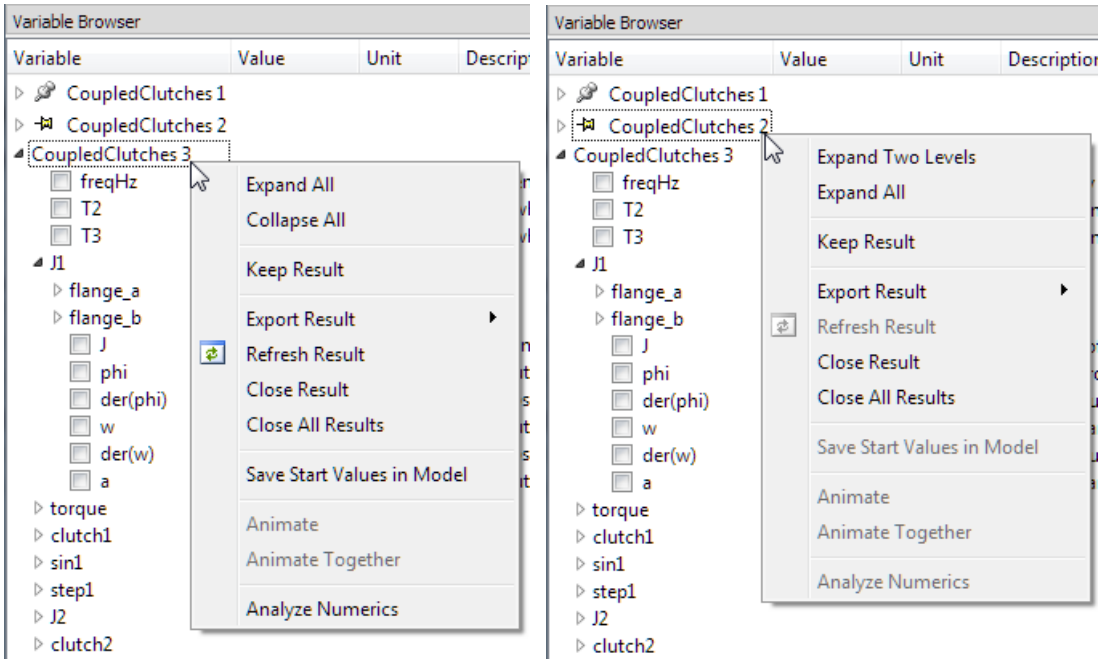
The button toggles between three levels of information:

- Show always tabs with buttons, and group names in the tab (default).
- Show always tab with buttons, but hide group names.
- Show tab with buttons only when tab is active (that is, when you click the tab name to give a command), but hide group name.

5.3.10 Context menu: Variable browser – nodes

The variable browser is introduced in “Variable browser” on page 588. See also “Variable browser interaction” starting on page 619.

The top-level nodes in the variable browser represent simulation result files; other nodes represent the component hierarchy and variables and parameters at the lowest level. The parameters and variables are not considered nodes and have a context menu of their own (see next section). The variable browser has a context menu with several important operations.

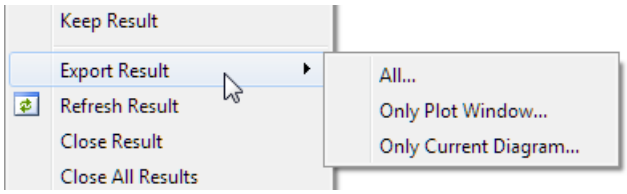


The first three choices are available for all **nodes** in the variable browser.

- Clicking on an expand icon ▷ opens one more level of the tree. The **Expand Two Levels** operation opens two levels, which makes the model structure clearer without creating a huge browse tree.
- The **Expand All** operation opens every sub-node in the browse tree. Large sub-trees may become very large and hard to navigate.
- Clicking on a collapse icon ◢ will close a node. The **Collapse All** operation will close all nodes in the browse tree. The difference is that the next time you open the node, all nodes will be closed.

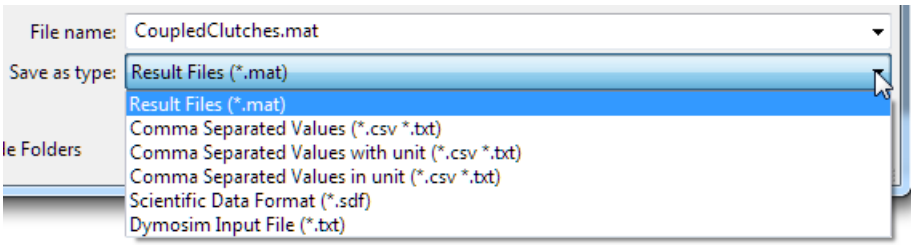
The last commands in the menus are only available for top-level (result file) nodes.

- The **Keep Result** keeps the result file when performing a new simulation. For more information about keeping results, please see section “Selecting which simulation result files to keep when performing a new simulation” starting on page 675.
- The **Export Result** commands allow the user to export the result file in several different file formats. It is possible to export the whole result file, variables in the currently active plot window, or just variables in the currently active plot diagram.



The commands are:

- **Export Result > All…**. This command.exports the whole result file. The possible file formats to export to are:



  Result Files (*.mat): The .mat file format used is Matlab format: easy to use in Matlab and can also be re-opened as a result in Dymola.

  Comma Separated Values (*.csv *.txt): For use in e.g. Excel. The values are exported in SI units. **Note:** some versions of Microsoft Excel uses the Regional Setting of List Separator when reading CSV files, please set this to ','.

  Comma Separated Values with unit (*.csv *.txt): As above, but the values are stored in in displayUnit and the signal name containing unit information.

  Comma Separated Value in unit (*.csv *.txt): As above, but the values are stored in displayUnit, but with no unit information in the signal name.

  Scientific Data Format (*.sdf): Concerning .sdf files, note that you also can convert a result file to .sdf format (Scientific Data Format) by a pre-defined post-processing command. See “Example: Support for Scientific Data Format” on page 790.

  Dymosim Inut File (*.txt): This .txt format can be used to create a file `dsu.txt`. Such a file can be used as an input file for further simulation studies in Dymola. See section “Dymosim as a stand-alone program”, starting on page 893, for more information about this file. Note that it is currently only possible to create such a file for the whole result.

- **Export Result > Only Plot Window…**. This command exports all plotted variables in the currently active plot window, that is, from all diagrams in the currently active plot window.

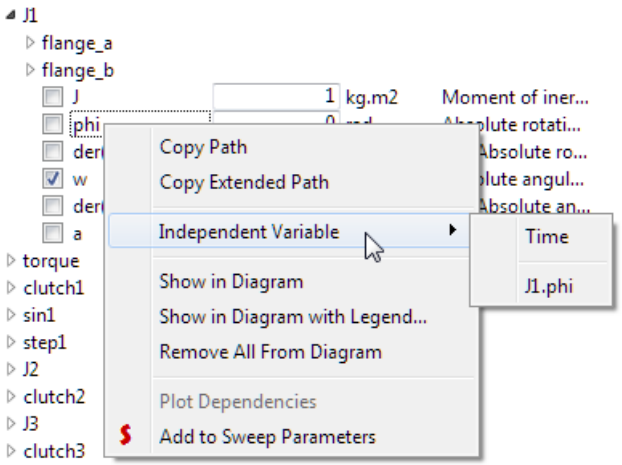
    - **Export Result > Only Current Diagram…**. This command exports all plotted variables in the currently active plot diagram.

- The Refresh Result command reloads the latest corresponding simulation result file without any changes in plot setup etc. Notes:
    - You can only refresh the latest result of simulating a certain model (please compare the context menus in the figure above).
    - Plots are automatically replotted bases on the new signal data.
    - For more information, see "Refreshing simulation result files" on page 678.
- The **Close Result** operation will delete the result file from Dymola and free the occupied storage space.
- The **Close All Results** will close all result files. However, the initial values of the current model will still be present, to enable changing these values and do a resimulation of the current model with better parameter settings. If a new model is opened instead, the initial values will also disappear, since they are no longer relevant. The command corresponds to the default of the built-in function `removeResults()`.
- The **Save Start Values in Model** will save modified initial conditions and parameter values entered in the variable browser to the current model or extend the model before saving the values. See section "Save start values in model" on page 625 for more information.
- The **Animate** operation will animate the data in the selected result file. This operation requires that an animation window is open and that the result file contains animation data. For more information about the command, see section "Animation of one result file" on page 758.
- The **Animate Together** operation enables animating several (selected) result files together in the same window. This operation requires that an animation window is open and that the result files contain animation data. For more information about this command, see section "Animation of several result files together" on page 759.
- The **Simulation Analysis** operation allows you to:
    - Analyze the numeric integration, including displaying time-plot for selected variables, and information about limits step size, dominates error, and more than 10 % of error. The behavior of the integrator can also be studied by displaying internal integrator step size and order. See section "Analyzing numeric integration" starting on page 649.
    - Analyze the events during initialization and simulation, including presenting an overview, analyzing individual events including filtering and plotting of values and iterations. See Section "Event logging user interface" starting on page 654.

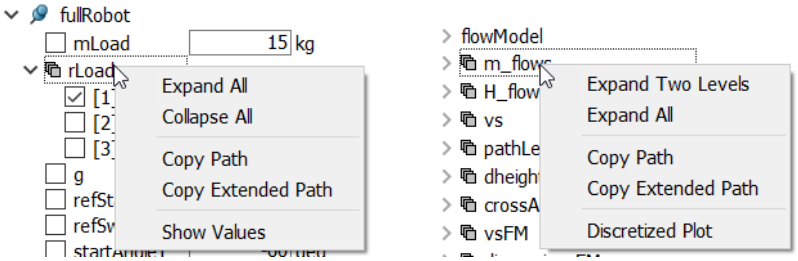
5.3.11 Context menu: Variable browser – signals and arrays

The variable browser is introduced in “Variable browser” on page 588. See also “Variable browser interaction” starting on page 619.

If the context menu is activated when the cursor is beside a signal (parameter or variable) the context menu will look the following.



If the context menu is activated when the cursor is beside an array, the context menu will look the following, depending on if the array is constant or not:



The alternatives in the menus are (for the “Expand/Collapse” alternatives in the array node context menus, see above section):

Copy Path will copy the path of the signal/array to the clipboard.

Copy Extended Path will copy the extended path of the signal/array to the clipboard. The extended path is the signal/array path with result file name and sequence name (in brackets) included.

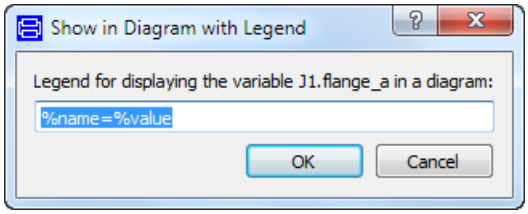
Independent Variable enables selection of what should be the value of the horizontal axis in the plot window. Default is usually Time. For more information, see section “Selecting independent variable” on page 629.

Show in Diagram will display the variable under the relevant component in the diagram layer. Please note that the diagram layer will be presented automatically if not activated by the user. By default, the value is presented in the format:

```
<name> = <value>
```

Please see section "Presenting values in the diagram layer" on page 684 for an example.

Show in Diagram with Legend... makes it possible to change the representation of the variable display. When clicking on this entry, the following window is displayed:



The string `%name` is expanded as the variable's name; the string `%value` is expanded as the variable's value.

Remove All From Diagram will erase all displayed units.

Plot Dependencies will display the dependencies of the variable by plotting them. See section "Plot dependencies" starting on page 633.

Add to Sweep Parameters will add the variable to a parameter sweep, as a "parameter to sweep" in the sweeping dialog. See the chapter "Model Experimentation", section "Sweeping parameters using new GUI".

Add to Sweep Parameters for Plot is similar to the above, the difference is that this commands adds the parameter as a "variable to plot" in the sweeping dialog.

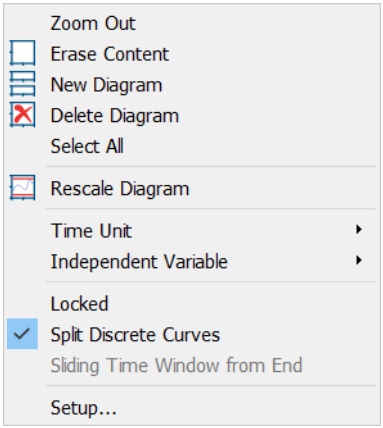
Show Values [only for constant arrays] will display the constant array as a matrix in a table window. See section "Displaying constant arrays as matrices" starting on page 630.

Discretized Plot [only for non-constant arrays] will display the non-constant array as a discretized plot. See section "Plotting a 1D discretization" starting on page 631.

5.3.12 Context menu: Plot window

The plot window is introduced in "Plot window and Plot tab" on page 591. Please also see "Plot window interaction" starting on page 692.

By right-clicking in the plot window, a context menu is displayed. Depending on what object in the plot window is right-clicked (a curve, a legend or an empty space) a number of possible commands are available. Below the context menu for right-clicking in empty space is described. The following sections cover other alternatives.



Zoom out will zoom out to the previous defined zoom level.



Erase Content will erase the content (curves and text objects, if any) in the diagram.



Delete Diagram will delete the currently active diagram.

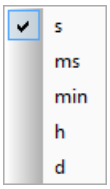
Select All selects all curves in a diagram. Note that texts added in the diagram are not selected.



Rescale Diagram will reset the zooming to the initial value for all signals in the diagram.

Horiz Display Unit (only available when having other variable than time as x-axis) is used to display unit of x-axis.

Time Unit is used to select the time unit for the active plot on the horizontal axis. The selections are the ones available as display units for time. An example is:



The default is always s (seconds). If another time unit is selected, this will be displayed in the legend of the horizontal axis.

Note the difference between this local setting and the simulation time. See section “Changing time unit on x-axis” on page 711 for more information on both settings.

Independent Variable enables selection of what should be the value of the horizontal axis in the plot window. Default is usually Time. For more information, see section “Selecting independent variable” on page 629.

Locked will lock the plot window. When locking is applied, the plot window cannot be changed from outside (by for example selecting/deselecting signals in the variable browser, or by resimulation).

Split Discrete Curves By default, Boolean, Integer, and enumeration values are plotted separated. By unticking this setting, they are plotted overlapping. For more information, see section “Display of Boolean, Integer, and enumeration signals” starting on page 698.

Sliding Time Window from End changes the diagram to a time sliding, that is, the window is dynamically updated with the latest simulation value at the right of the window. For more information, see “Applying time sliding window” on page 711.

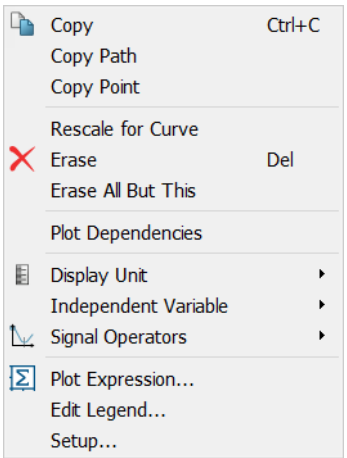
Setup… displays the plot setup menu. See “Plot: Options > Setup” starting on page 842.

Please note that concerning the zooming commands this menu is just a part of the larger zooming concept. Please see section “Plot window interaction” on page 692 for the full picture.

5.3.13 Context menu: Plot window – curve and legend

The plot window is introduced in “Plot window and Plot tab” on page 591. Please also see “Plot window interaction” starting on page 692.

By right-clicking on a curve or a legend, a context menu is displayed.



Depending what is selected, some alternatives are dimmed, and for a parametric curve, also a command **Parameter Levels** is available.

The following alternatives can be available:



Copy will copy the curve values to the clipboard. This enables further processing in other software.

Copy Path will copy the path (for example `J1.w`) to the clipboard.

Copy Point will copy the entire text of the corresponding tooltip to the clipboard.

Rescale for Curve will rescale the active diagram to display the selected curve with maximum scaling.



Erase will erase the selected curve.

Erase All But This will erase all curves but the selected one.

Plot Dependencies (only available if activated) will display dependencies in a plot. See section "Plot dependencies" starting on page 633.

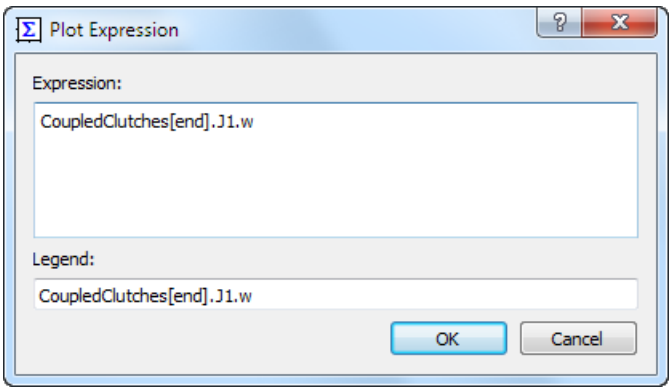
Display Unit makes it possible to change the display unit of signals. See section "Changing the displayed unit of signals" on page 712 for more information.

Independent Variable enables selection of what should be the value of the horizontal axis in the plot window. Default is usually Time. For more information, see section "Selecting independent variable" on page 629.

Signal Operators makes it possible to add signal operators to the curve. Please see section "Displaying signal operators" starting on page 722 for information about this.



Plot Expression… will display a dialog for creating a plot expression, prepopulated with the selected curve; e.g.



Please see section "Plotting general expressions" starting on page 736 for more information.

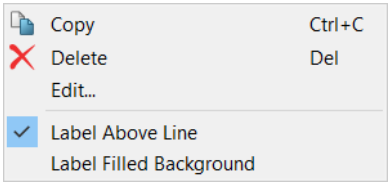
Edit Legend makes it possible to edit the legend of the curve.

Parameter Labels (only available for parametric curves) will display parameter labels along the curves.

Setup… displays the plot setup menu. See "Plot: Options > Setup" starting on page 842.

5.3.14 Context menu: Plot window – signal operators

Signals operators for curves are introduced in "Displaying signal operators" starting on page 722. Right-clicking a signal operator will display a context menu:



with the following alternatives:

Copy will copy the value of the signal operator to the clipboard.

Delete will delete the signal operator.

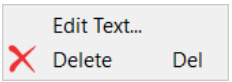
Edit… will display the dialog for the selected signal operator; that makes it possible to change it.

Label Above Line will by default display the label of the signal operator above the line indicating the operator. By deselecting it, the label will be displayed below the operator.

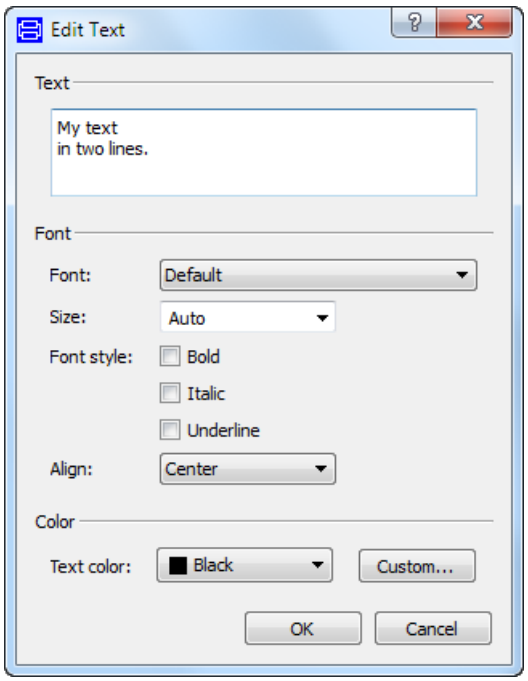
Label Filled Background will add a background under the label; making the label always possible to see when for example a number of curves interfere.

5.3.15 Context menu: Plot window – text objects

Texts inserted in plots are introduced in “Insert and edit text objects in the plot” on page 751. Such texts have specific context menus. Right-clicking a text will display the context menu.



Edit Text… will display the dialog box of editing text objects. An example:



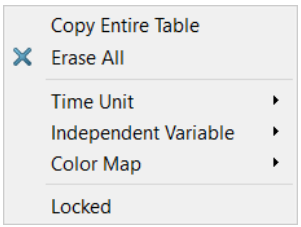
Delete will remove the text object from the plot.

For more information about this menu, please see “Texts” in previous chapter, section “Basic model editing”, sub-section “Creating graphical objects”.

5.3.16 Context menu: Table window

Displaying curves as values in a table window instead of curves in a plot window is introduced in “Displaying a table instead of curves” on page 740. Right-clicking in such a table, two context menus are available.

Right-clicking in an empty area displays the following context menu:



The following alternatives can be available:

Copy Entire Table will copy the entire table to clipboard.

Erase All will erase the table.

Time Unit will make it possible to change the time display unit for the active table window, like in a plot window. See section “Changing time unit on x-axis” on page 711.

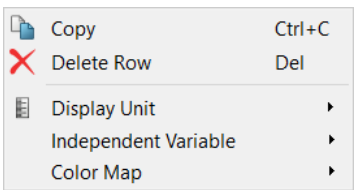
Independent Variable enables selection of what should be the value of the horizontal axis in the plot window. Default is usually Time. For more information, see section “Selecting independent variable” on page 629.

Color Map enables you to apply color-coding of table cells to simulation result file tables to indicate approximately the values. For more information, see section “Applying color-coding to simulation result file tables” on page 742.

Locked will lock the table window. When locking is applied, the table window cannot be changed from outside (by for example selecting/deselecting signals in the variable browser, or by resimulation).

5.3.17 Context menu: Table window - value

Right-clicking a value in a table window displays the following context menu:



The following alternatives can be available:

Copy will copy the selected content to the clipboard.

Delete Row will delete selected row.

Display Unit makes it possible to change the display unit, like in a plot window. See section "Changing the displayed unit of signals" on page 712.

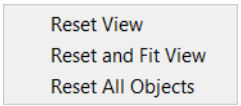
Independent Variable enables selection of what should be the value of the horizontal axis in the plot window. Default is usually Time. For more information, see section "Selecting independent variable" on page 629.

Color Map enables you to apply color-coding of table cells to simulation result file tables to indicate approximately the values. For more information, see section "Applying color-coding to simulation result file tables" on page 742.

5.3.18 Context menu: Animation window

The animation window is introduced in "Animation window and Animation tab" starting on page 596. Please also see "Animation window interaction" starting on page 760.

If no object is selected when taking up the context menu, you get:



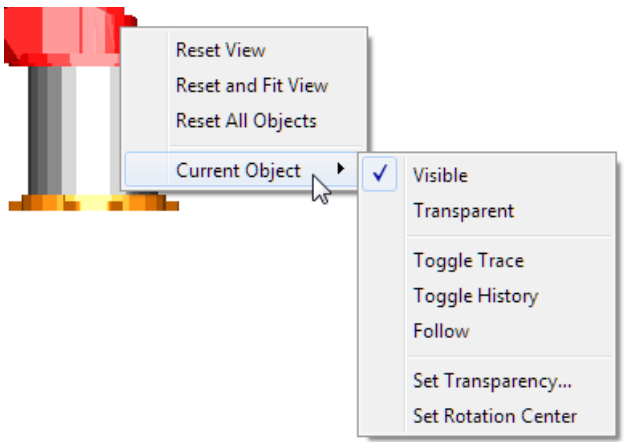
Reset View resets all manipulation of the view (panning, rotation and zooming), however not any selection in **Current Object** in this menu.

Reset and Fit View performs the above reset view command, but after the reset also fits the animated objects into the animation window. This feature can be useful to find again items that have disappeared outside the window while animating.

Reset All Objects will reset any selection made for the object using **Visible**, **Transparent**, **Toggle Trace** and **Set Transparency...** but not **Toggle History** or **Follow**.

By selecting an object in the animation window (the object will be marked with red color) and then right-click, the following context menu can be used:

The context menu of the animation window.



For the three first entries, see above.

Current Object gives the following subentries:

Visible is checked by default. Unchecking it makes the object invisible and impossible to select.

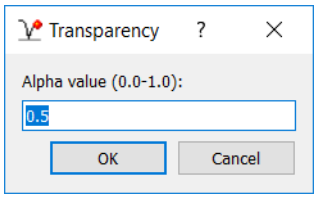
Transparent makes the object transparent.

Toggle Trace will toggle the trace of the object, that is, the path of the object can be displayed during (and after) simulation. Please observe that the trace of an object can always be seen by selecting it after a simulation, independent whether **Toggle Trace** has been selected or not.

Toggle History will toggle whether the history frames for the selected object will be shown. To have any effect, a number of history frames must be set using the command **Animation > Setup…> Frames** tab. Please see section “Frames tab” on page 863.

Follow will to a certain extent (decided by the **Visual** tab in the **Animation > Setup…** command) fix the selected object resulting other objects having to move relative the fixed object. Please also see section “Visual Tab” on page 862.

Set Transparency… will display the following menu for setting the transparency:



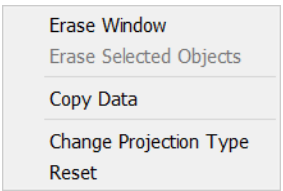
An alpha value of 1.0 means opaque, a value of 0 means invisible. An invisible object is not possible to select.

Set Rotation Center sets the rotation center to the selected component.

5.3.19 Context menu: Visualizer window

The visualizer window is introduced in “Visualizer window” on page 598.

By right-clicking in a visualizer window, the following context menu will appear:



Erase Window will erase all objects in the window.

Erase Selected Objects will erase the selected objects.

Copy Data will copy the 3D plots of the visualizer window to the clipboard, for further use in for example MS Excel, The data is copied in matrix form, x, y, and z. If you have more than one 3D plot in the window, the matrices are named x1, y1, z1, x2, y2, z2, etc.

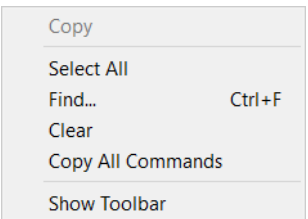
Change projection type will toggle between orthogonal projection (angles are preserved) and perspective projection.

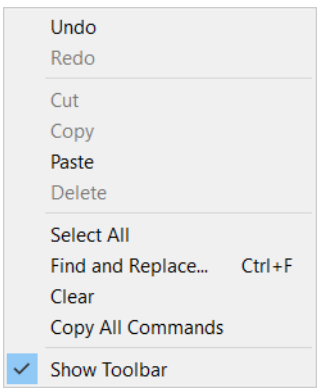
Reset will reset the objects in the window.

5.3.20 Context menu: Command window – Command log pane

The command log pane of the command window is introduced in “The command log pane” on page 600.

By right-clicking in the command log pane of the command window, any of the following context menus will be shown, depending on if you have activated the toolbar or not (by default not activated):





The commands that can be found in any of these menus are:

Undo, Redo undoes and redoes the last editing operation in the text editor.

Cut, Copy, Paste copies text between the clipboard and the editor. It is possible to copy text with hidden annotations to clipboard.

Delete will delete selected text. Please note that the command **File > Clear Log** can be used to clear all content of the command log pane.

Select All selects all text in the displayed command log.

Find and Replace… finds and replaces text in the text editor. For more information, see the command Find and Replace in the section "Context menu: Script editor" starting on page 885. *Please note* however that there is a specific setting for the command window if all found occurrences of whole words should be highlighted. It has the same name as the general setting, **Highlight all matches**, but is located in the **Command Window** group of the **Text Editor** tab of the command **Tools > Options**. (The general setting, which is not applicable for the command window, is located in the **Modelica text editor** group.) The setting is by default not activated, for performance reasons – the command log might be quite big.

Clear clears the command log pane.

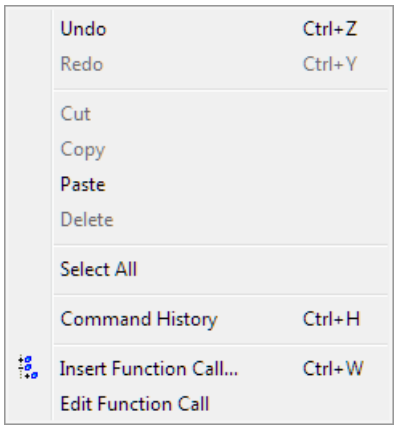
Copy All Commands copies the commands (not the results) in the command log. This command is used when the commands should be used in scripting – the copied commands can be inserted in the Modelica Text layer of a scripting function.

Show Toolbar is by default not activated – the command log pane is only a log of the commands; it is not possible to add or edit the content in the pane. If the toolbar is activated, you can add and edit content, it works like a documentation editor; and the corresponding toolbar is displayed. The button works in toggle mode. Note that the status can be different in, for example, the internal command window and a stand-alone command window. There is a flag `Advanced.CommandLogToolbar` available that corresponds to the state of the internal command window. This flag is by default `false`.

5.3.21 Context menu: Command window – Command input line

The command input line of the command window is introduced in "The command input line" on page 601.

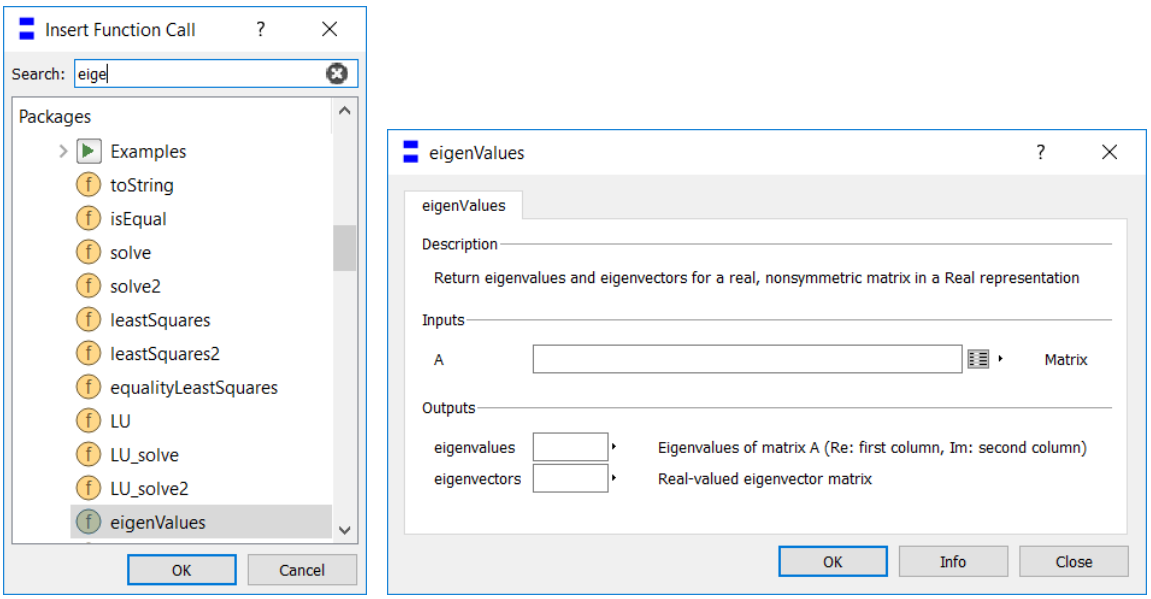
By right-clicking in the command input line of the command window, the following context menu will be shown:



For all commands except the last three, please see the above section.

Command History opens the command history window.

Insert Function Call… enables searching (or browsing) for a function and then entering the arguments, allowing easy insertion of a function call. As an example, entering `eige` in the search input field (in order to find `Modelica.Math.Matrices.eigenValues`) will give the following result (followed by the resulting parameter dialog of the function that will be displayed when **OK** is clicked):



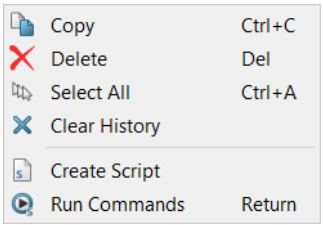
For more information how to use this function, please see section “Basic facts about scripting in Dymola” starting on page 904.

Edit Function Call allows you to modify the arguments inside a function call using a parameter dialog. Select a function call up to ending parenthesis (or put the cursor inside name), right-click and select the command.

5.3.22 Context menu: Command history window

The Command history window was introduced in section "Working with the command history" starting on page 780.

Right-clicking in the command history window displays a context menu with the following choices:



Copy copies selected parts of the command history to the clipboard.

Delete deletes the selected parts of the command history.

Select All selects all content in the commands history window.

Clear History clears all content in the commands history window.

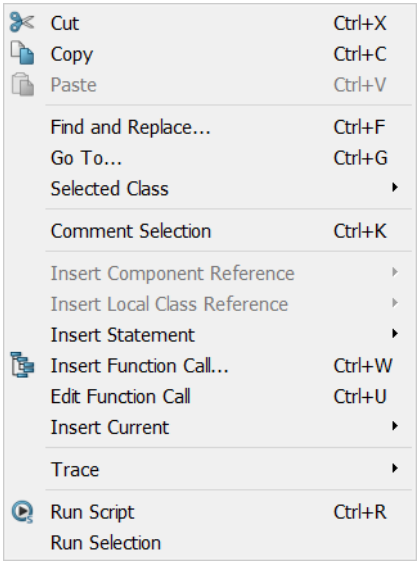
Create Script creates a script in the scripting window and inserts the selected parts from the history window.

Run Commands executes the selected parts of the command history.

5.3.23 Context menu: Script editor

The Dymola script editor was introduced in "Script Editor window and Script Editor tab" starting on page 602.

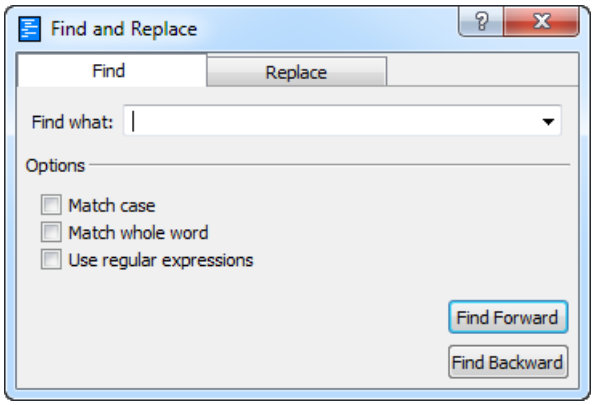
Right-clicking in the script editor displays a context menu with the following choices:



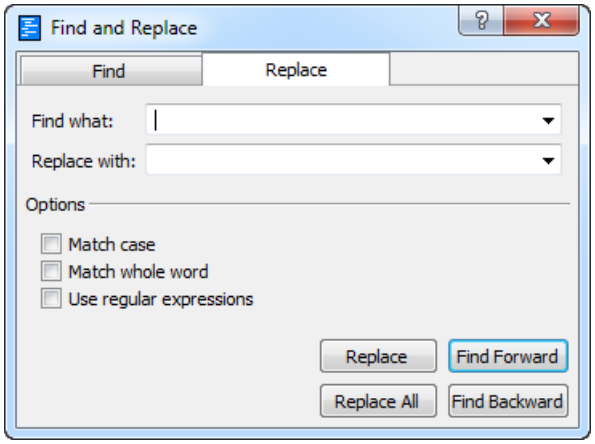
Cut, Copy, Paste copies text between the clipboard and the editor. It is possible to copy text with hidden annotations to clipboard.

Find and Replace finds and replaces text in the text editor. One tab is for find only, one tab is for find and replace.

Find tab.



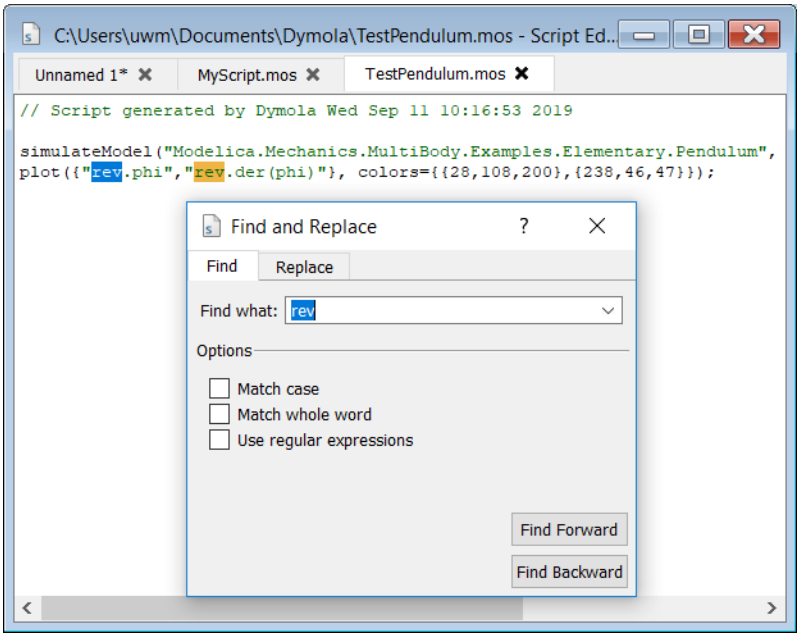
Replace tab.



If the text is read-only, only the **Find** tab is accessible; the **Replace** tab is dimmed.

If you have the cursor on a word when activating the command, this word will be prefilled in the **Find what** box.

When you click **Find Forward** or **Find Backward**, you will find the next occurrence of the searched word/text going forward or backward in the text. All occurrences of whole words will also be highlighted in orange; the hit will be in blue.



Note the difference between the **Find** command and the highlighting in orange; to avoid highlighting in orange of e.g. single characters, only whole words are highlighted in orange. As an example, the `Title` occurrence in `leftTitle` in the second line of the script above will be found by the **Find** command, but it will not be highlighted in orange since it is not a whole word.

You can disable the feature of highlighting all found occurrences of whole words by unticking the setting **Highlight all matches** in the **Modelica text editor** group, in the **Text Editor** tab of the command **Edit > Options…**. This corresponds to setting the flag `Advanced.HighlightAllMatches = false;`.

The key **F3** is shortcut for the command **Find Forward**; the keys **Shift+F3** are shortcut for the command **Find Backward**.

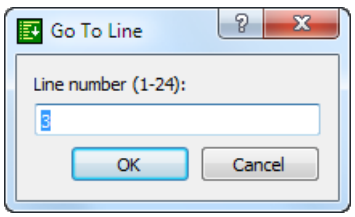
The text window is automatically scrolled to display the found occurence. If the occurrence is inside an annotation, that annotation will be expanded. A warning is given if the text is not found.

Search history is available by clicking the arrow in the end of the **Find what** field. Replace history is available clicking the arrow in the end of the **Replace with** field.

Regular expressions can be used in the search if this checkbox is checked. Special symbols in the regular expression are

*	Match everything.
?	Match any single character.
{ab}	Match characters a or b.
{a-z}	Match characters a through z.
{^ab}	Match any characters except a and b.
E+	Match one or more occurrence of E.
(ab|cd)	Match ab or cd.
\d	Match any digit.
\w	Match any digit or letter.
^	Match from start.
$	Match at end.

Go To… sets the cursor at specified line and scrolls window if necessary. The number of the current line is shown as default.



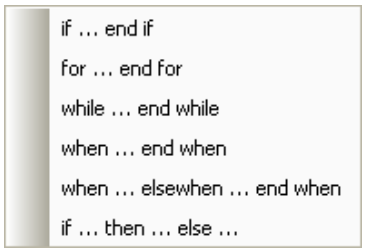
Selected Class can be used to opens the selected class in the current tab, in a new tab, or displays the info of t the class.

Comment Selection can be used to comment out selected rows.

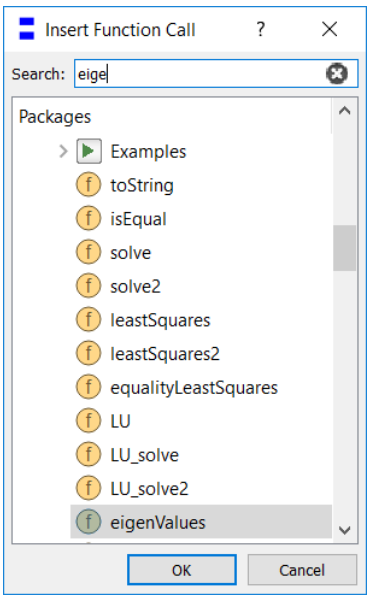
Insert Component Reference enables inserting a component reference in the script.

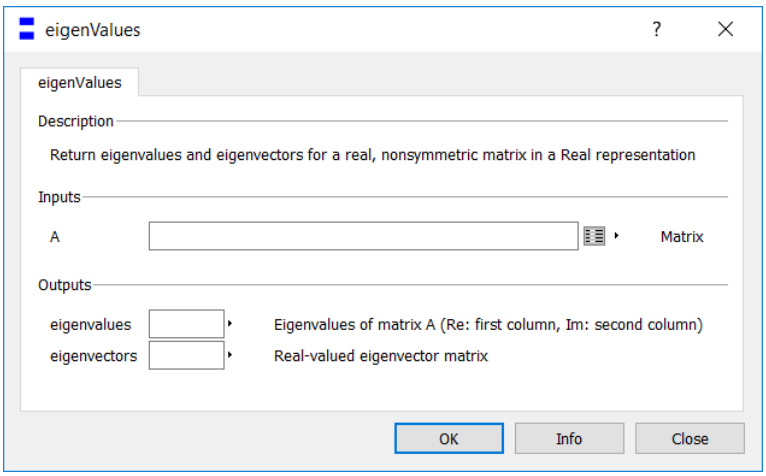
Insert Local Class Reference enables inserting a local class reference in the script.

Insert Statement presents a submenu with common Modelica constructs. This makes it easier to enter Modelica code.



Insert Function Call... enables searching (or browsing) for a function and then entering the arguments, allowing easy insertion of a function call. As an example, entering `eige` in the search input field (in order to find `Modelica.Math.Matrices.eigenValues)` will give the following result (followed by the resulting parameter dialog of the function that will pop when **OK** is clicked):

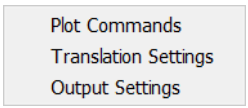




For more information how to use this function, please see “Basic facts about scripting in Dymola” on page 904.

Edit function call allows you to modify the arguments inside a function call using a parameter dialog. Select a function call up to ending parenthesis (or put the cursor inside name), right-click and select the command.

Insert Current presents a submenu with inserting current plot commands and settings in the script:



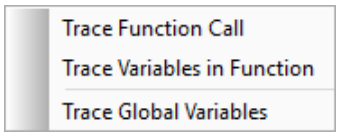
Plot Commands inserts the current plots (your displayed plots) in the script editor. This facilitates the creation of scripts for rebuilding a plot session that has been done interactively. For an example, see “Insert current plot commands in the script” on page 930.

Translation settings inserts translation settings, corresponding a selection of settings from the **Translation** tab in the simulation setup, in the script. For more information, see “Inserting translation settings in the script” on page 931.

Output settings inserts output settings, corresponding to a selection of settings from the **Output** tab of the simulation setup, in the script. For more information, see “Inserting output settings in the script” on page 932.

Run Selection executes the selected part of the displayed script, as was it a separate script.

Trace presents a submenu with a number of alternatives of tracing:

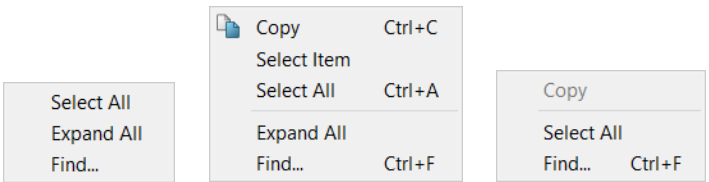


For more information about tracing, please see section “Tracing using the Dymola script editor” on page 932.

5.3.24 Context menu: Log window

The log window is introduced in “Log window” on page 603.

Right-clicking in the log window presents a menu with the following choices, depending on which tab you are working in:



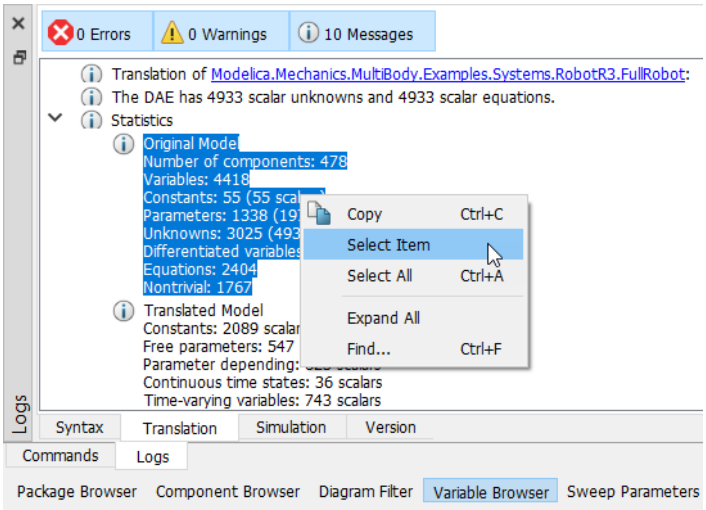
The image to the left shows the context menu for the **Syntax** tab, the middle image shows the context menu for the **Translation** tab, and the image to the right shows the context menu for the **Simulation** tab and **Version** tab.

The menu alternatives are:

Copy – copies the selected text to the clipboard.

Copy Link Location – copies selected link to clipboard (icons cannot be copied).

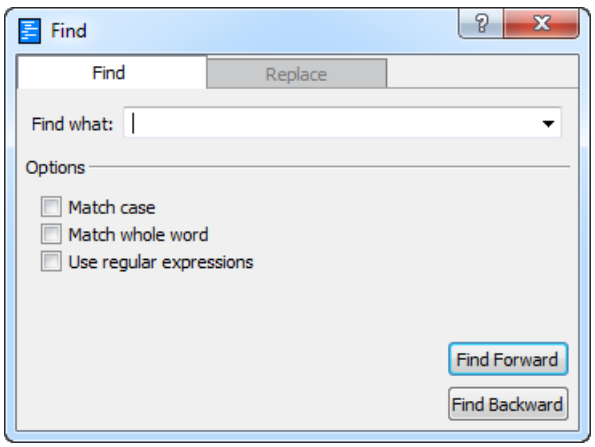
Select Item – selects the active node; this can be useful when wanting to copy a large node to the clipboard.



Select All – selects all text.

Expand All – expands all nodes in the messages tree structure.

Find… – finds text. The menu looks like:



For more information, see the command **Find and Replace** in the section "Context menu: Script editor" starting on page 885. (Only the Find tab is relevant here.)

5.4 Dynamic Model Simulator

5.4.1 Overview

What is Dymosim?

Dymosim stands for **Dy**namic **mo**del **sim**ulator and is the executable generated by Dymola in order to simulate the model, and then used to perform simulations and initial value computations. Dymosim contains the code necessary for continuous simulating, and event handling. Three types of events are supported: time-, state- and step-events. Model descriptions are compiled to machine code such that maximum execution speed is reached during simulation; no interpretation of model equations takes place.

Dymosim is a stand-alone program which can be used in several different environments. It is especially suited to be used in conjunction Dymola. Dymola transforms such a definition into a state space description which in turn is solved by the Dymosim integrators. The results of a Dymosim simulation can be plotted or animated.

Dymosim can be compiled as a Windows application with built-in DDE server with real-time capability. For more information, please see the chapter "Simulation Environments".

In the 64-bit version of Dymola, the user can decide if models should be compiled as 32-bit or 64-bit executables, using the flag `Advanced.CompileWith64`. The flag has three possible values:

- 0 (default value); If external libraries only exist as either 32-bit or 64-bit the variant that allows linking is selected. Otherwise dymosim.exe is compiled as 32-bit, dymosim with DDE server and dymosim.dll are compiled as 64-bit. There is a demand for correct library location, see the second note below.
- 1 All applications above compiled as 32-bit.
- 2 All applications above compiled as 64-bit.

If the user has set `Advanced.CompileWith64=1` or `Advanced.CompileWith64=2`, and the corresponding library is missing, a warning will be generated, but linking attempted with the selected setting.

If some libraries only exist in 32-bit version, and some only in 64-bit version, a warning is generated, but linking attempted with the selected setting.

Notes:

- Concerning FMUs, 64-bit FMUs will normally also contain 32-bit dlls. There is a specific setting for what FMU binaries to generate, see section "FMI Export tab" on page 822. Note that, due to this, the CompileWith64 flag does not affect FMU export; it is instead

controlled by the specific setting. In particular, the export will fail if any required variant (32-bit or 64-bit) of external libraries is missing.

- When using the default value of the CompileWith64 flag, it is important that external libraries are located in the recommended win32/win64 folder; otherwise, the compilation might not work as expected. As an example, if an external library is not located in the win64 folder, it will be compiled as 32-bit, even if was intended to be compiled as 64-bit. For more about this, see the index entry "*external libraries : recommended location*".

5.4.2 Running Dymosim

When translating a model in Dymola an executable, Dymosim, is generated, and later when selecting **Simulation > Simulate** or similar commands this program computes the solution. A statistics of the simulation run is always stored in file `dslog.txt`, which can be displayed by the menu command **Simulation > Show Log**. By using the command **Simulation > Setup** and the tabs **Output** and **Debug**, additional debug information etc. can be written to the log file. Please see section "Simulation > Setup" starting on page 799 for more information.

By default, Dymosim stores the results of a simulation run in (Matlab) binary format on file (`dsres.mat`). The data can be plotted using the plot functionality in Dymola; please see relevant sub-sections in section "Model simulation" starting on page 614 for more information. As explained in the next section, the results can also be directly loaded into Matlab and plotted with the Matlab plot functions.

(For more information about Matlab please see the chapter "Simulation Environments", section "Dymola-Matlab interface".)

Dymosim as a stand-alone program

Dymosim is essentially a stand-alone program without any graphical user interface which reads the experiment description from an input file, performs one simulation run, stores the result on an output file and terminates. Instead of controlling this action via Dymola's graphical user interface, Dymosim can also be called directly by the user.

A default input file for a simulation run, named `dsin.txt`, is generated by command "`dymosim -i`" in the shell (to get all possible command input line arguments and additional info use "`dymosim -h`"). The file contains the complete information about a simulation run, especially the stop time, the initial values for the state variables and the actual values of the model constants (= Dymola parameters). Use a text editor to overwrite the default values provided in the file.

A simulation run is executed by command "`dymosim`" or by "`dymosim dsin.txt dsres.mat`". In both cases the simulation run is performed by reading the input file `dsin.txt` and by storing the simulation result on the binary file `dsres.mat`.

If the file `dsu.txt` is present when Dymosim is started, Dymosim reads the trajectories of input signals from this file. Such a file can be created in two ways:

- Exporting the result from the variable browser as `dsu.txt`. See section "Context menu: Variable browser – nodes" on page 869 for more information about the different **Export Result** context commands.

- Using commands in Matlab. See the end of below section.

Working in Matlab

The data in `dsres.mat` can be plotted by Dymola. The result file can also be imported into Matlab by executing the command "`d=dymload`" in the Matlab environment. d is a Matlab STRUCT-variable containing the following fields:

.fname	*the file name*
.pname	*the path name*
.nnames	*the number of variable names*
.ndatamat	*the number of data matrices*
.name	*the names of the variables (each row a name)*
.description	*the description of the variables (each row a description)*
.dataInfo	*The dataInfo-array*
.data_#	*The data matrices. # ranges from 1 to .datamat*

The command "`d=dymload`" loads data from `dres.mat`, while the command "`d=dymload(`*`filename`*`)`" loads data from a file *`filename`*`.mat`.

If the simulation result for a specific variable is wanted, it can be loaded using the command "`d=dymget(dymstr, name)`" where dymstr is a structure obtained by executing dymload, name is the full variable name and d is a vector containing the data.

The command "`dymbrowse(`*`action`*`)`" makes it possible to interactively browse, plot and compare Dymola simulation results in Matlab environment. The **Browse** button in the GUI is used to select the result file. A specific filename can also be specified directly in the call to the dymbrowser function. The third alternative is to use the dymload function as argument.

The dymtools commands above (and some more) can be found in the folder Program Files\Dymola 2023\mfiles\dymtools. More information about all these commands can be found using help dymtools in Matlab.

A list of the most usable commands is given in section "Dymosim m-files" on page 902 (including some older commands).

Some older and much more memory-consuming commands are still supported; using them the result file can be imported into Matlab by executing the command "`[s,n] = tload`" in the Matlab environment. The signal names are stored in text matrix "`n`", whereas the simulation results are stored in the numeric matrix "`s`". The first column of "`s`" is the time vector, whereas a subsequent column "`i`" corresponds to signal "`i`". Signal indices and corresponding names are printed by the provided Matlab m-function "`tnlist (n)`". Signal "`i`" can be plotted by Matlab command "`plot(s(:,1),s(:,i))`". Alternatively, the provided Matlab m-function "`tplot (s,[i1,i2,i3],n)`" plots columns `i1,i2,i3` of "`s`" with respect to "`s(:,1)`", using the corresponding signal names stored in text matrix "`n`" as a legend.

In order to save memory, particularly when using the older command tload, the user can ensure that only the necessary signals are stored in the result file. This can be accomplished

by plotting the appropriate signals and then selecting **Save As...** from the context menu of the result file in the variable browser, and selecting the file type "Matlab file - only plotted".

As mentioned above, the file `dsu.txt` can also be created using Matlab, to enable resulting input signal trajectories to be used as input to another simulation. It is possible to generate an input function within Matlab, and save the data on file, e.g.:

```
t = (0:100)'/20;
s = [t, t.*t, sin(t)];
n = ['time     '
     'control  '
     'dist     '];
tsave ('dsu.txt',s,n);
```

Here, "`control`" and "`dist`" are names of variables which are declared as "`input`" at the highest hierarchical level in the Dymola model. Note that the strings have to be of the same length (that is the reason for the number of blanks in the example).

Note that the option **Generate Result** in the DymolaBlock GUI of the Dymola-Simulink interface contains creating a `dsu.txt` file as well. Please see the chapter "Simulation Environments", section "Dymola – Matlab interface" for more information about this.

5.4.3 Selecting the integration algorithm

Dymosim provides a number of different integration algorithms for the simulation of dynamic systems. In this section the major characteristics of these algorithms are discussed, and rules of thumb are given which algorithm should be selected for a problem at hand. Note however, that one should not rely on just one integration algorithm for simulation experiments. Instead, some selected results should be checked by two or three other (different) integration algorithms.

First of all, some important issues of integrators are explained, in order to classify the available integration algorithms in Dymosim. If you are familiar with these issues, just skip the next section.

Integrator properties

Relative and absolute error tolerances

Relative tolerances have to be assigned to tell the integration algorithms how accurately the solution **x**(t) should be computed. The tolerance must be greater than zero and is used in a local error test for each component[12] of the state vector x_i, roughly requiring at each step that

```
| local error | < tolrel * |xi| + tolabs
```

[12] More specifically, a root-mean-square norm is used to measure the size of vectors, and the error test uses the magnitude of the solution at the beginning of a step.

The relative tolerance `tolrel` approximately defines the number of expected true digits in the solution. That is, if 3-4 true digits are required in the solution, the default value of 10^{-4} should be used. If an element of **x** is exactly zero or near to zero, the relative tolerance is without meaning and the absolute tolerance `tolabs` approximately defines an upper (absolute) limit on the local error. Since user's often have difficulties to see the difference between the relative and the absolute tolerance, in Dymosim `tolabs` equals `tolrel` times the nominal attribute of the variable. If no nominal value is given, Dymosim normally uses a default of 1.

Global error

The *global error* is the difference between the true solution of the initial value problem and the computed approximation. Practically, all present-day codes, including the ones used in Dymosim, control the local error at each step and do not even attempt to control the global error directly. Usually, but not always, the accuracy of the computed state variables **x** is comparable to the relative error tolerances. The algorithms will usually, but not always, deliver a more accurate solution if the tolerances are reduced. By comparing two solutions with different tolerances, one can get a fairly good idea of the true error at the bigger tolerances.

Step considerations

One-step algorithms versus multi-step algorithms

One-step (or Runge-Kutta) algorithms are basically designed such that they start fresh on every step and thus the cost of restarting them after an event is substantially reduced compared to multi-step algorithms such as lsodar (implementing Adams algorithms) and dassl (implementing BDF algorithms). However, even if the algorithms are one-step algorithms the implementation often uses more information from the previous step.

Variable step size, dense output

Most integration algorithms available in Dymosim have a *variable step size* algorithm. At every step, an algorithm estimates the local error. The integration step size is chosen in such a way, that the local error is smaller than the desired maximum local error, defined via the relative and absolute tolerances. This implies, that usually smaller step sizes are used, if smaller tolerances are defined. In other words, a variable (or adaptive) step size implies that the algorithm adapts the step size to meet a local error criterion based on the tolerance.

There is one important difference between integrators with respect to the step size algorithm: The step sizes of some integrators are fixed and given by the output time grid without considering the tolerance. The output time grid is defined by the `StartTime`, the `StopTime` and an output grid size and determines the points, at which results must be stored. The mentioned integrators just proceed from one grid point to the next one, i.e. the maximum possible step size of these integrators is limited by the distance of two successive output points. The step size is always chosen in such a way, that the integrator meets the grid points exactly. For such algorithms, the output grid must be defined carefully. In order to handle stability problems, it is possible to define a smaller fixed step size, and in this case the output step size should be a multiple of this smaller fixed step size.

On the other hand there are integration algorithms, called *dense output* algorithms, which treat output points differently. The step size of such integrators is primarily chosen according to

the required tolerance and the estimated local error. Such algorithms integrate past the desired output points and determine the values of the state variables **x** at the output points by interpolation, which involves no evaluation of the differential equation. Dense output implies that the algorithm can handle state events efficiently and also produce evenly spaced output. The dense output has traditionally been added as an afterthought to the algorithms. Good exceptions are the cerk algorithms, where the algorithm coefficients were optimized including the dense output.

In Dymosim, the maximum allowed step size for dense output integrators can be explicitly restricted by the parameter `algorithm(hmax)` in the input file, additionally it is possible to turn off *dense output*. There are also, for advanced users, flags to control the step size, see section "Advanced setup for simulations by flags" on page 782.

Variable order

Integration algorithms approximate the solution **x**(t) internally by a polynomial of order k_{ord}. Some algorithms use a fixed order; other algorithms vary the order during the simulation. Fixed order means that you manually select the algorithm including order (where higher order should be used for stricter tolerance) instead of the solver automatically adapting the order as for lsodar and dassl.

The integration step size can be usually chosen larger (for the same maximum local error) if the order is bigger, which in turn implies a greater efficiency. The step size and error control of the integrators are based on the assumption, that the solution **x**(t) can be differentiated at least $k_{ord}+1$ times. Therefore, if it is known for example that the result of a system is not very smooth, a low order algorithm should be chosen.

In Dymosim, the maximum order of a variable order algorithm can be explicitly set in the input file via `algorithm(ordmax)`. If the maximum order is set to one, the variable order integration algorithms reduce to the simple (explicit or implicit) Euler formula.

Stiff systems, A-stable algorithms

"Usual" integrators get in trouble, if "fast" and "slow" signals are present in the solution **x**(t) (in other words, where the fastest time scale in the model is substantially faster than the interesting dynamics.) For linear differential equations this corresponds to problems where the system matrix has eigenvalues whose real part is negative and large in magnitude, compared to the reciprocal of the time span of interest. Such systems are called stiff. There exist different algorithms for the integration of stiff and non-stiff systems.

However, note that problems with (undamped) highly oscillating signals are not called stiff here. At present there exists no production code to cope with this problem satisfactorily (the step size is limited by the frequency of the highly oscillating components and therefore the simulation is slow).

The step size of an integration algorithm is always limited by an algorithm specific stability boundary. The integration only remains stable and produces reliable results, if the step size is lower than this boundary. If a system is integrated with a non-stiff integration algorithm, and the integrator step size is limited by the stability boundary and not by the maximum local error, the system is stiff. This usually means that the step size chosen from the non-stiff algorithm becomes very small and the efficiency of the integration degrades considerably.

However, note that the stiffness depends on the chosen error tolerances. If the error tolerances are made stricter, the system may become non-stiff, since the step size is now limited by the maximum local error and no longer by the stability boundary.

A number of the stiff algorithms are also designed to be A-stable, i.e. stable for all stable linear systems. This means that the algorithms are better suited for poorly damped stiff systems (i.e. with poles close to the imaginary axis). Furthermore, since they start with higher order they are more suited for systems with discontinuities or events.

Dymosim integrators

All integrators support the entire range of events present in Modelica models, i.e. state events, time events, and dynamic state selection.

Variable step size integration algorithms

Dymosim provides a number of *variable step size* integration algorithms. The most important characteristics of these algorithms are given in the following table[13]:

Algorithm	Order	Stiff	A-stable	Dense output	Root finder	Method
LSODAR	1-12, 1-5	Both	No	Yes	Yes	Multi-step/ Adams methods
DASSL	1-5	Yes	No	Yes	Yes	Multi-step/BDF
Radau IIa	5	Yes	Yes	Yes	Yes	Single-step/Runge-Kutta
Esdirk23a	3	Yes	Yes	Yes	Yes	Single-step/Runge-Kutta
Esdirk34a	4	Yes	Yes	Yes	Yes	Single-step/Runge-Kutta
Esdirk45a	5	Yes	Yes	Yes	Yes	Single-step/Runge-Kutta
Dopri45	5	No	NA	Yes	Yes	Single-step/Runge-Kutta
Dopri853	8	No	NA	Yes	Yes	Single-step/Runge-Kutta
Sdirk34hw	4	Yes	Yes	Yes	Yes	Single-step/Runge-Kutta
Cerk23	3	No	NA	Yes	Yes	Single-step/Runge-Kutta
Cerk34	4	No	NA	Yes	Yes	Single-step/Runge-Kutta
Cerk45	5	No	NA	Yes	Yes	Single-step/Runge-Kutta
Cvode[14]	1-5	Yes	No	Yes	Yes	Multi-step/BDF

Fixed step size integration algorithms

There are also four different fixed step size integration algorithms, intended for real-time simulations:

[13] (The algorithms from Radau IIa and onwards (except Cvode) in this table are sometimes referred to as "Godess solvers" (from Generic Ordinary Differential Equations Solver Systems).)
[14] For further details about the SUNDIALS CVODE solver, see https://computation.llnl.gov/casc/sundials/main.html.

Algorithm	Order	Stiff	Dense output	Root finder
Euler	1	No	No	Yes
Rkfix2	2	No	No	No
Rkfix3	3	No	No	No
Rkfix4	4	No	No	No

When should a certain algorithm be used?

A number of algorithms are available for the experienced user to choose from. For users wanting some tip where to start, the following can be of help:

If the model does not contain many events, the traditional LSODAR can be used if the model is stiff on some time interval and non-stiff on other time intervals - otherwise DASSL might be used.

If the model may contain many events, use the single-step algorithms. If the system is known to be non-stiff Cerk or Dopri can be used, otherwise Radau IIa, Esdirk or Sdirk can be used. Use higher order if stricter tolerance is wanted.

DAE mode

Dymola supports DAE (Differential-Algebraic Equations) mode. DAE mode is activated by setting the flag

```
Advanced.Define.DAEsolver = true
```

(The flag is by default false.)

DAE mode is supported by the following solvers in Dymola:

- DASSL
- Radau IIa
- Esdirk23a, 34a, 45a
- Sdirk34hw

A translated model may contain several nonlinear equation systems. (These are listed in the translation log under Statistics.) When DAE solver is not enabled these equations are solved numerically whenever the integrator calls the model. When DAE solver is enabled the equations in the output and dynamics section of the model are not solved during calls to the model. They are instead handled by the integrator as part of the nonlinear system of equations that the integrator solves each step. If the translated model contains several or large nonlinear equation systems, then DAE solvers may be more efficient since fewer nonlinear systems are solved.

When a simulation is run with the DAE flag enabled and one of the above solvers are used a message is printed in the simulation log. The message

```
The translated model is a DAE. Integrating using DAE solver.
```

means that there are nonlinear equations in the output and/or dynamics section of the translated model and that these are handled by the integrator as described above. The message

```
The translated model is an ODE. Integrating using DAE solver.
```

means that there are no nonlinear equations in the output and dynamics section of the translated model. However, the DAE solver formulation is still used. In this case, no performance differences should be expected compared to simulation without the flag enabled.

When trying to run a simulation in DAE mode with a solver that is not supporting DAE mode, the simulation is terminated and an error message is printed.

It is allowed to export a model where the DAE flag is set but the selected solver does not support DAE mode using Binary Model Export or Source Code Generation, but the model will generate error messages when trying to simulate it.

Exporting a model in DAE mode as an FMU is supported if the selected solver supports DAE mode. If this is not the case, the export is not allowed – an error message will be displayed.

Sparse solvers

Dymola supports sparse solvers for the following solvers in Dymola:

- LSODAR
- DASSL
- Cvode
- RadauIIa
- Esdirk23a, 34a, 45a
- Sdirk34hw

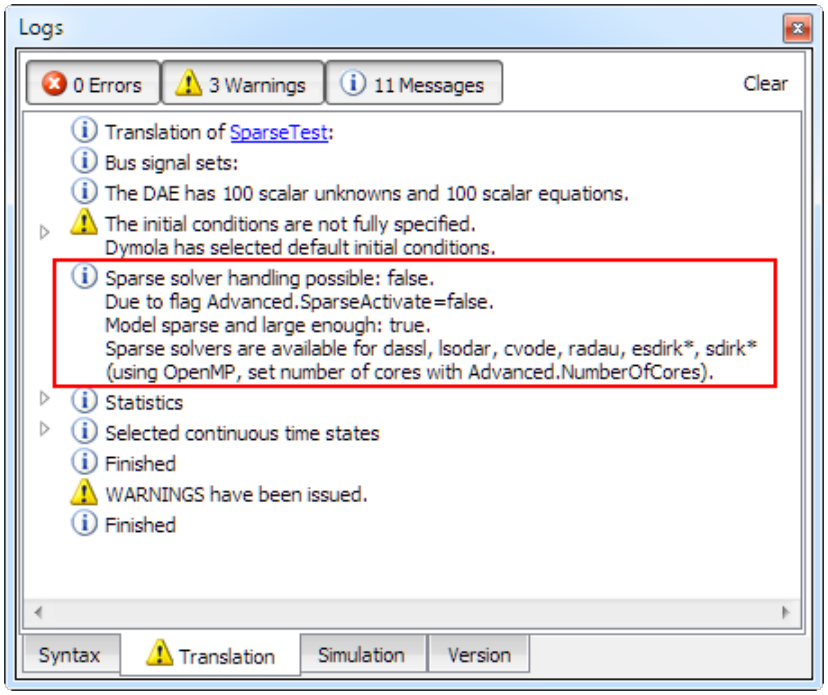
The sparse solver uses a multithreaded (OpenMP) variant of SuperLU.

Implicit integrators, like the ones above, need to solve a nonlinear system of equations during each internal step. This system of equations involves all the continuous-time states of the model and may therefore be large. (The states of a model are listed in the translation log under "Selected continuous time states".) The solution of a nonlinear system is typically performed by a Newton-type algorithm, which rewrites the problem into a series of linear systems of equations to be solved. Each involving all the continuous-time states.

These linear systems can either be solved using dense or sparse linear algebra. Sparse solvers may be faster because they do not consider the zeros of the system matrix. On the other hand, they come with an overhead and therefore most Modelica models are more efficiently simulated using dense linear algebra. Dymola will automatically detect when a model is sparse enough, that is, when the system matrix has a high percentage of zeros. Common examples include models with a space discretization, like high-fidelty descriptions of thermal convection of steam, or models with a spatial distribution where each mode is only connected to a few others, like dynamic models of electric power distribution.

The work effort required to solve a linear system using a dense solver grows as n^3, where n is the number of states. For large models, this work may dominate the CPU time. However, if the model is sparse and a sparse solver is employed, the work effort typically grows linearly with the number of states.

If a model is sparse and large enough, a message is automatically displayed in the translation log when simulating or translating the model. An example:



To activate the use of sparse solvers, set the flag `Advanced.SparseActivate=true;` and select any of the solvers listed above. (The flag is by default set to false.)

The flags defining if a model is suitable or not for sparse solvers are already tuned to good values, we don´t recommend changing them. The flags are (the last one in percentage):

```
Advanced.SparseMinimumStates = 90;
Advanced.SparseMaximumDensity = 5;
```

`Advanced.SparseMaximumDensity` defines the maximum percentage of non-zero elements the system matrix of a model can have to be considered sparse. By default, a maximum of 5% of the elements are allowed to be structurally non-zero.

Both the dense and the sparse linear solvers used by Dymola are direct solvers. That means exact solution of the linear systems, except for round-off errors. Therefore, the choise of dense or sparse normally only has a small effect on the result.

The number of cores used by the multithreaded (Open MP) variant of SuperLU can be set using the flag `Advanced.NumberOfCores`. If the value of the flag is set to `0` (default value) the number of cores is set to 1.

SuperLU parallelization can be used in combination with parallelization of the model code. See section “Multi-core support” starting on page 1006.

5.4.4 Dymosim reference

Dymosim m-files

Together with Dymosim, some Matlab m-files are shipped to ease the usage of Dymosim within Matlab In order to use the m-files, the Matlab path has to be extended by the three directories `Program Files\Dymola 2023\mfiles`, `Program Files\Dymola 2023\mfiles\dymtools` and `Program Files\Dymola 2023\mfiles\traj`. (More information about using Dymola and Matlab is given in the chapter "Simulation Environments".)

The Dymosim m-files (t*) are based on a *trajectory datastructure*:

A trajectory is defined by two matrixes:	

s	A numeric matrix where column 1 is the time vector (=monitoring increasing values) and the other columns are the corresponding signal values. Intermediate values are obtained by linear interpolation. If a trajectory is discontinuous, two successive time instants are identical. Example:
	`s = [0 0 0`
	`1 1 2`
	`2 4 4`
	`3 9 6];`

N	A string matrix, where row "j" of "n" is the name of the signal of column "j" of "s". Example:
	`n = ['time'`
	`'t*t '`
	`'2*t ']`

The newer dymtools m-files (dym*) support an extended trajectory datastructure with reduced memory footprint.

The following m-functions are provided (as usual, you get detailed help for an m-file command within Matlab by command "`help <name>`"). They have been divided into recommended m-files and older (still supported) m-files. (Even more m-files are actually available in the folder `Program Files\Dymola 2023\mfiles` and included sub-folders, but are not recommended for the vast majority of users.)

Recommended m-files

The recommended m-files below should handle most user cases. In some specific case an older command might be used.

Functions on trajectories (data + names):	
dymbrowse	interactively browse, plot and compare Dymola simulation results.
Dymosim	perform time simulation of a Dymola model.

Other functions:	
dymget	Loads trajectory for specific variable into Matlab workspace.
dymload	loads trajectory (e.g. dymosim simulation) into Matlab workspace.
tloadlin	load linear system generated by dymosim into Matlab work space.
tsave	save trajectory on mat-file (e.g. to be used as input signals for dymosim).

Older (still supported) m-files

Functions on trajectories (data + names):	
tcomp	compress trajectory data.
tcut	extract signals and corresponding names from a trajectory.
tder	calculate first derivative of trajectory numerically.
tdiff	determine absolute difference of two trajectories.
tinteg	calculate integral of trajectory numerically.
tplot	plot trajectories and use signal name as legend.
tplotm	plot trajectories in multiple diagrams and use signal names as legend.
trange	find trajectory index range for a given time range.
tsame	make same time axis for 2 trajectory matrices.
tzoom	cut out a trajectory of a given time range.

Functions on trajectory name matrix:	
tnhead	add the same string to all signal names of a trajectory-name matrix.
tnindex	get row-index of string in trajectory-name matrix.
tnlist	list trajectory-names with preceding time range.

Other functions:	
tload	load trajectory (e.g. dymosim simulation) into Matlab workspace. Please consider using dymload or dymget instead for better memory performance.
tfigure	set meaningful default values for figures (e.g. white background and same color on screen and printer).

Dymosim command line arguments

Dymosim is a standalone program which accepts command line arguments with the following syntax:

Usage: *dymosim [options] [file_in [file_out]]*
simulate a DSblock-model with Dymosim.

Options:

-s	simulate (default).
-h	list command summary. If the executable was generated using binary model export or if it will require a runtime license[15] will be displayed (after the command summary), as will any licensed libraries used.
-d file	use input default from "file", i.e., read first all input data from "file" and afterwards read a subset of the input data from the Dymosim input file "`dsin.txt`" or from "[file_in]".
-i	generate Dymosim input file (containing e.g. the names and default / initial values of all parameters and state variables).
-ib	generate Dymosim input file in (Matlab) binary format (mat-file).
-p	precede Dymosim output lines by program name (for real-time Dymosim, in order to identify the process which prints the line).
-v	verify input, i.e. store complete input in file "`dsinver.txt`".
-c cmd	execute command `cmd` after simulation end. This can be useful if Dymosim is started as a separate process in the background.

5.5 Scripting

This section describes the scripting facility in Dymola. The content is divided in a couple of general sub-sections, followed by a number of sections on scripting using function calls. A number of sub-sections describe the scripting using script files (.mos files). These sections "mirror" the sub-sections on functions. Finally the built-in functions are presented in the last sub-section.

5.5.1 Basic facts about scripting in Dymola

When performing simple/single simulations, it is sufficient to select menu commands or to type commands in the command input line of the command window. But wanting to perform more complex actions (e.g. automatically repeat more complicated parameter studies a number of times) it is much more convenient to use the scripting facility. The goal is often to fully automate the simulation.

[15] The demand for a runtime license is now a demand for the Dymola runtime concept, that is, a dymosim executable developed by a user without any export option can still be executed on another computer that has a Dymola license (without any Dymola installation). For more information (on e.g.how to point to the license file) see the index entry "*license : runtime concept*" in the index in the end of this manual.

The script facility makes it possible to e.g. load model libraries, set parameters, set start values, simulate and plot variables.

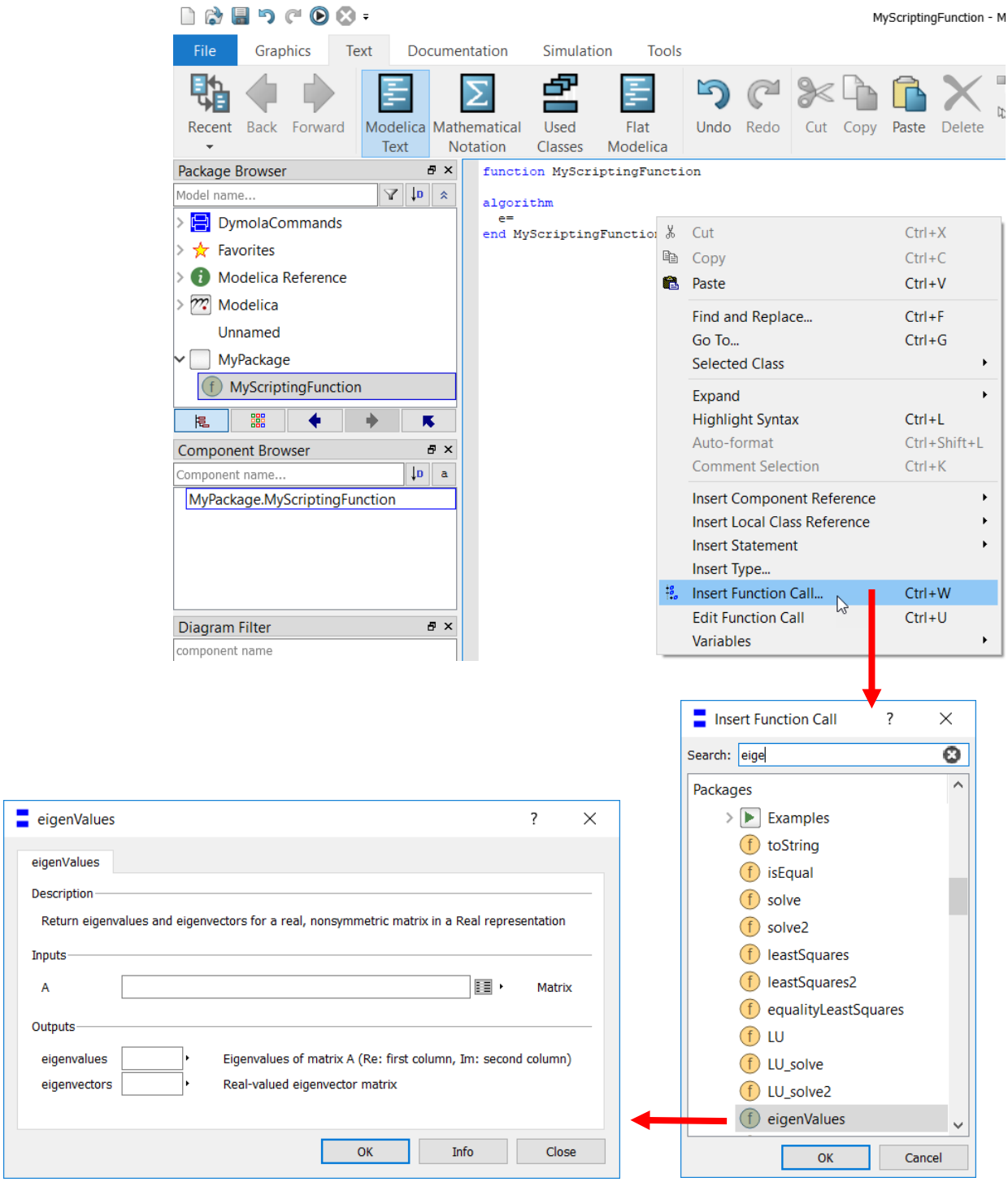
Scripting can be seen as a way of storing a successful (or promising) sequence of interactive events. The user experiments with certain commands and settings and suddenly finds something worth saving for later reuse (and development).

Dymola support easy handling of scripting, both with functions and script files (.mos files):

- Whether a function or a Modelica script file (.mos) should be used is up to the user, essentially the same functionality can be obtained with both.
    - When a function should be the final result, a function is created, and the functionality is then created as an algorithm in this function using the Modelica Text layer of the function as an editor.
    - When a Modelica script file (.mos) should be created, the command input line can be used for input, creating a command log that can be saved as a script.
- The context menu of the command input line and the Modelica Text layer of the edit window contain the entry **Insert Function Call…**. When using this entry, a package browser pops that make it easy to insert a function call, a record constructor or a constant.
- Scripts can be nested; functions can be nested and a Modelica script file may run other Modelica script files.
- Interactive scripting is easy.
    - Commands (e.g. simulation, plotting) can be given by command buttons; each command will be logged in the command log that is available in the command window.
    - An earlier executed command is available using "arrow up" in the command input line.
    - From the command input line the context command **Insert Function Call…** can be given to execute function calls; also these function calls will be available in the command log.
    - Functions in the command input line and the Modelica Text window can easily be changed using the context menu command **Edit Function Call** and executed again.
- When a useful sequence of commands have been created in the command log using the strategy above, the commands in the command log can be copied using the context command **Copy All Commands** in the command log. The sequence of commands can then be inserted in e.g. a function; this is an essential part of script function authoring.

The above might be sufficient for users with some experience of Dymola to start elaborating with the scripting functionality. As a "pre-taster", please look at the following example of inserting the function call `Modelica.Math.Matrices.eigenValues` into the function `MyScriptFunction`:

The below example shows how to use **Insert Function Call...** to insert the function call `Modelica.Math.Matrices.eigenValues` in the function `MyScriptFunction`.


MyScriptingFunction - M
File
Graphics
Text
Documentation
Simulation
Tools
Recent
Back
Forward
Modelica Text
Mathematical Notation
Used Classes
Flat Modelica
Undo
Redo
Cut
Copy
Paste
Delete
Package Browser
Model name...
DymolaCommands
Favorites
Modelica Reference
Modelica
Unnamed
MyPackage
MyScriptingFunction
Component Browser
Component name...
MyPackage.MyScriptingFunction
Diagram Filter
component name
function MyScriptingFunction
algorithm
e=
end MyScriptingFunctio
Cut Ctrl+X
Copy Ctrl+C
Paste Ctrl+V
Find and Replace... Ctrl+F
Go To... Ctrl+G
Selected Class
Expand
Highlight Syntax Ctrl+L
Auto-format Ctrl+Shift+L
Comment Selection Ctrl+K
Insert Component Reference
Insert Local Class Reference
Insert Statement
Insert Type...
Insert Function Call... Ctrl+W
Edit Function Call Ctrl+U
Variables
Insert Function Call
Search: eige
Packages
Examples
toString
isEqual
solve
solve2
leastSquares
leastSquares2
equalityLeastSquares
LU
LU_solve
LU_solve2
eigenValues
OK
Cancel
eigenValues
eigenValues
Description
Return eigenvalues and eigenvectors for a real, nonsymmetric matrix in a Real representation
Inputs
A
Matrix
Outputs
eigenvalues
Eigenvalues of matrix A (Re: first column, Im: second column)
eigenvectors
Real-valued eigenvector matrix
OK
Info
Close

The context menu is popped by right-clicking and selecting **Insert Function Call...** (even simpler is to use **Ctrl+W**). Typing `eige` in the **Search** input field will be enough for Dymola to show the function `eigenValues` as default selection. Clicking **OK** will display the function call menu for that function.

Note that the function call dialog contains entries to fill in output variables, i.e. in what variables of the scripting environment to store the results. If none of the output variable entries are filled in, the results are output in the Command window.

Please also note that selected commands (built-in functions) are automatically accessed from DymolaCommands. This library is by default opened when opening Dymola. If not opened, use the command **File > Libraries > DymolaCommands**. For more information about built-in functions, please section "Built-in functions in Dymola" starting on page 937.

5.5.2 Scripting in Dymola – Functions/Script files

Scripting in Dymola must be seen as the broad definition of scripting, rather than the traditional view of packing a number of commands into a "batch" file – although the latter is possible as well. Scripting can be seen as a way of storing a successful (or promising) sequence of interactive events. The user experiments with certain commands and settings and suddenly finds something worth saving for later reuse (and development).

The main selection to be done when using scripting in Dymola is whether the final result should be a Modelica function or (more traditionally) a Modelica script file (.mos).

The script as a function

If the functionality should be a function, such a function must be created – see later. In the Modelica text layer of this function, statements can be built up to create the functionality (using e.g. other Modelica function calls).

The function is usually saved in a suitable package.

The function can be executed by right-clicking in the command input line and use the context command **Insert Function Call...**. The parameter dialog will pop automatically, parameter values can be entered and the function executed.

A function call can be included as a user command (reachable using **Commands**) in a model if wanted; see section "Saving a function call as a command in the model" on page 914.

The script as a Modelica script file (a .mos file)

If the functionality should be a Modelica script file (.mos), such a file must be created. This can be done in several ways – please see later. It is common to use the command input line to create a command log file that can be saved as a .mos file using a certain command in Dymola.

Inside a Modelica script file (.mos) a Modelica function can be used by calling it.

An already existing script file can easily be edited using the Dymola script editor. (It can also be edited by any common text editor, e.g. MS Notepad, but with fewer facilities).



The script can be executed in several ways, e.g. using a command button in Dymola or using the function call `RunScript("scriptname.mos")` in the command input line. Note that it

also can be executed from the script editor. Also a selected part of the script can be executed from the script editor, as were it a free-standing script.

A call of the script file can be included as a user command (reachable using **Commands**) in a model if wanted; see section "Saving a script file as a command in the model" on page 921.

5.5.3 The possible content in a Modelica function or script file

Basic operations

It is possible to set and get model parameters, initial values and translator flags. Computations using all Modelica functions and array operations are allowed. It is possible to call translated functions, thus gaining access to external functions. Interactive variables are automatically created. Set and get of both arrays as a whole and of individual elements are supported. Named arguments are allowed to functions like `simulateModel` with default values. Modelica expressions can be stored in script files or as Modelica functions.

Dymola contains a large number of built-in functions that can be used. For convenience they are presented in a section of their own, please see section "Built-in functions in Dymola" starting on page 937.

The Modelica based scripting language is extensible since functions written in Modelica can be called in the same way as the predefined (built-in) functions. Thus the functions in the Modelica standard library can be used directly from the scripting language, allowing e.g. table interpolation.

Please note that variables, functions etc are case sensitive.

Input from the command input line and output from the system can be rendered with mathematical notation (by deafault this is not the case). See section "Rendering as text or not for inputs and outputs" on page 772. However, since this cannot be saved in scripting, below the default of not performing such rendering is kept; see the reference for how to activate the mathematical notation.

Interaction

By typing a Modelica expression in the command input line, the result is output (the input is written in bold face):

```
Modelica.Math.sin(5)
  = (-0.958924274663138)

Modelica.Math.sin(0:0.5:1.5)
  = {0, 0.479425538604203, 0.841470984807897,
  0.997494986604054}

{{1,2},{3,4}}*{{1,2},{3,4}}
  =
[7, 10;
15, 22]
```

i.e. the tool “completes” the equation with an equal sign and the result. The semicolon at the end may be omitted.

Import statement can be used in the command input line.

```
import Modelica.Math.*
 = true
sin(1)
 = 0.841470984807897
```

It is possible to continue a statement on several input lines

```
simulateModel("Modelica.Blocks.Examples.PID_Controller",
Continue line:
stopTime=4, method="dassl", resultFile="PID_Controller")
  = true
```

by ending a line before a complete assignment or expression has been entered. However, breaking an input (e.g. a string) in e.g. a function call is not always allowed. This rule allows omitting the semicolon at the end of a statement.

Several expressions are also allowed if separated by semicolon:

```
2*3; 4*5
  = 6
  = 20
```

Diagnostics is given immediately when errors are encountered:

```
transpose({{1,2},{3,4,5}})
Error: The parts of
{{1, 2}, {3, 4, 5}}
are not of equal sizes:
 {2}{3}
```

Assignments

It is possible to use a Modelica assignment, :=. For convenience, the = operator is used below to denote assignment. Comments can be written as usual using // or /* */.

```
model.p1 = 5     // update model.p1
model[3].q[model.p1] = Modelica.Math.sin(6)
```

Interactive variables

The result might be stored in a variable. The variable is declared with the type and size of the expression on the right hand side of the assignment. Typing the expression consisting of just the variable name, outputs the value.

```
i = zeros(3)
Declaring variable: Integer i [3];

i
 = {0, 0, 0}

r = 1:0.5:2
Declaring variable: Real r [3];
```

```
r
 = {1.0, 1.5, 2.0}

b = {false, Modelica.Math.sin(5)<0}
Declaring variable: Boolean b [2];

b
 = {false, true}

s = {"Modelica","script"}
Declaring variable: String s [2];

s
 = {"Modelica", "script"}
```

Such variables can be used in calculations.

```
r*r      // scalar product
  = 7.25

r[2] = 1.5
s[2]
  = "script"

t = s[1] + " " + s[2] + "ing: r[2] = " + String(r[2])
Declaring variable: String t ;

t
  = "Modelica scripting: r[2] = 7.5"
```

A list of the interactive variables and their correct values can be obtained by the command variables():

```
variables()
List of variables:
Integer i[3] = {0, 0, 0};
Real r[3] = {1, 1.5, 2};
Boolean b[2] = {false, true};
String s[2] = {"Modelica", "script"};
String t = "Modelica scripting: r[2] = 1.5";
```

Re-declaration of size for variables in work space is allowed.

```
a=1:10
Declaring variable: Integer a [10];
a=1:20
Redeclaring variable: Integer a [20];
```

Predefined variables

We might want to introduce several variable categories like system variables, interactive variables, model variables, etc. System variables might involve pragmas to the translator about preferred handling.

```
Evaluate = true
// utilize parameter values in manipulation
```

These settings are also available in the **Simulation > Setup** menu, the **Translation** tab, where e.g. Evaluate corresponds to selecting **Evaluate parameters to reduce models (improves simulation speed)**. See also the section "Translation tab" on page 805.

Functions for string manipulation

The string manipulation routines are constructed for formatting numerical results as strings. The routine used is `String` for converting numbers to strings and the Modelica built-in operator + that concatenate strings.

These functions can be used for e.g. presenting results

```
r:=12.3;
"Modelica scripting: r = "+realString(number=r);
  ="Modelica scripting: r = 12.3"
```

and for accessing a sequence of files:

```
sumA=0;
for i in 1:10 loop

A=readMatrix("a"+String(i,significantDigits=2)+ ".txt,"A",2,2);
  sumA:=sumA+sum(A);
end for;
```

The last example assumes that a number of files a01.txt, a02.txt etc. with relevant data are available in the working directory.

Deleting variables in workspace

The function `deleteVariable` can be used to delete variables in workspace.

```
function deleteVariables
  input String variables[:];
  output Integer nrDeleted:
end deleteVariables
```

Example:

```
deleteVariables({"A", "B"});
```

5.5.4 What is a Modelica function?

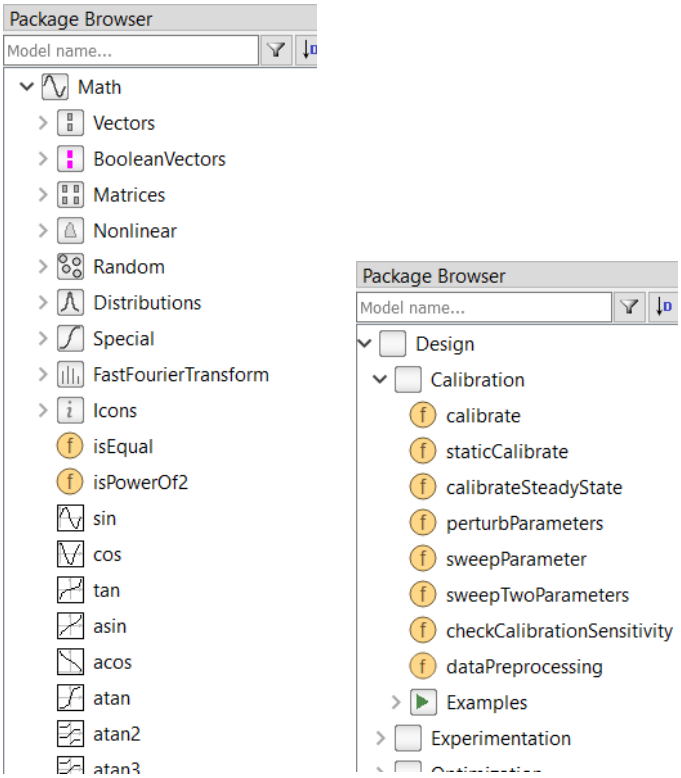
A function is principally a "black box", acting on values given to it and returning outputs without revealing intermediate results. This also means that functions do not use the general

Dymola workspace for any calculations; each function has a workspace of its own. Please compare this with a Modelica script file (.mos) that works totally different (see later section).

A Modelica function can also be used as a mean to model a part of a system in a procedural programming style. The Modelica function construct is described in detail in Modelica Language Specification. Please see this document.

A Modelica function will be present in the package browser of Dymola, sometimes symbolized with an “f”, sometimes with a more specific symbol.

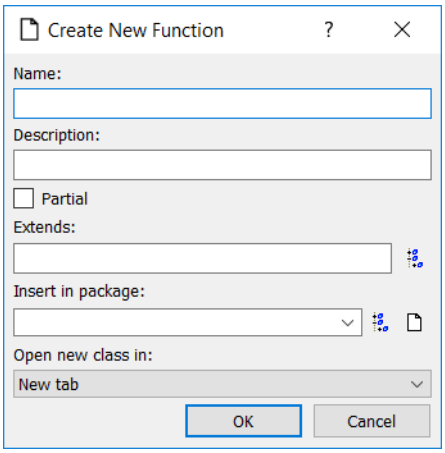
Examples of functions from Modelica.Math (in lower part of figure) and Design.Calibration.



5.5.5 Creating a Modelica function

A Modelica function is created when developing a model, using the command **File > New > Function**. A dialog pops up:

Creating a new function.



The function name has to be specified, and the last entry can be used to specify in what package the function should reside.

(For more information about the menu **File > New** generally, please see that command in previous chapter, section “Editor command reference when developing a model”.)



If no graphics have been included in the function, the “f” symbol will symbolize it in the package browser. If another symbol should be used, insert or create that symbol in the icon layer of the function.

For creation of nested functions, please see next section.

It is advisable to create the functions in packages, avoiding a large number of top-level functions. The package(s) path(s) could be stored in the MODELICAPATH; then it is possible to call the functions from the command input line or inside models without having to load files first. As an example calling a function `ReadA()` in the package `myFunctions` can then be done with `myFunctions.ReadA()`.

5.5.6 Saving a function call as a command in the model

It is possible to save a function call to an existing function as a command in the model using the command **Simulation > Commands > Add Command...**. See section “Simulation > Commands > Add Command” starting on page 793 for more information.

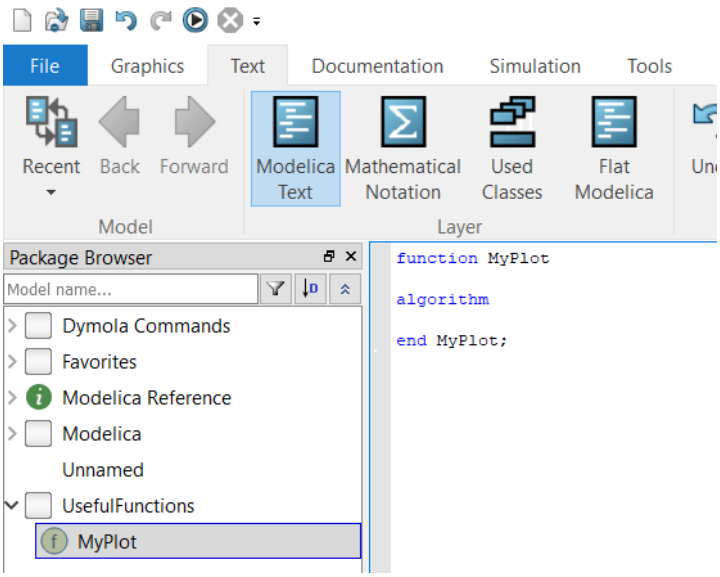
If you for some reason change the location of a model containing commands corresponding to functions called, please check that the commands still work after the change of location.

5.5.7 Editing a Modelica function

Editor

When the function has been created, the Modelica Text layer of the function is used for editing it. As an example, if a function myPlot has been created in the package UsefulFunctions, the following appear when having defined the function:

Example of newly created (empty) function.



When it comes to defining parameters, writing expressions etc please see relevant sections in previous chapter “Developing a model”. Only the handling of functions will be treated in this chapter.

Please note the possibility to copy the result of a successful interactive scripting session from the command log by using the context command **Copy All Commands** in the command log and pasting the result into a function. See section “Basic facts about scripting in Dymola” starting on page 904.

Handling functions

Inserting a function call

A function can be inserted in the algorithm taking up the context menu and using **Insert Function Call...**. A package browser will pop. This makes it easy to insert any function, record or constant. Please see an example of this in the beginning of the scripting section.

Please note that selected commands (built-in functions) are automatically accessed from DymolaCommands. This library is by default opened when opening Dymola. If not opened, use the command **File > Libraries > DymolaCommands**. For more information about built-in functions, please section “Built-in functions in Dymola” starting on page 937.

Handling of nested function calls

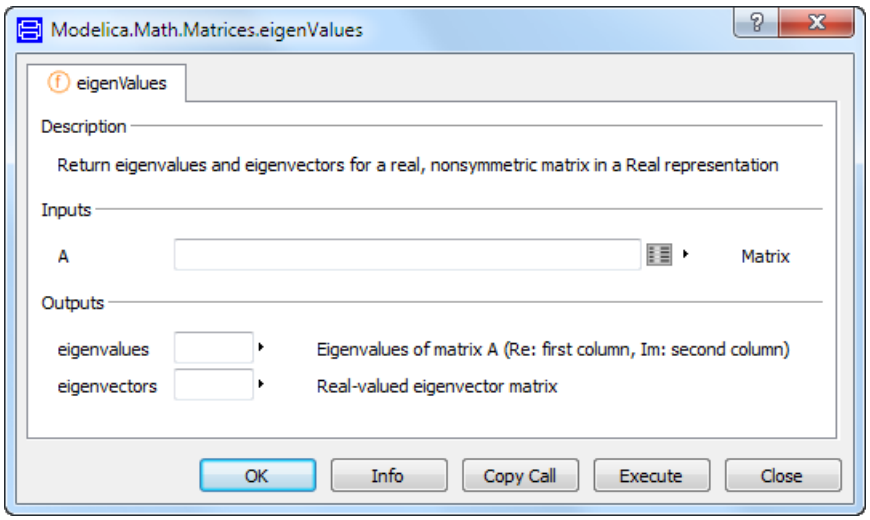
Nested function calls can be created. The main advantage with nested functions is the extensive information exchange between the primary ("calling") function and the nested ("called") function. The nested function can use and alter its input values, and can also create new ones. After the call, the primary function can continue to work with the result from the nested function.

By inserting a function (above), the result is a nested function. If nesting in several levels should be used, the new (nested) function can be inserted by putting the cursor on the primary function and taking up the context menu and using **Insert Function Call...** to insert the nested function.

5.5.8 Executing a Modelica function call

A function call can be executed from the package browser, by right-clicking the function and selecting **Call Function...**. This will pop the parameter dialog of the function call:

Example of a parameter dialog of a function call.



When clicking **OK** or typing Return the function call is executed and the window is closed. **Execute** will execute the function call without closing the window. (**Copy Call** can be used to copy the function call to clipboard for further use.)

5.5.9 Advanced use of functions

Usage of compiled functions

Expressions using compiled functions are not evaluated unless needed, for example, if they contain any parameter with the annotation `Evaluate=true`.

For compability reasons, if models have been constructed to rely on all expressions using compiled functions being evaluated, previous behavior can be preserved by setting the flag

```
Advanced.RelyOnCompiledFunctions=true
```

The flag is `false` by default.

In Dymola 2019 and earlier, compiled functions were never recompiled. From Dymola 2019 FD01 a flag is available for controlling when used functions are recompiled:

```
Advanced.WhenRecompileFunctions
```

The present possible values are:

- 0 Never recompile used functions (corresponding to Dymola 2019 and earlier)
- 2 Recompile editable used functions (e.g. functions not in Modelica Standard Library or in any encrypted library). This is the default value.
- 3 Always recompile used functions

Notes:

- Setting the flag to 0 gives faster execution but runs the risk of using old compiled functions.
- The flag value 1 is for future use and is yet not implemented.

Constructing pre-compiled functions

It is possible to construct pre-compiled functions by selecting **Translate** in the menu of the function. This constructs an executable, and it is possible to call exactly as the non-compiled function. When calling functions from the command input line pre-compilation automatically occurs for external functions.

Note: The executable must be in the path for the command to be interactively callable. It is not checked that the executable is up to date.

It is possible to trace function execution using the built-in function

```
trace(variables = false,statements = false,calls = false,
onlyFunction = "");
```

This will optionally trace and log values of assigned variables, executed statements, calls of the function, and can optionally be overridden for a specific function. Calling trace with default value for `onlyFunction` overrides the trace-settings for all functions.

5.5.10 Pre-defined Modelica functions

Numerous functions are available in Dymola – most libraries contain a number of functions.

5.5.11 What is a Modelica script file?

A script file can be said to be a convenient way to "pack" a number of actions (commands or function calls) to simplify the use of them. Please note the difference between a script file and a function, the function is principally an independent "black box", acting on values given to it and returning outputs without revealing intermediate results.

The script facility makes it possible to e.g. load model libraries, set parameters, set start values, simulate and plot variables by executing script files. This is how the demo examples are setup.

Scripts can be nested, i.e. a script file may run other script files.

Numerous built-in (predefined) functions are available in scripts. Functions written in Modelica can be called in the same way as built-in functions, thus the functions in the Modelica standard library (and other libraries) can be used directly from the scripting language. (However, free-standing equations and algorithms in the model are not supported.)

Modelica scripts files can be automatically constructed using the command **Script Editor > Generate Script > Command log**.

Since the script files are readable text-files (`.mos` files, from Modelica scripts) they can easily be edited, parameter values changed, and a simple interactive session can be changed to a parameter-sweep by adding an enclosing for-loop.

Restrictions

The main restriction is that only operations allowed in algorithms are allowed in script files.

5.5.12 Creating a Modelica script file

There are two ways to create script files:

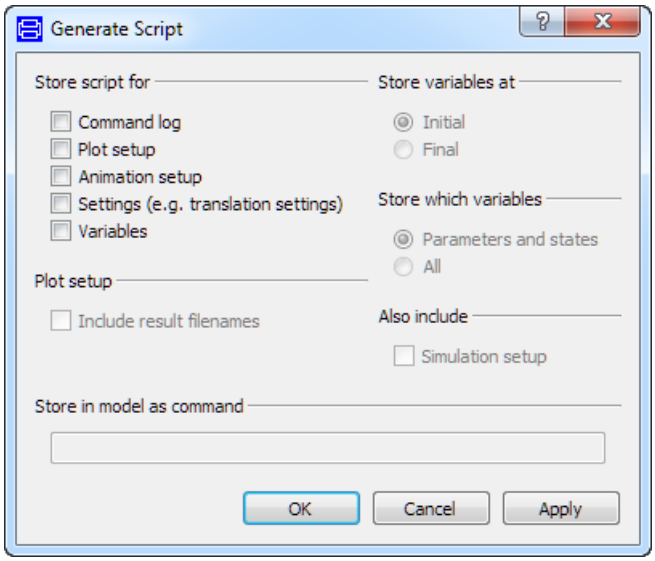
- To create them from scratch using the Dymola Script Editor. See section "Creating, editing and saving scripts using the Dymola script editor" starting on page 927
- To create them automatically using the command **Script Editor > Generate Script** in Dymola.

Please note that the context command **Copy All Commands** in the command log makes it very easy to obtain the same result using the first alternative as the second one. Please see next section for details.

Using the command Script Editor > Generate Script, to create a script file

When creating a script file using Dymola, the command **Script Editor > Generate Script** is a good start. When selected, the following window will be shown:

Generate script file alternatives.



The first alternative below offers a general way of creating a script file, while the other alternatives create a pre-defined script file according to the alternative selected.

Note that when creating a script file for the Command log, Plot setup or Animation setup, commands (e.g. plot commands) are not shown in the log until another command is performed. The reason is that it would not be convenient to show each command if a number of changes in the plot window is performed.

When creating a script file using the command **Script Editor > Generate Script** above, Dymola suggests saving the script file in the working directory of the model in which the script file has been created. There are two reasons for this.

The main reason is that the script file often includes references to other files (as an example a plot script file refer to the file `modelname.mat` that is created in the working directory when simulating the model). Such references assume that the script file is stored in the same folder, since the full path for files are not automatically included in the script file.

Another reason is that if the user chooses to run the script file by any of the commands **Simulation > Run Script** or **Script Editor > Run Script**, the working directory of Dymola will by default be changed to where the script file is located before executing it, which makes it a good idea to have the script saved where the corresponding model is located. The default setting of running the script where it is located can be changed by the toggle command **Change to Script Directory**, for those commands (However, using the command **Commands > *commandname*** (if the script file is saved as a command) or using the Modelica function `RunScript("path/scriptname.mos")` will *not* change the file path, independent of the status of the **Change to Script Directory** command.)

(To check the current path to the working directory of Dymola, type `cd` (followed by return) in the command input line.)

Saving the Command log as a script file

Scripting can be seen as a way of storing a successful (or promising) sequence of interactive events. The user experiments with certain commands and settings and suddenly finds something worth saving for later reuse (and development).

The selection of the **Command log** alternative will use the command log as the tool for creating such a script file. By selecting this alternative, the present command log will be saved in a script file. Please note that neither outputs from Dymola (results etc.) nor commands that have no equivalent Modelica function will be included in the script file.

The workflow of creating such a script file can be:

1. Display a command window, see section "Displaying the command window" on page 764.
2. Create the content of the script. A simple script file can be created by interactively performing the operations that should be included in the script file (e.g. open a certain model, initializing it in a certain way, simulating it and presenting the results). Menu commands or commands entered in the Command input line can be used, as well as more elaborated actions e.g. inserting nested functions. Please see below, section "Editing a Modelica script file" starting on page 922 for details. (If no operations are performed before saving the file, you will create an empty script file.)
3. Use the command **Script Editor > Generate Script**, tick **Command log**. (If you also want to have the script file saved as a command in the model, you can enter the name of such a command in the input field **Store in model as command**, as described earlier.) Click **OK**. Now you can enter the wanted name of the script file and the location – please see the note below on script file location - Click **OK** when those selections have been made.
4. Test the script file.
5. The content in the command log can now be cleared using the command **File > Clear Log.**

Saving the plot setup as a script file

The total plot setup, that is, all displayed plot windows including displayed curves and settings, can be stored as a script file using the command **Script Editor > Generate Script**, ticking **Plot setup**.

Notes:

- Presently zooming cannot be captured in the script file; the curves are always shown as initially presented ("rescaled").
- It is possible to store the plot setup in the model without a script file. See section "Simulation > Commands > Add Command" starting on page 793 for more information.

Saving the animation setup as a script file

The total animation setup, that is, all displayed animation windows with displayed objects and corresponding settings (including zooming and other view manipulations), can be stored as a script file using the command **Script Editor > Generate Script**, ticking **Animation setup**. Note that it is possible to store the animation setup in the model without a script file. See section "Simulation > Commands > Add Command" starting on page 793 for more information.

Saving settings as a script file

Ticking the alternative **Settings (e.g. translation settings)** will enable the saving of all non-standard setting of flags to a script file.

Saving variables as a script file

Ticking the alternative **Variables** will in turn enable a number of alternatives in the menu.

A selection **Store variables at** has to be done; a selection whether the saved variable values should be the values before the simulation (**Initial**) or after the simulation (**Final**). A common case is to use Final in order to be able to save values after a simulation that will be used as a starting point of a later simulation.

A selection **Store which variables** (values of) has to be done.

- **Parameters, and states** will save the values of parameters and variables where initial values are required
- **All** will save all available values of variables.

Finally, you can select, in the **Also include** group, **Simulation setup**. By ticking this alternative, the start time, stop time and integrator is also saved. Note that there are, for advanced users, some additional flags that are related to the simulation setup, see section "Advanced setup for simulations by flags" on page 782.

It is possible to use scripts instead of these commands; e.g. wanting to save all final variables in the script file MyScript can be obtained using

```
exportInitial("dsfinal.txt","MyScript.mos",true,true)
```

Please compare with functions in section "Simulator API" starting on page 939.

5.5.13 Saving a script file as a command in the model

It is possible to save an existing script file as a command in the model using the command **Simulation > Commands > Add Command**. See section "Simulation > Commands > Add Command" starting on page 793 for more information.

When creating a script file using the command **Script Editor > Generate Script** it is possible to define it as a command when creating the script file (see previous section).

If you for some reason change the location of a model containing commands corresponding to script files, please check that the commands still work after the change of location.

5.5.14 Editing a Modelica script file

The first section below describes how the command input line can be used to edit a script the first time it is created.

The second section describes the general way of editing a script file using Dymola script editor. (Any common text editor, e.g. MS Nodepad, can actually be used, but with less features.)

Using the command input line in Dymola

Workflow

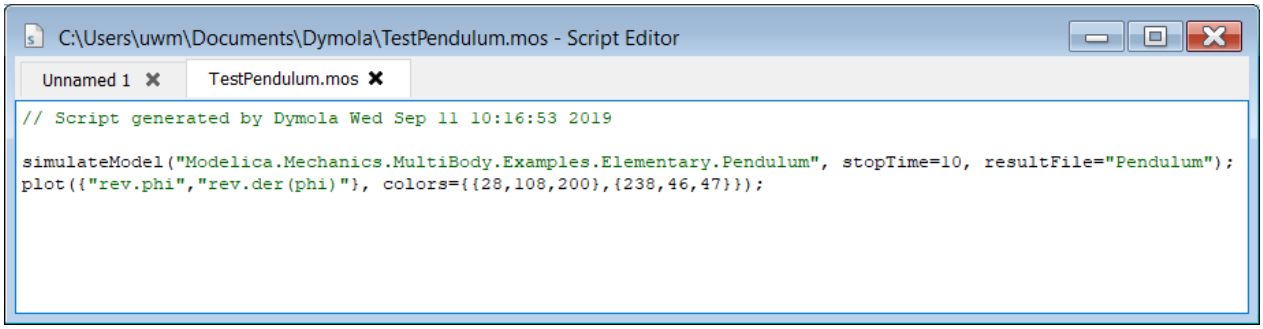
Please be aware that this possibility only exists when creating the script file for the first time, there is no way to open a script file into the command window of Dymola (except running the script file). This means that the alternatives here is only of interest while creating a script file using the command log (please see previous section).

All actions that result in lines in the command log will be stored in the script file. Please note that not all operations will create code, which means that such an operation will not be included in the script file. Please also note that each command might not show up as a single line in the command log, e.g. a number of settings of simulation will be included in a `simulateModel` function call line. In other words, what counts in the end of the day is if the action can be seen in the script file!

A simple script file can be created by interactively performing the operations that should be included in the script file. An example of creation of the content of such a script file (we include the items that has to do with the creation of the script file in [brackets] for completeness in this basic case):

1. [Display a command window. By default, you have it in the Simulation tab.]
2. [Use the command **File > Clear log** to clear the content of the command log.]
3. Go to the **Graphics** tab, and from the package browser, open the model (e.g. Modelica.Mechanics.MultiBody.Examples.Elementary.Pendulum).
4. Go to the **Simulation** tab, and change the simulation stop time (to e.g. 10 s).
5. Simulate the model.
6. Show certain signals in a plot window (e.g. rev.phi and rev.der(phi)).
7. Click on the command input line of the command window to finalize the plotting command – the command will now be displayed in the command window.
8. Use the command **Simulation > New Script > Generate Script…**, tick **Command log**. Click **OK**. Now you can enter the wanted name of the script file (e.g. TestPendulum) and the location (in this case, do not change the location). Click **Save** when those selections have been made.
9. The created script is opened in the **Script Editor**, to acknowledge that the creation was successful. You can now close the script if you want to, or edit it further. Don´t forget to test the script file.

The resulting script file might look like (If you happended to close it, you can open it by the command **Simulation > Open Script…**):



The first line is a comment automatically generated by Dymola. The four items of action above are compressed in this script file is expressed into two single lines.

More elaborate actions can also be entered in the script file; the handling of functions is an important area:

Handling of simple functions

A "simple" function can be inserted in the command input line (or directly in the command log) in three ways:

Typing

Any function can of course be entered by typing it in the command input line (followed by enter to input it in the command log – and to execute the function). If the command log already contains a similar function, arrow up can be used to display commands previously entered in the command log. Copy/paste can also be used.

Parameter values can also be typed in. If the function can be visible in the package browser an alternative can be used. By right-clicking and selecting **Edit Function Call** the parameter dialog of the function will be available; here the parameter values can be changed.

Import-statements can be used to simplify long function paths, please see below.

Using Insert Function Call… in the context menu

If the function can be made visible in the package browser, is can also be inserted in the command input line using the context command **Insert Function Call….** Using this command will pop a package browser where function calls, record constructors or constants can be inserted.

Please note that selected commands (built-in functions) are automatically accessed from DymolaCommands. This library is by default opened when opening Dymola. If not opened, use the command **File > Librareis > DymolaCommands**. For more information about built-in functions, please section "Built-in functions in Dymola" starting on page 937.

When it comes to nesting of function calls, please see earlier corresponding section dealing with function calls.

Using import-statements

Import-statements can be used to simplify the handling of functions in script files by enabling using shorter function paths when using them.

Please note that import-statements can be used the same way to simplify paths when programming in Modelica.

Two types of imports can be used, qualified import and unqualified import.

Qualified import

A common way using qualified import is to use it to "remove" the path preceding the package name when using a function.

Example

```
import Modelica.Utilities.System;
System.getWorkDirectory();
```

can be used to replace:

```
Modelica.Utilities.System.getWorkDirectory();
```

The advantages are shorter paths (the import statement is only given once) and increased readability.

Please note that the import-statement must also be copied if "copy/paste" are used on functions affected by the import-statement, otherwise the paths are "lost" when pasting the code to another script file!

Qualified import can also be used to replace the package name with a user-defined name, e.g. (compare with above):

Example

```
import Sys=Modelica.Utilities.System;
Sys.getWorkDirectory();
```

Some caution is needed here, the advantage is that a general package name can be more adapted to the script file (usage), enabling higher understanding. On the other hand, the intuitive connection to a general package is lost, which might lead to misunderstandings.

For more details about qualified import, please see Modelica Language Specification, Version 3.0 or later.

Unqualified import

Unqualified import can be used to remove the path of functions altogether.

Example (compare with example above):

Example

```
import Modelica.Utilities.System.*;
getWorkDirectory();
```

can be used to replace:

```
Modelica.Utilities.System.getWorkDirectory();
```

In most cased, the recommendation would be not to use this feature. The reason is that any connection to the original package is lost for the reader, and if more than one package is imported this way, misunderstandings will almost inevitably occur. From what package does a certain function come from? What if two imported packages have functions with the same name? What if a function name is erroneously typed – where to search information?

Using the Dymola script editor

The general way of editing a Modelica script file is by using the Dymola script editor. For more information about the script editor, please see section "The Dymola script editor" starting on page 926

An important feature that helps extracting information from the command log to the script editor is the context command **Copy All Commands** in the command log. This command makes it possible to copy an interactive simulation session and paste it into the script file for further work. Please see also section "Basic facts about scripting in Dymola" starting on page 904.

Please note that **Copy Call** can be used when functions should be inserted. If the function can be made visible in the package browser, right-clicking on it will display a function dialog where the **Copy Call** command button is available. This button will copy the function call (including the parameter list) to the clipboard. Then it can be pasted into the command input line (followed by enter to input it into the command log) – or into the command log directly. In the latter case, the function will not be executed.

Please note the convenience of entering the relevant parameter values using the function dialog before doing **Copy Call**.

To simplify the function paths, import-statements can be used (please see section above), but no automatic shortening of names are available in an editor like this.

(A Modelica script file can be actually be edited using any simple text editor, e.g. MS Notepad. Please note that no interactive input or any syntax checking is available in such an editor.)

5.5.15 Running a Modelica script file

Please note that the resulting variable values of running a Modelica script file are accessible also after the script file has been executed. Please note however that these values can be overwritten by other script files and commands.

If an error is encountered in a command script file, all script files are closed and interactive command input is resumed.

There are five ways to run a script file:

1. Using the command **Simulation > Run Script**  . It is possible to browse for the script file, but please note that the working directory will be changed to location of the script file. (Once in the script editor, you can change this default setting with the toggle command **Simulation > Change to Directory**  .)

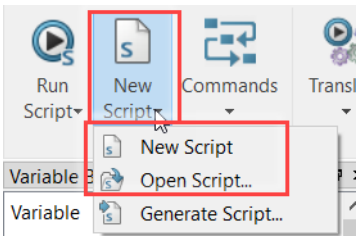
2. If the script file has been associated with a command ("*commandname*") that command can be used to run the script file using **Simulation > Commands > *commandname***. (A script file can be associated with a command either by using **Store in model as command** in the command **Simulation > New Script > Generate Script...** when the script file was created or by using the command **Simulation > Commands > Add Command...** to associate the script file with a command afterwards.) The path of the working directory of Dymola is *not* changed when using such a command.
3. The third alternative is that the script file can be run using the Modelica function `RunScript("path/scriptname.mos")`. This alternative is the most interesting one in the following, partly since this enables automating the simulation, partly since this enables nesting of script files. We will come back to this. If the script file is located in the working directory of Dymola no path should be entered (just `RunScript("scriptname.mos")`). The path of the working directory of Dymola is *not* changed when using such a command.
4. The script (or part of it) can be executed from the Dymola script editor, if the script is open and active. For details, see "Executing a script or a selected part of a script from the Dymola script editor" on page 932.
5. Using the command input line by typing `@path/scriptname.mos` followed by return. If the script file is located in the working directory of Dymola no path should be entered (just `@scriptname.mos`). Please note that if a path is typed, the working directory of Dymola will be changed accordingly.
6. A Dymola script file can be given as a command line argument when starting Dymola from the Command Prompt in MS Windows (reached using **Start > All Programs > Accessories > Command Prompt** in Windows). For an example, please see the parameter studies example below.

5.5.16 The Dymola script editor

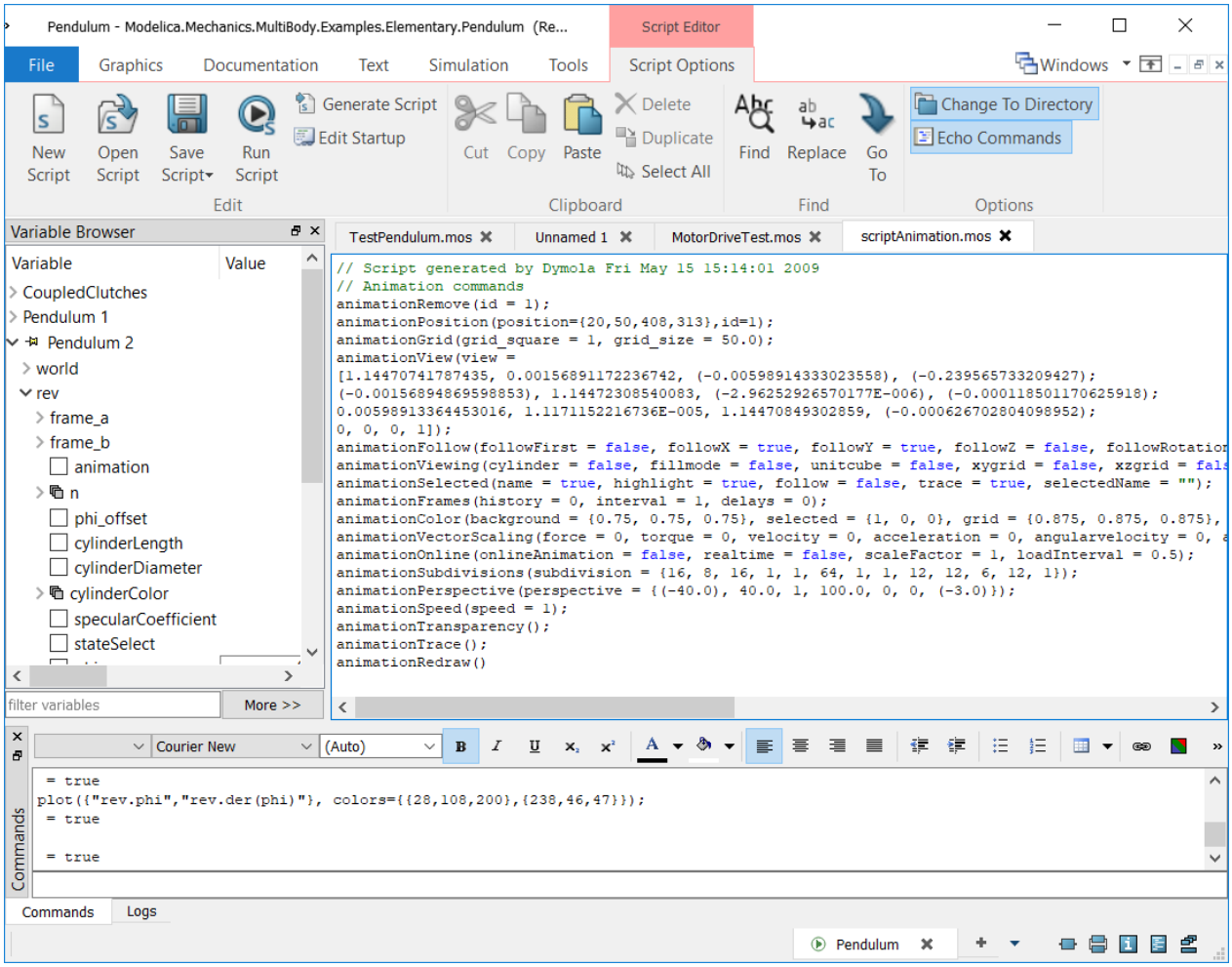
Accessing the Dymola script editor

The following commands in the **Simulation** tab will open the Dymola script editor.

- **Simulation > New Script > New Script**
- **Simulation > New Script > Open Script...**



An example of the Dymola script editor:



Note the tab system in the script editor.

The script editor is by default displayed as a subwindow, maximized, but can also be displayed maximized or minimized. In the example above, the script editor has been maximized, this can be changed by the common buttons almost top right of the main window in the figure above.

Creating, editing and saving scripts using the Dymola script editor

Handling of several scripts in the script editor

The tab system in the script editor (see figure above) enables several scripts being open in the editor, however only one can be displayed and active at each time. Click the wanted tab to select that script as displayed and active.

A tab can be closed by clicking the cross on it. If the corresponding script is changed (indicated by a star) a warning will be displayed with option to save the changed script before closing the tab.

Creating a new empty script

There are three ways to create a new empty script in the script editor:

- By the command **File > New > New Script**. This command will put open the **Script Editor** tab, with a new script in the script editor.
- In the **Simulation** tab, using the command **New Script > New Script**.  . (You can also click directly on the button **New Script**.)
- If the **Script Editor** tab is already open, you can use the command **New Script**  .

In any case a new tab is created with the name `Unnamed` or `Unnamed x`, where x is a number 2 or larger, depending on how many "Unnamed" tabs already present in the script editor.

Note. For the command **Script Editor > Edit Startup** see "Creating and editing a customized setup file in `.mos` format" below.

Opening an existing script

There are two ways to open an existing script:

- In the **Simulation** tab, using the command **New Script > Open Script…**  .
- If the **Script Editor** tab is already open, you can use the command **Open Script**  .

A new tab is created in the script editor containing the script; the name of the script is displayed in the tab.

Note. For the command **Script Editor > Edit Startup** see next section.

Creating and editing a customized setup file startup.mos

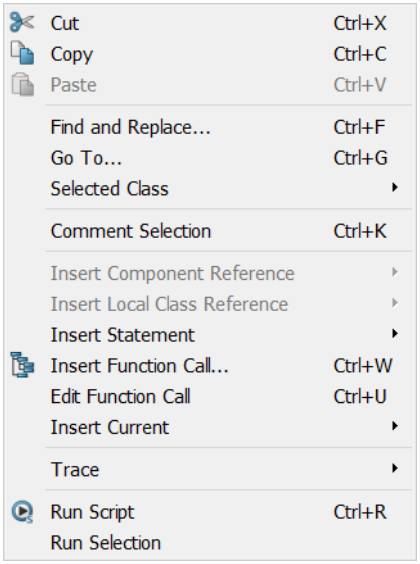
You can use the script editor to create and edit a customized setup file `startup.mos`. For information about the handling and use of such a file, see the index entry "*setup file: customized setup file*" in this manual.

To create of edit this file, you can use the command **Script Editor > Edit Startup**  .

Editing a script

Any editing of a script not being saved is indicated by a star in the tab.

Apart from ordinary text editing (including code completion, see below), a context menu provides a number of options for editing:



The options (except **Insert Current** alternatives, **Run Selection**, and **Trace**) are actually a selection of the corresponding commands in the Modelica Text editor, and work the same way. See previous the description of this editor in previous chapter for details.

For description of the context menu, see section “Context menu: Script editor” on page 885.

Code completion can be activated by pressing **Ctrl+Space** in the editor. This will bring up a context menu containing all available contextual words beginning with the letters written so far, or all contextual words available if nothing has been written yet. As you type, the list of possible choises will decrease until only one word remains. If something not matching the words in the completion list is typed, the menu will automatically close.

The code completion is contextual, that means that the menu contains:

- All Modelica keywords (highlighted in blue).
- Names of open packages.
- Names of models available at the code completion level.
- Names of built-in functions available in DymolaCommands library

The code completion also works for texts that have syntactic errors. Thus you can add a declaration of a new component and the code completion will give you its subcomponents, even if the declaration of the component is not yet syntactically correct.

The code completion supports lookup in Modelica classes. When completing e.g. Modelica.Math, the menu will only show functions within that library.

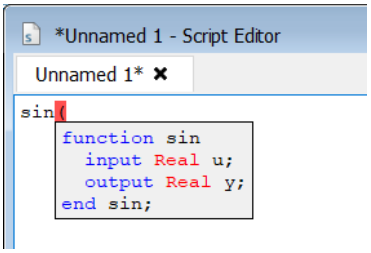
To select the word to input and enter it, do any of the following:

- Click on the wanted word in the menu.

- Navigate to the wanted word and then press **Enter**. You can navigate by the arrow keys, but you can navigate to the next alternative also by clicking **Ctrl+Space** again, or keeping **Ctrl** pressed and clicking **Space** again.

Closing the menu without inputting any word can be done by pressing **Escape** or clicking outside the menu.

Note: You can display a tooltip containing the short description (signature) for a function in the script editor. Type the function name (or use the code completion feature above) and then press left bracket `(` to display the short description. As an example, type `sin` and then left bracket like below:



You can also press **Ctrl+Space** after the left bracket to display the function description.

You can hide the function description by pressing right bracket `)`, **Escape**, **Enter**, **Up**, or **Down**.

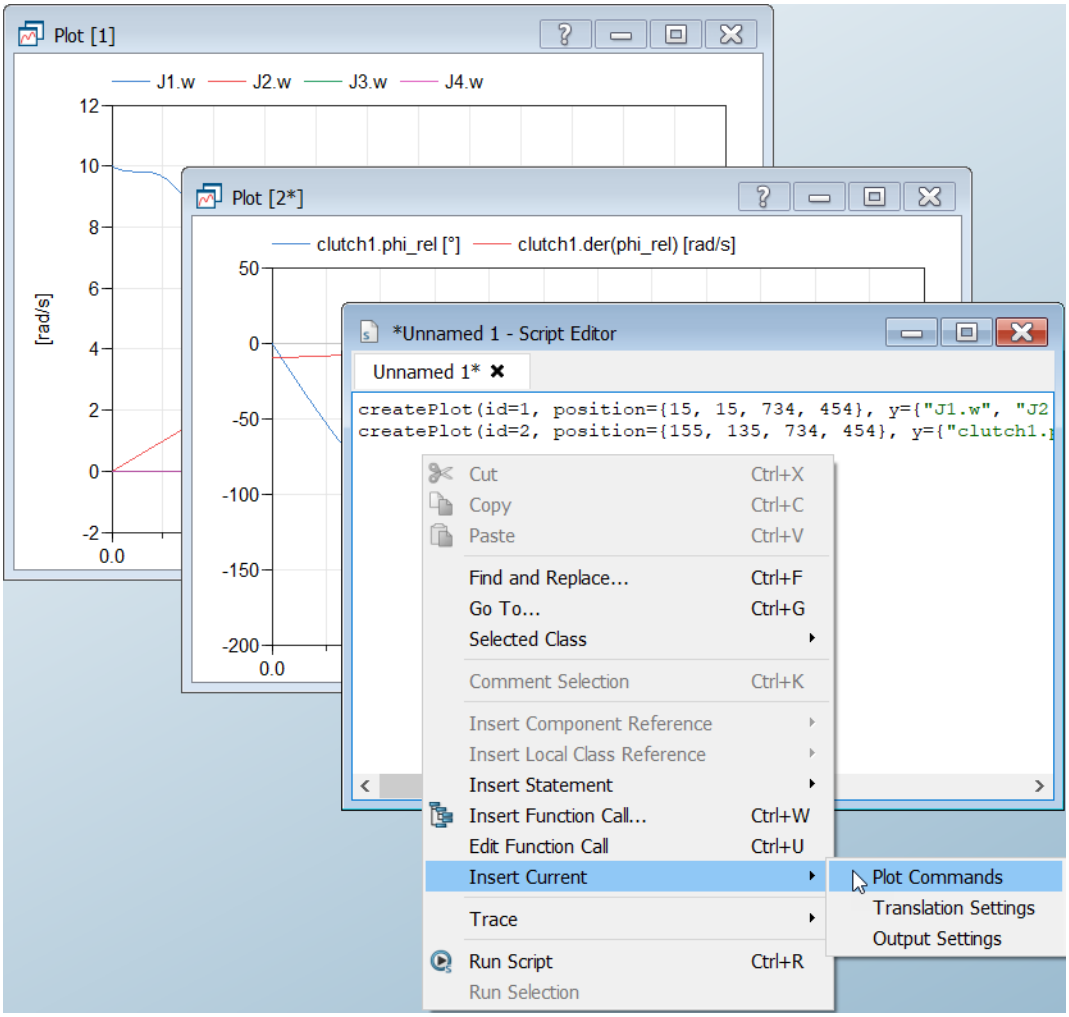
Saving a script




Having edited a script in the scripting editor, it can be saved, in the **Script Editor** tab, by any of the commands **Save Script > Save** or **Save Script > Save As…**, depending on it you want to save it with the current name or if you want to save it to a new script file with a new name. If you click directly on the **Save Script** button, the script is saved to the current file name.

Insert current plot commands in the script

You can insert the script commands for your current plot session (your displayed plots) in the script editor by the context command **Insert Current > Plot Commands**:



This facilitates the creation of scripts for rebuilding a plot session that has been done interactively.

Inserting translation settings in the script

You can insert the current input settings by the context command **Insert Currrent > Translation settings**. These input settings corresponds to a selection of settings in the **Model translation** group of the **Translation** tab of the simulation setup. For details, see the option **Translation settings** in the script editor, that fully corresponds to the option here. See “The Store script for group” starting on page 834.

Inserting output settings in the script

You can insert the current output settings, as the built in function `experimentSetupOutput`, in the script by the context command **Insert Current > Output Settings**. These output settings correspond to the groups **Format**, **Store**, and **Output selection** in the **Output** tab of the simulation setup. For details, see the option **Output settings** in the script editor, that fully corresponds to the option here. See "The Store script for group" starting on page 834.

Executing a script or a selected part of a script from the Dymola script editor



To execute a whole script from the script editor, the script must be opened and active in the script editor. Clicking the **Run Script** button in the header of the script editor will execute the script.

To execute a selected part of a script, select the part and right-click to display the context menu. Select **Run Selection** to execute the selected part.



Note I: By default, the script is executed where it resides, by automatically changing the working directory to the location where the script resides, before executing it. This default behavior can however be changed, in the **Script Editor** tab, by the command **Change To Directory**. The command works in toggle mode, and is by default activated, displayed as depressed.



Note II: By default the script commands are echoed when executing the script. This can be changed, in the **Script Editor** tab, by the command **Echo Commands**. The command works in toggle mode, and is by default activated, displayed as depressed. The state of button/command is saved by the flag `Advanced.EchoScriptCommands`, which is saved between sessions.
Important! The command/button only applies to scripts run from the script editor or from the commands in the script toolbar. (For the built-in function `RunScript` there is an argument `silent` that can be used correspondingly, the default state of echoing corresponds to `RunScript(…, silent=false)`

Tracing using the Dymola script editor

Function calls, variables in functions, and global variables can be traced when executing the script.

Tracing of global variables is always accessible by right-clicking in the script and selecting **Trace > Trace Global Variables**.

To trace a function call or the variables in a function, the function call has to be selected by double-clicking the name before taking up the context menu and selecting **Trace > Trace Function Call** or **Trace > Trace Variables in Function**.

The trace output is displayed in the command log.

5.5.17 Some examples of Modelica script files

Handling of selected parameters (initial values, saving etc)

The function `simulateExtendedModel` is used in some of the examples below. Please also see "simulateExtendedModel" on page 946. Note the example that is given here as well!

When it comes to parameter studies, please also see the features of the Design package in the chapters "Model Experimentation", "Model Calibration", and "Design Optimization".

Setting parameter values before simulation using a text file

This can be obtained by using the command **Script Editor > Generate Script > Variables**. Further selection depends on what the user wants. (Please also see above.)

By selecting **Final** and **All** all available final variable values for a certain simulation will be saved. This can be used when the user wants e.g. stable values from a first simulation as starting point for a second simulation.

The command will create a script file that contains all names and values, and this script file can be easily be modified using e.g. MS Notepad.

The script file can be executed before simulation to set wanted values.

Parameter studies by running Dymola a number of time in "batch mode"

It is possible to create a number of result files, corresponding to different parameter sets. The following simple example shows how to do this for one parameter.

Consider the simple model

```
model Test
   Real x(start=1);
   parameter Real a = 1;
equation
   der(x)=-a*x
end Test;
```

Let this model reside in `C:\BatchRun`.

In this folder, we create a script file `MyBatch.mos`

```
openModel("C:/BatchRun/Test.mo")

for i in 1:10 loop
  simulateExtendedModel("Test", initialNames={"a"},
  initialValues={i}, finalNames={"x"},
  resultFile="test"+String(i));
end for;

Modelica.Utilities.System.exit()
```

This script file will, when executed, open the model and then simulate it 10 times with different values for the parameter "a". ("a" will have value 1 – 10). For each simulation a new

result file will be created (`test1.mat` etc). After the last simulation Dymola will be terminated.

It is common wanting to run such a script file without opening the GUI of Dymola. This can be done using the Command Prompt in MS Windows:

`C:\BatchRun>`**"C:\Program Files\Dymola 2023\bin\Dymola.exe" /nowindow MyBatch.mos**

(assuming the standard location of Dymola.exe).

To not miss translation messages, you can set the flag `Advanced.TranslationInCommandLog=true` in the beginning of your script; that will cause the translation messages to appear in the Command window. You can use e.g. `savelog("translationLog.html")` to save the log in a file. If this produces much text, you can create a code like, in principal:

```
ok=translateModel(…);
if not ok then
//clear log
//set flag, re-translate to catch error
//save log
end if;
```

Setting parameter values when clicking on the Simulate button

Sometimes parameter values should be set before simulation when clicking on the Simulate button.

The annotation `__Dymola_preInstantiate` can be used to specify a function that should be run before a component of the model is checked, translated or simulated. (Please see chapter "Advanced Modelica support", section "Some supported features of the Modelica language", sub-section "Running a function before check/translation/simulation" for more information about this annotation.)

The annotation only work on explicit functions, but since such functions can contain script files; we can "pack" any wanted script file into a user-defined function. This enables e.g. using a script file to set parameter values before translation of a model.

Reading simulation results

The functions `readTrajectory` and `readTrajectorySize` are very useful to read simulation results. The following simple script file will simulate the demo model `CoupledClutches` and then extract the simulation results for the variable `J1.w` and put them in `traj`:

```
simulateModel("Modelica.Mechanics.Rotational.Examples.CoupledCl
utches", resultFile="CoupledClutches")
n=readTrajectorySize("CoupledClutches.mat");
traj=readTrajectory("CoupledClutches.mat", {"J1.w"}, n)
```

To extract the time vector (in order to find the index of a certain time point in the results) the following line can be added:

```
time = readTrajectory("CoupledClutches.mat", {"Time"}, n)
```

Please note that if values for matrices should be read, a space must follow each comma sign when specifying the values, like x[1, 1] in the following function call `readTrajectory("dsres.mat",{"x[1, 1]"},4);`

Of course more than one variable can be handled at the same time, please see below.

Saving variable values in a CSV file

When saving variable values to file the package `DataFiles` can be used. This package is automatically opened when referred to in a script file. (The package is located in Program `Files\Dymola 2023\Modelica\Library`.)

Saving all values into a CSV file

The following script file will store all variables from the model `PID_Controller` in the file `AllVariables.csv`. The model has to be simulated before running the script file (the script file reads values from the .mat file created when the model is simulated).

(The model PID_Controller is a demo example located in `Modelica.Blocks.Examples`.)

```
// Define name of trajectory file (fileName) and CVS file
// (CSVfile)
fileName="PID_Controller.mat";
CSVfile="AllVariables.csv";

// Read the size of the trajectories in the result file and
// store in 'n'
n=readTrajectorySize(fileName);

// Read the names of the trajectories
names = readTrajectoryNames(fileName);

// Read the trajectories 'names' (and store in 'traj')
traj=readTrajectory(fileName,names,n);

// transpose traj
traj_transposed=transpose(traj);

// write the .csv file using the package 'DataFiles'
DataFiles.writeCSVmatrix(CSVfile, names, traj_transposed);
```

Saving selected values in a CSV file

Consider the above example. If e.g. only the signals PI.y and PI.u_m should be stored, the script file can be changed to (only the definition of names must be changed, but it is advisable to also have a better name of the csv file in this example):

```
// Define name of trajectory file (fileName) and CVS file
// (CSVfile)
fileName="PID_Controller.mat";
CSVfile="SelectedVariables.csv";
```

```
// Read the size of the trajectories in the result file and
// store in 'n'
n=readTrajectorySize(fileName);

// Define what variables should be included
names={"PI.y","PI.u_m"};

// Read the trajectories 'names' (and store in 'traj')
traj=readTrajectory(fileName,names,n);

// transpose traj
traj_transposed=transpose(traj);

// write the .csv file using the package 'DataFiles'
DataFiles.writeCSVmatrix(CSVfile, names, traj_transposed);
```

Saving selected variables in a .mat file

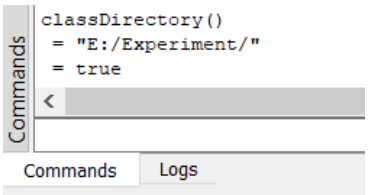
The last example in the above section applies, if `DataFiles.writeCSVmatrix` is exchanged to `writeTrajectory`. The file to store the data must also be changed to a `.mat` file.

The reason for wanting to do this is to decrease the size of the result file for use in e.g. postprocessing with Matlab. In many cases just a subset of signals is needed, and in some cases the size of the result file containing all signals will just be too big to work with.

Getting the path of a currently executed Modelica script

It is possible to get the path of a currently executed `.mos` script by using the `classDirectory()` function.

As an example, adding this function to a line in a script that is located in the folder `E:\Experiment` and then running the script will output, in the command log:



Note that you need to have the option **Echo Commands** active to get this output. (The option is available in the **Script Editor** tab, in the **Options** group.)

5.5.18 Predefined Modelica script files

The following is some examples of where predefined Modelica script files are used.

Model demos

All demos available in Dymola via the command **File > Demos** are handled using script files.

displayunit.mos – handling display units

Unit conversion and the initial default display unit (for e.g. plotting) can be specified by displayunit.mos (and displayunit_us.mos). Please see section "Changing the displayed unit of signals" starting on page 712 for more information.

Conversion script files

A number of conversion script files are available for e.g. conversion of models to new versions of libraries. For more information of such script files, please see the documentation of each library and the chapter "Appendix – Migration".

5.5.19 Built-in functions in Dymola

Dymola has a number of built-in functions. They are presented below, divided in logical groups.

The following groups have been used:

Function groups	Description
Help	List and presents information about functions and flags.
Simulator API	Functions for handling of the simulator API
System	Functions closely related to Simulator API but not directly handling the simulation.
Plot	Functions for plot handling.
Trajectories	Functions for trajectory handling.
Animation	Functions for animation handling.
Matrix IO	Functions handling arrays, matrices and vectors.
Documentaton	Functons to support creation of dynamic reports.
Others	Functions for version handling.

The group "Simulator API" is closely related to "System" and "Plot" is closely related to "Trajectories"

The description below for each function is the header comment displayed when using the function **help** ***functionname*** - or the function **document("*functionname*")**. Additional information is given in brackets [ ].

Important notes:

- **Functions are case sensitive!**
- Arguments must always be added, even for functions having no arguments, for example when using the function getLastError, the call must be `getLastError();` to be valid.
- Selected built-in functions are automatically accessed from DymolaCommands. This library is by default opened when opening Dymola. If not opened, use the command **File > Libraries > DymolaCommands**. The library contains the built-in functions presented here, with the same structure. Displaying them in the package browser enables using the command **Insert Function Call…** to access them. This is very convenient, since that command is available in several editors.

- File paths cannot be specified using "\"; please use "/" or "\\" instead, e.g. `E:/MyExperiment/Plots/...`
- Please note the "Help" group, the function/commands associated with that group can be used to get more information; e.g. inputs and outputs.

Help

Function	Description
document	"write calling syntax for named function" [e.g. `document("importFMU")`]
help functionname	"write calling syntax for named function" [e.g. `help importFMU`] Note. This is not a pure function, and not seen in the group.
help	"List help commands" Note. This is not a pure function, and not seen in the group.

The following routines are available for interactive help:

document

```
document("functionname/flag")
```

The function document gives full documentation for a function or a flag, usually including also an example. For instance, typing `document("importFMU")` in the command input line of the command widow, and pressing **Enter**, gives:

```
function importFMU "Import an FMU"
  input String fileName "The FMU file";
  input Boolean includeAllVariables := true "Include other variables than inputs, outputs and parameters";
  input Boolean integrate := true "Integrate outside the FMU, set to false for co-simulation";
  input Boolean promptReplacement := false "Prompt for name and save location when importing";
  input String packageName := "" "Name of package to insert FMU in";
  output Boolean result "True if successful";
  external "builtin";
  annotation(Documentation(info="
```

Imports an FMU, i. e. unzips, XSL transforms the model description and opens the resulting Modelica model. Note: The model description file from any previous import is replaced. This also applies to the binary library files.

This built-in function corresponds to the command **File > Import > FMU...**.

For more information, please see the manual "Dymola User Manual Volume 2", chapter 6 "Other Simulation Environments", section "FMI Support in Dymola".

Note: For big models it is recommended to set `includeAllVariables=false` to avoid the Modelica wrapper becoming huge.

Example

```
importFMU("C:/test/Modelica_Mechanics_Rotational_Examples_CoupledClutches.fmu", true, true, false, "");
"));end importFMU;
```

An example of a flag is

```
document("Advanced.LogBusSignalSets")
```

that gives:

```
Boolean Advanced.LogBusSignalSets = false "Log bus signal sets (contents of augmentation sets for expandable connectors)";
```

Note that the current value of the flag is also presented.

Note also that for Boolean flags, you can display (and set) flags using a GUI available from the command **Tools > Options**, clicking the **Flags...** button. For more information, refer to the index entry *"flags : GUI for displaying and setting of Boolean, integer, real, and string flags"* in the index in the end of this manual.

help

```
help()
```

Gives a short overview of these commands

```
help functionname/flagname
```

Write the Modelica declaration of the given built-in function or flag, and describe the function/flag (including the value of the flag). This command often gives a shorter documentation than the "document" function above. As an example, typing `help importFMU` in the command input line of the command widow, and pressing **Enter**, gives:

```
function importFMU "Import an FMU"
  input String fileName "The FMU file";
  input Boolean includeAllVariables := true "Include other variables than inputs, outputs and parameters";
  input Boolean integrate := true "Integrate outside the FMU, set to false for co-simulation";
  input Boolean promptReplacement := false "Prompt for name and save location when importing";
  input String packageName := "" "Name of package to insert FMU in";
  output Boolean result "True if successful";
  external "builtin";
end importFMU;
```

Looking at the flag, the command `help Advanced.LogBusSignalSets` gives the same output as using `document`, see that description (and notes) above.

Simulator API

Function	Description
checkConversion	"Verify conversions"
checkModel	"Check a model"
experiment	"Set up default experiment"
experimentGetOutput	"Return the current simulation output configuration"
experimentSetupOutput	"Setup/configure simulation output"
exportInitial	"Generate script from initial values"
exportInitialDsin	"Generate a copy of internal dsin.txt"
exportSSP	"Export an SSP (System Structure Package)"
GetDymolaCompiler	"Get compiler settings (only Windows)"
getExperiment	"Get current experiment setting"
importFMU	"Import an FMU"
importInitial	"Import start values from file"

importInitialResult	"Import start values from file"
importSSP	"Import an SSP (System Structure Package)"
importSSV	"Import an SSV file (SSP parameter set)"
initialized	"Instruct the simulator that states (and other variables) have already been computed from initial equations"
linearizeModel	"Linearize a Modelica model"
list	"List variables on screen (or file)" [The list also includes interactive settings of translator flags. If only interactive variables are of interest, use "variables" instead.]
listfunctions	"Write a list of builtin functions and their descriptions to the screen"
openModel	"Open a Modelica-file"
openModelFile	"Open a Modelica-model from a file"
RunScript	"Run a Modelica script file"
setClassText	"Set the Modelica text for an existing or new class"
SetDymolaCompiler	"Set compiler path and options (only Windows)"
simulateExtendedModel	"Simulate a model with modified parameters and start values"
simulateModel	"Simulate a Modelica model"
simulateMultiExtendedModel	"Multiple simulation of a model with modified parameters and start values"
simulateMultiResultsModel	"Multiple simulation with trajectory storage of a model with modified parameters and start values"
system	"Execute an external command by calling system"
translateModel	"Translate a model"
translateModelExport	"Translate a model to external environment
translateModelFMU	"Translate a model to an FMU"
variables	"List variables on screen (or file)"
verifyCompiler	"Verify compiler settings (only Windows)"
visualize3dModel	"Visualize initial configuration of a 3D model"

Most routines return a Boolean status to indicate the success (true) or failure (false). Some examples of Simulator API functions:

checkModel

```
checkModel(problem="model", simulate=false, constraint=false)
```

Check the model validity. This corresponds to **Check (Normal)** in the menus.

If `simulate=true` in the call, associated commands will also be included in the check. The commands will be executed and the model will be simulated with stored simulation setup. This corresponds to **Check (With simulation)** in the menus.

experiment

```
experiment(StartTime=0.0, StopTime=1.0, NumberOfIntervals=0,
OutputInterval=0.0, Algorithm="",Tolerance=0.0001,
FixedStepSize=0.0)
```

Defines the default simulation setup. `Algorithm` is a string with the name of the integration algorithm; the names correspond to the ones found in the popup-menu and the string is case insensitive. `FixedStepSize` is only used if the algorithm Euler is selected.

The entire command corresponds to the simulation setup reached by the command **Simulation > Setup**, the **General** tab, except the Experiment group in that menu. Note that when it comes to scripting the simulation, there are, for advanced users, some additional flags that are related to the simulation setup, see section "Advanced setup for simulations by flags" on page 782.

exportInitial

```
exportInitial(dsName="....txt", scriptName="....mos")
```

The function generates a Modelica script, such that running the script re-creates the simulation setup. After running the generated script, it is possible to override specific parameters or start-values before simulating. By generating a script from a "steady-state" dsfinal.txt it is possible to perform parameter studies from that point.

Note: This cannot be combined with non-standard setting of fixed for variables if `dsName="dsin.txt"`. All other cases work fine.

exportSSP

```
exportSSP(modelName, filename="")
```

Exports an SSP (System Structure Package). `modelName` is the name of the model or package to export, `fileName` is the SSP file name.

This built-in function corresponds to the command **File > Save > Export SSP…**.

For more information, please see the chapter "FMI, eFMI, and SSP Support in Dymola".

GetDymolaCompiler

```
GetDymolaCompiler()
```

The function extracts the current compiler settings from the **Compiler** tab of the simulation setup, reached by the command **Simulation > Setup**, except the Custom optons settings – those ones are handled by the flags `Advanced.LinkerOptions` and `Advanced.CompilerOptions`, respectivelly.

The compiler settings are extracted to two output parameters of the function, `compiler`, that is the compiler type, and `settings[:]` that is the rest of the settings (except the Custom options settings).

An example of a call and the output of that call for a Microsoft Visual Studio compiler:

```
GetDymolaCompiler()
```

```
 = "vs", {"CCompiler=MSVC", "IntelDir=C:/Program Files
     (x86)/IntelSWTools/compilers_and_libraries_2016.1.146",
   "IntelVsVersion=", "GCC32Path=C:/MinGW/bin/gcc.exe",
   "GCC64Path=C:/MinGW64/mingw64/bin/gcc.exe",
   "MSVCDir=C:/Program Files (x86)/Microsoft Visual Studio
     14.0/Vc",
   "gcc32Disabled=0", "gcc64Disabled=0", "DDE=0", "DLL=0"}
```

An example of a call and the output of that call for a GCC compiler:

```
GetDymolaCompiler()
 = "gcc", {"CCompiler=GCC", "IntelDir=C:/Program Files
     (x86)/IntelSWTools/compilers_and_libraries_2016.1.146",
   "IntelVsNr=3", "IntelVsVersion=vs2015",
   "GCC32Path=C:/MinGW/bin/gcc.exe",
   "GCC64Path=C:/MinGW64/mingw64/bin/gcc.exe",
   "MSVCDir=C:/Program Files (x86)/Microsoft Visual Studio
     14.0/Vc",
   "gcc32Disabled=0", "gcc64Disabled=0", "DDE=0", "DLL=0"}
```

And finally an example for an Intel compiler:

```
GetDymolaCompiler()
 = "intel", {"CCompiler=Intel", "IntelDir=C:/Program Files
     (x86)/IntelSWTools/compilers_and_libraries_2016.1.146",
   "IntelVsNr=3", "IntelVsVersion=vs2015",
   "GCC32Path=C:/MinGW/bin/gcc.exe",
   "GCC64Path=C:/MinGW64/mingw64/bin/gcc.exe",
   "MSVCDir=C:/Program Files (x86)/Microsoft Visual Studio
     14.0/Vc",
   "gcc32Disabled=0", "gcc64Disabled=0", "DDE=0", "DLL=0"}
```

The function can be used, for example, to test if an application relying on a specific compiler setup is supported on a certain machine, by defining and populating the variables `compiler` and `settings` by the call `(compiler, settings)=GetDymolaCompiler()` and then comparing with the demands.

This built-in function corresponds to the built-in function `SetDymolaCompiler` (see that function below). As an example, to get the correct call for `SetDymolaCompiler`, you can, if possible, set the compiler setting you want to have using the **Compiler** tab in the simulation setup, reached by the command **Simulation > Setup**, and then use `GetDymolaCompiler` to retrieve those settings by defining and populating variables as aboe, to be able to use them in a `SetDymolaCompiler(compiler,settings)` call.

Note. The Custom options that can be set in the last part of the **Compiler** tab in the simulation setup are not included in this function, they are handled using separate flags. See section “Compiler tab” on page 817.

getExperiment

```
getExperiment()
```

Get the current experiment (simulation) setting. Outputs from this command are `StartTime`, `StopTime`, `NumberOfIntervals`, `OutputInterval`, `Algorithm`, `Tolerance`, and `FixedStepSize`.

importFMU

```
importFMU(fileName, includeAllVariables, integrate,
promptReplacement, packageName, includeVariables[:])
```

Imports an FMU, i. e. unzips, XSL transforms the model description and opens the resulting Modelica model. Note: The model description file from any previous import is replaced. This also applies to the binary library files.

This built-in function corresponds to the command **File > Open > Import FMU…**.

For more information, including flags for scripting, please see chapter "FMI, eFMI, and SSP Support in Dymola".

importInitial

```
importInitial(dsName="dsfinal.txt")
```

This function sets up integration or linearization to start from the initial conditions given in the file specified (including start and stop-time and choice of integration algorithm). The default is "dsfinal.txt".

(Calling the function `importInitial` with the (unchanged) default file, followed by calling the function `simulate` corresponds to the command **Simulation > Continue > Continue**. The function `simulate` works like `simulateModel`, but works with the default model.)

After calling `importInitial` it is possible to override specific parameters or start-values before simulating by using the usual parameter setting in the variable browser.

Calling the function `importInitial` with a text file that differs from the unchanged default file corresponds to the command **Simulation > Continue > Import Initial…**.

Please see the section "Simulation > Continue > Import Initial…" on page 827 for additional important information.

Note: Basically `importInitial()` corresponds to copying dsfinal.txt (the default variable output file containing variable values etc. at the end of the simulation) to dsin.txt (the default variable input file for a simulation run). Please compare the command below.

importInitialResult

```
importInitialResult(dsResName,atTime)
```

This function is similar to the previous function, with the following exceptions:

- The start value file has to be specified, and it has to be a simulation result, i.e. a file that you can plot/animate.

- The start time has to be specified.
- The integration method will be the one presently selected.

Concerning other information, please see the previous function.

importSSP

```
importSSP(fileName, filterVariables=false, packageName="")
```

Imports an SSP package. `fileName` is the name of the imported SSP package file, `filterVariables` is a Boolean argument, if set to `true` you can filter which FMU variables in the SSP package that are to be exposed in the imported package. `packageName` is the name of the Modelica package where the imported SSP model is to be located. By default, it is imported to a new Modelica library, where it will be the root package. If you specify a Modelica package name, the SSP package will be imported into that Modelica package.

The output argument `result` will contain the full path of the created model if successful, otherwise an empty string.

This built-in function corresponds to the command **File > Open > Import SSP…**.

For more information, please see chapter "FMI, eFMI, and SSP Support in Dymola".

importSSV

```
importSSV(fileName, modelName, packageName="")
```

Imports an SSV file, that is, a SSP parameter set. `fileName` is the name of the imported SSV package file, `modelName` is the name of the existing Modelica model that the parameters are applied to. `packageName` is the name of the package where the new model is inserted.

(The name of the model is taken from the parameter set in the SSV file.) By default, this is an empty string, meaning that the created model is placed at the top level of the package hierarchy.

The output argument `result` will contain the full path of the created model if successful, otherwise an empty string.

This built-in function corresponds to the command **File > Open > Import SSP…** applied on an SSV file.

For more information, please see the chapter "FMI, eFMI, and SSP Support in Dymola".

initialized

```
initialized(allVars=false, isInitialized=true)
```

This function is used to turn off or on initial equations. The call `initialized(isInitialized=true)` sets all states and parameters to fixed and all other variables to free. This means that the initial equations will not be used, corresponding to continuing a simulation. After this call the initial values for all states can be modified using the Variable Browser or Commands Window. Calling `initialized(isInitialized=false)` re-activates the initial equation (this is the default state of a translated model). This means that the simulator will initialize according to the

initial equations at the start of the simulation. If `allVars=true` and `isInitialized=true` the start values for all variables are used to start the simulation. The `allVars` option is not used when `isInitialized=false`.

linearizeModel

```
linearizeModel(problem="", startTime=0.0, stopTime=1.0,
    numberOfIntervals=0, outputInterval=0.0, method="Dassl",
    tolerance=0.0001, fixedstepsize=0.0, resultFile="dslin")
```

The function translates a model (if not done previously) and then calculates a linearized model at the initial values. The linearized model is by default stored in the Dymola working directory in Matlab format as the file `dslin.mat`.

This built-in function corresponds to the command **Simulation > Linearize**. For more information about the content of the dslin.mat file and handling of linearization, please see the section about that command, section "Simulation > Linearize" starting at page 827. In particular, note how to linearize around other values than the initial ones (the corresponding parameters in the function cannot be used to change the time-point of linearization).

list

```
list(filename="VariableValues.txt", variables{"Var1", "Var2"})
```

The function lists (on screen or to a file) the interactive variables in the variable workspace with their type and value. Predefined variables are also described by a comment. In addition, interactive settings of translator flags such as Evaluate are listed.

The output from the function is in alphabethical order, and grouped.

The wildcards * and ? are supported, for example:

- `list(variables={"A*"})` – lists all items starting with `A`.
- `list(variables={"Advanced.*"})` – lists all items starting with `Advanced.` – that is, list all Advanced flags settings.
- `list(variables={"*Output*"})` – lists all items containing `Output` in the text.

It is possible to write the variables to a script file (which can be executed) `filename="script.mos"`, and limit it to certain variables by using `variables={"var1","var2"}`.

The command **File > Clear All** also clears the variable workspace.

listfunctions

```
listfunctions(filter="*", longForm)
```

Writes a list of the built-in functions and their descriptions to the screen.

The wildcards * and ? are supported in the filter.

`longForm` is by default false. If you set it to true, the command lists in "long listing form", that is, the documentation of the built-in functions is listed as well.

Note. This function lists all built-in functions. We recommend using the ones in DymolaCommands in the first place.

openModel

```
openModel(path, mustRead=false)
```

Reads the file specified by `path`, for example `openModel(path="E:\Experiments\MyLib.mo")`, and displays its window. This corresponds to **File > Open** in the menus. Note: This will automatically change directory to the right directory.

`mustRead=false` means that if the file already is opened/read, it is not opened/read again. If `mustRead=true` in such a case the user is promted for removing the present one and open it again. The default value `false` can be useful in scriping, when only wanting to assure that a certain file has been read.

RunScript

```
RunScript(script, silent=false)
```

Executes the specified script file, see example in section "Running a Modelica script file" on page 925. `silent` means that commands are not echoed if this setting is true.

SetDymolaCompiler

This built-in function can be used to set compiler settings as specified in the **Compiler** tab in the simulation setup (reached by the command **Simulation > Setup**), except the Custom group settings, which are handled by separate flags.

The function is complementaty to GetDymolaCompiler, which can be use to extract the input parameter values to use in SetDymolaCompiler, given that the simulation setup has been made to reflect the wanted compiler settings.

Please see section "GetDymolaCompiler" on page 941 for more information and examples.

simulateExtendedModel

```
simulateExtendedModel(problem="", startTime=0.0, stopTime=1.0,
    numberOfIntervals=0, outputInterval=0.0, method="Dassl",
    tolerance=0.0001, fixedstepsize=0.0, resultFile="dsres",
    initialNames[:], initialValues[size(initialNames,1)],
    finalNames[:], autoLoad=true)
```

An extension of `simulateModel` (please see that routine, also for comparison between a number of similar routines). This routine gives the possibility to set parameters and start-values before simulation and to get the final values at end-point of simulation. `autoLoad=true` is default. If false, the result file is not loaded in the plot window (and variables are not replotted).

Notes:

- For `initialNames`, structural parameters that are evaluated cannot be part of that argument, use modifiers for them, as described for `simulateModel` (see that function).

- For `initialValues`, Integer and Boolean variables (Boolean coded as 0 or 1) are also supported, although `initialValues` are stated as Real.

Example: Parameter studies of selected parameters

Consider the demo model `Modelica.Mechanics.Rotational.CoupledClutches`. The parameters `J1.J` and `J2.J` should be varied and the resulting `J1.w` and `J4.w` should be measured and saved at the end of the simulation. That will be the result of the following function call:

Please note that you for this example first have to open the model (using **File > Demos > Coupled Clutches**) since it is a read-only demo.

Entering in the command input line (followed by enter):

```
simulateExtendedModel("Modelica.Mechanics.Rotational.Examples.
CoupledClutches",initialNames={"J1.J","J2.J"},initialValues={2,
3},finalNames={"J1.w","J4.w"});
```

The output visible in the command window will be:

```
                    true
= {6.2134129586543, 0.999999999999999}
```

It can be seen that the function call was executed successfully (`= true`); then the value of `J1.w` (6.213…) and `J4.w` (0.99999…) is presented.

By changing `J1.J` and `J2.J` and simulating the resulting `J1.w` and `J4.w` can be studied.

setClassText

```
setClassText(parentName, fullText)
```

Sets the Modelica text for an existing or new class. `parentName` is the package in which to add a class, `fullText` is the Modelica text to add. If the class specified by `parentName` does not exist, it is ceated. If `parentName` is an empty string, a top level class is created. If you set the Modelica text for an existing class, the old Modelica text is overwritten.

For reading the Modelica text of a class, see section “getClassText” on page 973.

simulateModel

```
simulateModel(problem="", startTime=0.0, stopTime=1.0,
    numberOfIntervals=0, outputInterval=0.0, method="Dassl",
    tolerance=0.0001, fixedstepsize=0.0, resultFile="dsres")
```

Simulate the model for the given time. `method` is a string with the name of the integration algorithm; the names correspond to the ones found in the popup-menu and the string is case insensitive. `fixedstepsize` is only used if the method Euler is selected. Note that file extension is automatically added to `resultFile` (normally `".mat"`). For backwards compatibility the default for `resultFile` is `"dsres"`.

The entire command corresponds to **Simulate** in the menus.

Values specified in the model will be used unless the corresponding modifier is given in the `simulateModel` command.

Note: `translateModel`, `simulateModel`, `simulateExtendedModel`, `simulateMultiExtendedModel`, and `simulateMultiResultsModel` have named arguments (as is indicated above) and the default for problem is "" corresponding to the most recently used model. Thus `simulateModel(stopTime=10,method="Euler");` corresponds to `simulateModel("", 0, 10, 0, 0, "Euler", 1e-4);`

It is possible to specify a model name with modifiers for `translateModel`, `simulateModel`, `simulateExtendedModel`, `simulateMultiExtendedModel`, and `simulateMultiResultsModel`, e.g.

```
for source in {"Step","Constant","Ramp","Sine"} loop
  simulateModel("TestSource(redeclare
Modelica.Blocks.Sources."+source+" Source)");
end for;
```

There are a number of functions that are extensions of `simulateModel`. The following table illustrates the main differences:

Function	Additional input	Output
simulateModel		Trajectories for one simulation.
simulateExtendedModel	Parameter values and start values (for one simulation).	Endpoints for one simulation.
simulateMultiExtendedModel	As simulateExtended model, but for several simulations.	Endpoints for several simulations.
simulateMultiResultsModel	As simulateExtended model, but for several simulations.	Trajectories for several simulations.

simulateMultiExtendedModel

```
simulateMultiExtendedModel(problem="", startTime=0.0,
    stopTime=1.0, numberOfIntervals=0, outputInterval=0.0,
    method="Dassl", tolerance=0.0001, fixedstepsize=0.0,
    resultFile="dsres", initialNames[:],
    initialValues[:,size(initialNames,1)], finalNames[:])
```

An extension of `simulateModel` (please see that routine, also for comparison between a number of similar routines). The function handles a number of simulations. For each simulation it is possible to set parameters and start-values before simulation and to get the final values at end-point of simulation.

Notes:

- For `initialNames`, structural parameters that are evaluated cannot be part of that argument, use modifiers for them, as described for `simulateModel` (see that function).
- For `initialValues`, Integer and Boolean variables (Boolean coded as 0 or 1) are also supported, although `initialValues` are stated as Real.

- If a simulation failure occurs, the whole function call is terminated. To make the function call run all parameter values except the ones that fails, add "`terminal`" as last argument in `finalNames`.
- The function by default runs the simulations in parallel on multiple cores on the CPU. By default, all cores are used, but if needed the number of parallel runs can be specified by setting the flag `Advanced.ParallelSimulations`. The default value is 0, meaning all cores are used. Setting it to 1 disables parallelization; as it will only run on one core.

The function is valuable e.g. when wanting to study the best parameter setup or the robustness of a parameter setup for a static simulation (no states involved). For the best solver selection in this case, see section "Steady state initialization" on page 1037.

Example (comparison to the example above)

Entering in the command input line (followed by enter):

```
simulateMultiExtendedModel("Modelica.Mechanics.Rotational.
Examples.CoupledClutches", initialNames={"J1.J","J2.J"},
initialValues=[2,3;3,4;4,5], finalNames={"J1.w", "J4.w"})
```

The output visible in the command window will be:

```
          true
=  [6.2134129586543  1.0]
   [7.4835581910107  1.0]
   [8.1074463797378  1.0]
```

simulateMultiResultsModel

```
simulateMultiResultsModel(problem="", startTime=0.0,
    stopTime=1.0, numberOfIntervals=0, outputInterval=0.0,
    method="Dassl", tolerance=0.0001, fixedstepsize=0.0,
    resultFile="dsres", initialNames[:],
    initialValues[:,size(initialNames,1)], resultNames[:])
```

An extension of `simulateModel` (please see that routine, also for comparison between a number of similar routines).

Notes:

- You must set `numberOfIntervals`. If the value is `0`, the function will not work.
- For `initialNames`, structural parameters that are evaluated cannot be part of that argument, use modifiers for them, as described for `simulateModel` (see that function).
- For `initialValues`, Integer and Boolean variables (Boolean coded as 0 or 1) are also supported, although `initialValues` are stated as Real.
- The function by default runs the simulations in parallel on multiple cores on the CPU. By default, all cores are used, but if needed the number of parallel runs can be specified by setting the flag `Advanced.ParallelSimulations`. The default value is 0, meaning all cores are used. Setting it to 1 disables parallelization; as it will only run on one core.

An example of call is:

```
simulateMultiResultsModel(
"Modelica.Mechanics.Rotational.Examples.CoupledClutches",
stopTime=1.2, numberOfIntervals=10, resultFile="CoupleCluches",
initialNames={"freqHz"}, initialValues=[0.1;0.2;0.3;0.4],
resultNames={"J1.w","J3.w"});
```

translateModel

```
translateModel(problem="model")
```

Compile the model (with current settings). This is similar to **Simulation > Translate (Normal)** in the menus. To get the exact same result as the menu command you need to do `closeModel();translateModel(problem="model")` since the script command avoids unnecessary translations for performance reasons. The menu command forces a re-translation in case the first translation failed for external reasons; e.g. out of disk space.

Note: If you give the name of the model, you can skip `translateModel` and go directly to `simulateModel`.

Note: `translateModel`, `simulateModel`, `simulateExtendedModel`, `simulateMultiExtendedModel`, and `simulateMultiResultsModel` have named arguments (as is indicated above) and the default for problem is "" corresponding to the most recently used model. Thus `simulateModel(stopTime=10,method="Euler");` corresponds to `simulateModel("", 0, 10, 0, 0, "Euler", 1e-4);`

It is possible to specify a model name with modifiers for `translateModel`, `simulateModel`, `simulateExendedModel`, `simulateMultiExtendedModel`, and `simulateMultiResultsModel` e.g.

```
for source in {"Step","Constant","Ramp","Sine"} loop
  simulateModel("TestSource(redeclare
Modelica.Blocks.Sources."+source+" Source)");
end for;
```

It is possible to set parameters before/at translation;

```
translateModel("Modelica.Blocks.Examples.PID_Controller(PI(k=20
0))")
```

will set the parameter PI.k to 200 before/when translating.

translateModelExport

```
translateModelExport(model)
```

Translates the active model to code executable on any platform without a Dymola license at the target system.

This built-in function corresponds to the command **Simulation > Translate > Export**.

This functionality demands license. For more information, please see the chapter “Simulation Environments”, section “Code and Model Export”.

translateModelFMU

```
translateModelFMU(modelToOpen, storeResult, modelName,
fmiVersion, fmiType, includeSource, includeImage)
```

Translates a model to an FMU. The input string `modelToOpen` defines the model to open in the same way as the traditional `translateModel` command in Dymola.

This built-in function corresponds to the commands **Simulation > Translate > FMU**. Refer to section "Simulation > Translate > FMU" on page 797.

For more information, including flags for scripting, please see the chapter "FMI, eFMI, and SSP Support in Dymola".

variables

```
variables(filename="VariableValues.txt", variables{"Var1",
"Var2"})
```

Works as `list` above, but does not list interactive settings of translator flags.

verifyCompiler

```
verifyCompiler()
```

The built-in function works as the button with the same name in the **Compiler** tab in the simulation setup menu; a small file is executed to test the selected compiler, `true` is returned if the test works, otherwise a message. This message is also added as last error that can be retrieved by `getLastError`. **Note**: This function only works in Windows.

System

Function	Description
AddModelicaPath	"Append to ModelicaPath"
cd	"Change directory and/or report current directory"
classDirectory	"Return current directory" [the directory where the call resides; useful for accessing local external files]
clear	"Clear everything"
clearFlags	"Clear flags and integer constants"
clearlog	"Clear current log window"
eraseClasses	"Erase the given classes"
Execute	"Execute file/command"
exit	"Exit Dymola"
getLastError	"Return the last error message from the last command, and the number of errors, warnings, and info messages"
savelog	"Save log in file"
saveSettings	"Stores current setting on a file"
saveTotalModel	"Save a total model"
ShowComponent	"Show component"
showMessageWindow	"Show/hide log"
trace	"Trace execution of interactive functions"

AddModelicaPath

```
AddModelicaPath(path, erase=false)
```

The function appends a directory, specified by the string variable `path`, to the Modelica path environment variable (MODELICAPATH) (unless `erase=true`).

An alternative is to use

```
Modelica.Utilities.System.setEnvironmentVariable("MODELICAPATH"
, "…");
```

Notes:

- The menus **File > Libraries** and **File > Demos** are rebuilt after a change of MODELICAPATH.
- To keep the change of Modelica path between sessions, this functionality has to be addedin setup.mos. It is currently not possible to store the change of the Modelica path from GUI.

cd

```
cd(dir="")
```

Displays or changes the current working directory. `cd` displays the current working directory, `cd("path")` changes the current working directory to the path specified, e.g. `cd("E:/Experiments")`. It is possible to also enter `cd E:/Experiments`.

If no changes have been made by any command etc., the default working directory for Dymola is `…\Documents\Dymola`.

Notes:

- The current working directory can also be displayed/changed/accessed by the **File > Working Directory** commands, from the **File** menu or from the **Working Directory** button in the Commands window. See section “Handling the current working directory” on page 770 for more information.
- The startup directory, that is, the directory that will be current directory when starting Dymola again, can be set in different ways. See index entry “*directory : startup directory*” in this manual for more information.

classDirectory

```
classDirectory()
```

Returns current directory (the local directory where the call resides). This can be used, for example, to access local external files.

However, the presently recommended way is to use URIs instead, by using `Modelica.Utilities.Files.loadResource`. This function call returns a file reference when an URI is input. Note that the specified URI must be a string literal or a string parameter.

clearFlags

```
clearFlags()
```

Resets all flags and integer constants to their default values.

eraseClasses

```
eraseClasses({"M1","PackageA.M2", ...})
```

Function to unload the given models. It requires that no models outside of this list depend on them. This is not primarily an interactive function, but designed to be called by other programs constructing and changing models. The function corresponds to **Unload** in the package browser.

getLastError

```
getLastError()
```

Returns the last error message from the last command. If the last command was successful, an empty string is returned. For check, translate, etc, the log is returned.

The number of errors, warnings, and info messages reported in the translation log are also returned, as last part parts of the output from the function.

savelog

```
savelog(fileName=".....")
```

The function saves the command log on a file. Please note that depending on file extension specified, filtering of the content saved is activated or not. If a `.txt` file extension is used, all text in the log is saved. If however a `.mos` extension (e. g. `"fileName=MyLog.mos"`) is used, neither outputs from Dymola (results etc.) nor commands that have no equivalent Modelica function will be included in the saved script file. The function corresponds to the **File > Save Log…**command.

Using the `.mos` extension (creating a script file) enables saving e. g. a promising sequence of interactive events for further reuse and development. The `.txt` extension can be used when documenting.

saveSettings

```
saveSettings(fileName=".....", storePlot=false,
storeAnimation=false, storeSettings=false,
storeVariables=false, storeInitial=true,
storeAllVariables=true, storeSimulator=true,
storePlotFilenames=false)
```

The function `saveSettings` corresponds to the command **Simulation > New Script > Generate Script…** except storing of the command log and storing the script file as a command in the model. (Storing of the command log can be handled by the function `savelog`.) Please see the section “Script Editor > Generate Script” on page 833 for more information.

Please note that if a script file should be created, the file extension must be `.mos` (e.g. `fileName="MyScript.mos"`).

When storing variable values, a condition is that `storeVariables=true` in the function call. `storeInitial=false` will store final values. `storeAllVariables=false` will store only parameters and states.

If you want to store a plot with curves from different result files, set `storePlotFilenames` to `true`.

Plot

Function	Description
calculateNumberPrecision	"Calculate the needed precision to print the numbers of the input vector"
clearPlot	"Erase curves and annotations in the diagram"
createPlot	"Create a plot window with all settings"
createTable	"Create a table window with all settings"
ExportPlotAsImage	"Export (save) a plot window to a .png image"
plot	"Plot given variables"
plotArray	"Plot given data"
plotArrays	"Plot given data"
plotDiscretized	"Discretized plot"
plotDocumentation	"Add documentation to a plot window"
plotExpression	"Plot an expression"
plotExternal	"Show external document or image"
plotHeading	"Add a heading to a plot window"
plotMovingAverage	"Plot moving average in the active diagram of a plot window"
plotParametricCurve	"Plot parametric curve"
plotParametricCurves	"Plot parameteric curves"
plotRowColumnLabels	"Set column and row labels in plot window"
plotScatter	"Scatter plot"
plotSignalDifference	"Plots the discrete difference of a signal, defined as y(i)=u(i)-u(i-1). "
plotSignalOperator	"Plot signal operator in the active diagram of a plot window"
plotSignalOperatorHarmonic	"Plot signal operator in the active diagram of a plot window"
plotText	"Plot text in the active diagram"
plotTitle	"Add a title to a plot window"
plotWindowSetup	"Generate a createPlot command of a plot window or a createTable command of a table window"
printPlot	"Plot and print given variables"
printPlotArray	"Plot and print given data"
printPlotArrays	"Plot and print given data"
removePlots	"Remove all plot windows. Optionally close all result files."

removeResults	"Removes the specified results or, if results input is not set, removes all results from plot windows"
signalOperatorValue	"Return the value of a signal operator for a given variable"

calculateNumberPrecision

```
calculateNumberPrecision(values[:])
```

Given any array of numbers, calculates the number of decimals needed to accurately represent the numbers, as the Integer output `presision`. The function can be used, for example, to adapt result file names to small parameter values to be able to differ the result files when for example plotting very small sweeped parameter values.

clearPlot

```
clearPlot(id=0)
```

Erases curves and annotations in the diagram. `Id` is identity of window (0 means last).

createPlot

```
createPlot(id, position[4], x, y[:], heading, range[4], erase,
autoscale, autoerase, autoreplot, description, grid, color,
online, legend, timeWindow, filename, legendLocation,
legendHorizontal, legendFrame, supressMarker, logX, logY,
legends[size(y,1)], subplot, uniformScaling, leftTitleType,
leftTitle, bottomTitleType, bottomTitle, colors[size(y,1),3],
patterns[size(y,1)], markers[size(y,1)],
thicknesses[size(y,1)], range[2], logY2, rightTitleType,
rightTitle, axes[size(y,1)], timeUnit, displayUnits[size(y,1)],
showOriginal, showDifference _window)
```

Create a plot window with all settings.

This built-in function contains a number of input parameters also used in other built-in functions documented below. All parameters are input parameters except the last one. Some parameters are further commented in notes below the table.

Notes:

- If you have a plot already, the command **Simulation > New Script > Generate Script…**, selecting **Plot setup** will produce a script (.mos) file with relevant flags and the corresponding `createPlot` function call for the plot. See example below, note the additional setting **Include result filenames**. (For the command itself, see "Script Editor > Generate Script" on page 833.)
- As an alternative to the above, however without creating a script and without being able to use the additional setting **Include result filenames**, you can use the command **Simulation > Commands > Add Command**, selecting **Plot window setup**, to save a function call in the model as a command. See section "Simulation > Commands > Add Command" starting on page 793 for more information.

- There is no scripting support for merging of diagram cells; this is only possible using GUI commands.

Parameter	Type and default	Comment
id	Integer = 0	Window id. See (1).
Position	Integer[4]	Window position. See (2).
X	String = "Time"	Independent variable.
Y	String[:]	Variables. See (3).
Heading	String=" "	Heading.
Range	Real[4] = {0.0,1.0,0.0,1.0}	Range.
Erase	Boolean = true	Start with a fresh window.
Autoscale	Boolean = true	Autoscaling of y-axis.
Autoerase	Boolean = true	Erase previous when replotting. See (4)
autoreplot	Boolean = true	Replot after simulation.
Description	Boolean = false	Include description in label.
Grid	Boolean = false	Add grid.
Color	N/A	Deprecated.
Online	Boolean = false	Online plotting.
Legend	Boolean = true	Variable legend.
timeWindow	Real = 0.0	Time window for online plotting.
Filename	String = " "	Result file to read data from. See (5).
legendLocation	Integer = 1	Where to place the legend. (1=above, 2=right, 3=below, 4-7=inside). See (6).
legendHorizontal	Boolean = true	Horizontal legend.
legendFrame	Boolean = false	Draw frame around legend.
supressMarker	N/A	Depreciated.
logX	Boolean = false	Logaritmic X scale.
logY	Boolean = false	Logaritmic Y scale.
Legends	String[size(y,1)]	Legends. See (7).
subPlot	Integer = 1	Subplot number. See (8).
uniformScaling	Boolean = false	Same vertical and horizontal axis increment.
leftTitleType	Integer = 0	Type of left axis title (0=none, 1=description, 2=custom). For alt. 2, see next parameter.
leftTitle	String = " "	Custom left axis title. See (9).
bottomTitleType	Integer = 0	Type of bottom axis title (0=none, 1=description, 2=custom). For alt. 2, see next parameter.
bottomTitle	String = " "	Custom bottom axis title. See (9).
Colors	Integer[size(y,1),3]	Line colors. See (7).
Patterns	LinePattern[size(y,1)]	Line patterns. See (7).
Markers	MarkerStyle[size(y,1)]	Line markers. See (7).
Thicknesses	Real[size(y,1)]	Line thicknesses. See (7).
Range2	Real[2] = {0.0,1.0}	Range, right vertical axis.

logY2	Boolean = false	Logarithmic right Y scale.
rightTitleType	Integer = 1	Type of right axis title (0=none, 1=description, 2=custom). For alt. 2, see next parameter.
rightTitle	String = " "	Custom right axis title. See (9).
Axes	Integer[size(y,1)]	Vertical axes, 1=left, 2=right.
timeUnit	String= "s"	Time unit
displayUnits	String[size(y,1)]	Display units. Empty string means use the default display units.
showOriginal	Boolean = true	When enabled, original curves are shown. See (10).
showDifference	Boolean = false	When enabled, the difference between curves are shown. See (10).
plotInAll	Boolean = false	When enabled, variables from all found result files are plotted. See (11).
_window	Integer	Output parameter. See (12).

Notes:

1. Id can be set to -1. This means that the Id will automatically be selected as the highest id in any plot window present, +1, for example `Id=11` if the present plot window with the highest id has `Id=10`. (This also means that `Id=-1` will not destroy any present plot window.)
2. A position {15, 10, 400, 283} indicates that the upper left corner of the plot window has the position (15, 10) and the lower corner to the right has the position (400, 283), calculated from an origin of the upper left corner of the enclosing space.
3. Variable names with and without result file name and sequence name (in brackets) are supported: "resultFile[resultID].VariableName", as well as the shorter "VariableName". For the longer name, resultID `end` and `end-1` corresponds to the latest and second latest result files; the absolute sequence number can be used in other cases. The shorter name refers to the latest result file. Long and short name examples: `"CoupledClutches[end].J1.w"` and `"J1.w"`. **Note**: To negate a curve, you can put a minus sign in front of the signal, for example `"-J1.w"`.
4. This parameter is not influenced by the flag `Advanced.DefaultAutoErase`; this flag is only used when creating plot window by GUI commands.
5. The result file name is by default not included in the `createPlot` command when for example saving plots by creating scripts with the command **Simulation > New Script > Generate Script…**, selecting **Plot setup** will. However, there is a setting **Include result filenames** in the menu of the generate script command that can be ticked to include the result file name. This is needed if you want to create a script for a plot with curves from different result files.
6. This parameter corresponds to the Location group in the Legend tab in the plot setup menu. 4=Top-Left, 5=Top-Right, 6=Bottom-Left, 7=Bottom-Right.
7. See the built-in function `plot` below for description.

8. To create a plot with subplots several `createPlot` commands with the same id but different values of subplot have to be executed.
9. Macros are supported in the custom titles. For more information, please see the small section “Creating plot axis titles by macros” after the example below.
10. These arguments correspond to the GUI settings **Original**, **Difference**, and **Original and Difference** in the **More** section of the variable browser.
11. When set to true, all open result files will be searched for the variables specified to be plot, and all found occurences are plotted. This is useful if you want to compare the result of different simulations.
12. This output parameter will have the same value as the parameter id for the created plot window.

Example: To illustrate how `createPlot` might look like (and also some flags from plot), use the command **File > Demos** to open the model Coupled clutches, then use the command **Simulation > Commands > Simulate and Plot** to get a plot window.

Now use the command **Simulation > New Script > Generate Script…**, selecting **Plot setup** to save a script (.mos) file. (There is also a setting **Include result filenames**, ticking this setting gives an extra line; see comment in the script below.) Give it a name, e.g. PlotTest, and keep the default directory for saving. Note what directory that is.

Finding the file and opening it in the Dymola script editor (or in e.g. Notepad) will display:

```
// Script generated by Dymola Wed Dec 02 10:48:17 2015
// Plot commands
removePlots(false);
Advanced.FilenameInLegend = false;
Advanced.SequenceInLegend = true;
Advanced.PlotLegendTooltip = true;
Advanced.FullPlotTooltip = true;
Advanced.DefaultAutoErase = true;
Advanced.Legend.Horizontal = true;
Advanced.Legend.Frame = false;
Advanced.Legend.Transparant = true;
Advanced.Legend.Location = 1;
Advanced.FilesToKeep = 2;
createPlot(id = 1,
 position = {15, 10, 584, 421},
 y = {"J1.w", "J2.w", "J3.w", "J4.w"},
 range = {0.0, 1.2000000000000002, -2.0, 11.0},
 grid = true,
 filename = "dsres.mat", (only if Include result filenames is set)
 colors = {{28,108,200}, {238,46,47}, {0,140,72},
 {217,67,180}});
```

Creating plot axis titles by macros

As remarked in note (8) above, the input parameters leftTitle, rightTitle, and bottomTitle in the built-in function createPlot support macros in the input string. The following macros (and combinations of them) are supported:

- `%name` (the signal name)
- `%unit` (the current signal display unit)
- `%desc` (the signal description)
- `%hname` (either "Time" or the name of the independent variable, if such is used)
- `%hunit` (the horizontal display unit, which is the time unit or the display unit flor the independent variable, if such is used)
- `%title` (is hat is normally shown as title for the horizontal axis, but will also show "Time[s]" if plotted against time and using seconds. The latter is otherwise hidden.

Notes:

- For the string to be used, the title type has to be defined as custom.
- For the first three input parameters avove, if more than one signal is plotted, the macros will contain empty strings.

createTable

```
createTable(id, position[4], x, y[:], heading, autoerase,
autoreplot, description, grid, color, online, legend,
timeWindow, filename, legends[size(y,1)], timeUnit,
displayUnits[size(y,1)], showOriginal, showDifference _window)
```

Create a table window with all settings. The built-in function corresponds to `createPlot`, see this function above for explanatinons of the parameters.

ExportPlotAsImage

```
ExportPlotAsImage(filename="PlotImageName.png", id=-1,
includeInLog=true, onlyActiveSubplot=true)
```

Export (save) a plot window as an image. The image can only be saved in .png format. The parameter id specifies what plot window will be saved. The default -1 means the first (lowest number) plot window in the Dymola main window. The `includeInLog` specifies whether the plot should be included in the command log. The `onlyActiveSubplot` specify whether all subplots should be included, or only the active subplot.

Example: `ExportPlotAsImage(E:/MyExperiment/Plots/Plot3.png, id=3)` will save the plot window Plot[3] as the image Plot3.png in the folder E:\MyExperiment\Plots. It will also be saved in the command log.

plot

```
plot(y[:], legends[size(y,1)], plotInAll, colors[size(y,1),3],
patterns[size(y,1)], markers[size(y,1)], thicknesses[size(y,1),
], axes[size(y,1)])
```

Plot the given variables in the plot window. It is currently not possible to set ranges or independent variable. The function can also handle table windows.

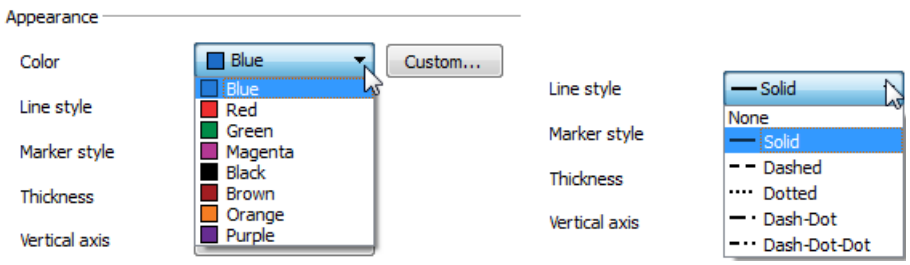
Note: the argument is a vector of strings; the names correspond to the names when selecting variables in the plot window. Subcomponents are accessed by dot-notation.

Variable names with and without result file name and sequence name (in brackets) are supported, as well as negating curves. See comment on variables in the built-in function `createPlot` above.

The built-in function `size()` works the following:

- `size(X,i)` returns the i:th size of an array X, 1<=i<=ndims(X),
- `size(X)` returns a vector of sizes=`{size(X,1),…, size(x,ndims(X)}`

The plot window features of custom-defined color, line style, marker style, thickness, and vertical axis are supported; the possible selections in the GUI are (see further below for default values of line style, marker style and thickness):



corresponds to the following input parameter values in the function:

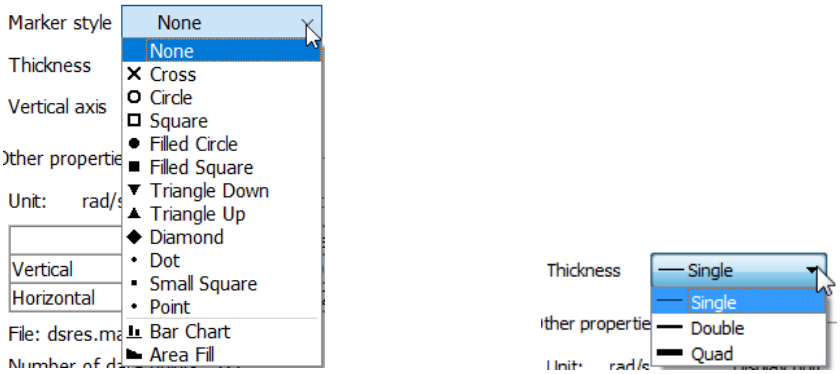
Colors:

```
{28,108,200}
{238,46,47}
{0,140,72}
{180,56,148}
{0,0,0}
{162,29,33}
{244,125,35}
{102,44,145}
```

Patterns:

```
LinePattern.Solid
LinePattern.Dash
LinePattern.Dot
LinePattern.DashDot
LinePattern.DashDotDot
```

(It might be noted that concerning colors, some more brilliant colors are blue {0,0,255}, red {255,0,0}, green {0,128,0}, magenta {255,0,255}. They are used in some of the examples below.)



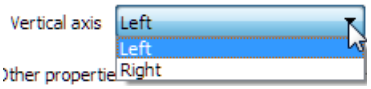
corresponds to the following input parameter values in the function:

Markers:

```
MarkerStyle.None
MarkerStyle.Cross
MarkerStyle.Circle
MarkerStyle.Square
MarkerStyle.FilledSquare
MarkerStyle.TriangleDown
MarkerStyle.TriangleUp
MarkerStyle.Diamond
MarkerStyle.Dot
MarkerStyle.SmallSquare
MarkerStyle.Point
MarkerStyle.BarChart
MarkerStyle.AreaFill
```

Thicknesses:

```
0.25
0.5
1.0
```



Corresponds to the following input values in the function:

```
1
2
```

The corresponding GUI is described in the **Variables** tab; see “Variables tab” on page 843.

The default values of line style, marker style, and thickness are `LinePattern.Default`, `MarkerStyle.Default`, and `-1`, respectively. They mean auto-assign, which normally means that the line style will get the value `LinePattern.Solid`, the marker style will get the value `MarkerStyle.None`, and the line thickness will get the value `1`.

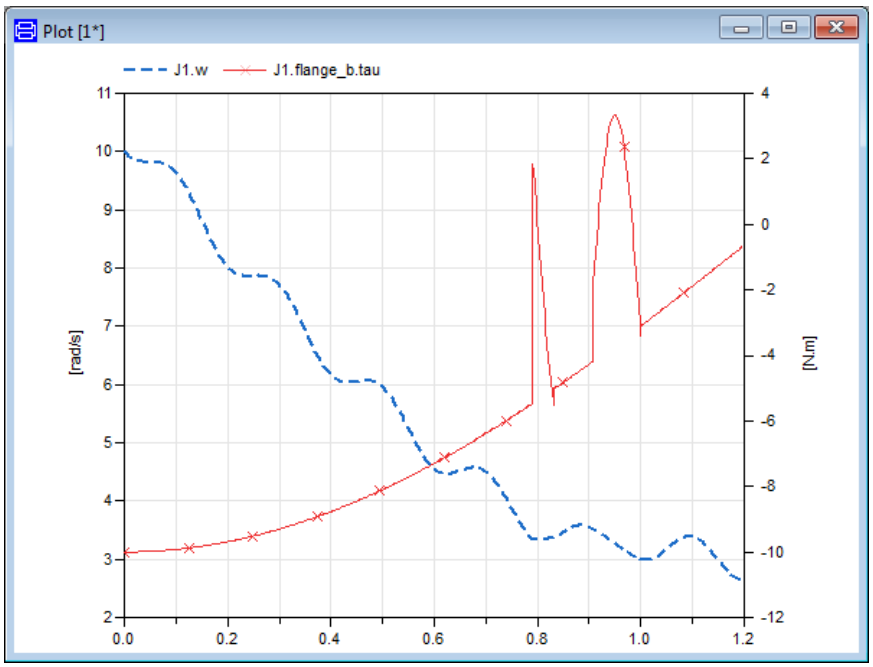
Example

Example of `plot()`:

Using the demo CoupledClutches, the function call

```
plot({"J1.w", "J1.flange_b.tau"}, colors={{28,108,200},
{238,46,47}}, patterns={LinePattern.Dash, LinePattern.Solid},
markers={MarkerStyle.None, MarkerStyle.Cross},
thicknesses={0.500000, 0.250000}, axes={1, 2});
```

corresponds to:



plotArray

```
plotArray(x[:], y[size(x,1)], style, legend, id, color[3],
pattern, marker, thickness, erase, axis, unit)
```

X-y plot for plotting of data computed in functions or scripts. For plot of arrays, please see the next function.

To plot a set of data points from the result file, use

```
Plot({"x[1].a", "x[2].b[1]"})
```

The `id` is the identity of the window (default 0 means last). Concerning size, pattern, marker and thickness, please see the `plot()` built-in function above. (The input `style` is deprecated.) The `axis` is 1 for left vertical axis, 2 for right vertical axis, 1 is default.

For `unit` (default empty string) note the following:

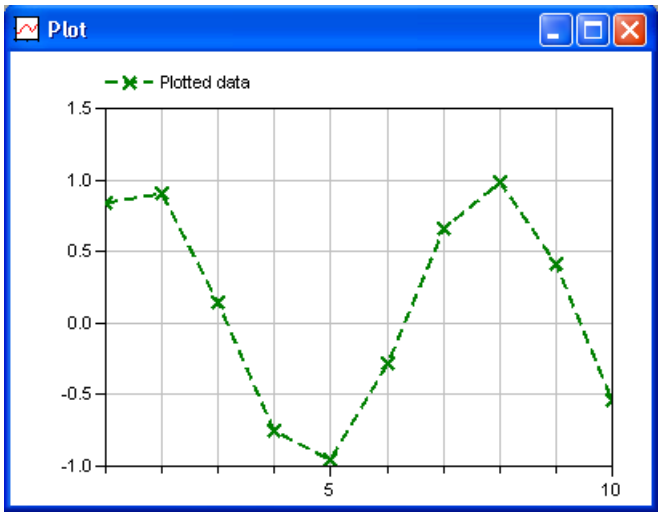
- Since the resulting curve has a unit, the **Display Unit** command is available in the context menu of the resulting curve.
- The `unit` string can be enhanced to also specify the display unit by using the format unit|displayUnit, for example `unit="rad/s|rpm"`.

Example (not from result file)

The function call

```
plotArray(x=1:10,y=sin(1:10), pattern=LinePattern.Dash,
marker=MarkerStyle.Cross,color={0,128,0},thickness=0.5,
legend="Plotted data");
```

will result in:



plotArrays

```
plotArrays(x[:], y[size(x,1),:], style[:], legend[:], id,
title, color[size(y,2),3], patterns[size(y,2)],
markers[size(y,2)], thicknesses[size(y,2)], axes[size(y,2),
units[size(y,2)])
```

X-y plot for plotting of data computed in functions or scripts. Note similarity with the above function. (The input `style[:]` is deprecated.) Note also the Example 2 in “plotSignalOperator” on pge 967.

plotDiscretized

```
plotDiscretized(vars[:], legend, independent[size(vars,1)], id,
erase)
```

Creates a discretized plot for a non-constant array. For more information about discretized plots, see section “Plotting a 1D discretization” starting on page 631.

Note that the built-in function allows you to plot several variables in the same discretized plot window.

plotDocumentation

```
plotDocumentation(doc, id)
```

- Setting `doc` to an empty string removes the documentation, if any. HTML formatting is supported. This functionality is also available from the GUI, see "Printing the result.

Note that the argument `fileName` in the built-in function also supports URL´s, in addition to file paths.

Adding titles and documentation to plot windows" on page 748.

plotExpression

```
plotExpression(mapFunction, eraseOld, expressionName, id, axis,
unit)
```

The function plots an expression in a specified plot window. The corresponding GUI handling is described in section "Plotting general expressions" starting on page 736.

The `expressionName` is the description string of the expression; it will be displayed as the label of the expression. The `id` is the identity of the plot window, where "0" is the last window, -1 the second last etc. The `axis` is the vertical axis, 1 (default) is the left axis, 2 the right axis. The `unit` gives possibility to set unit/display unit on the plot expression in the form "unit|displayUnit" – to set only unit, use only "unit".

Example

```
plotExpression(apply(CoupledClutches[end].J1.w+CoupledClutches[
end-1].J1.w), false,
"CoupledClutches[end].J1.w+CoupledClutches[end-1].J1.w", 1,
unit="rad/s|deg/s");
```

plotExternal

```
plotExternal(fileName, window_id)
```

This function plots external files, in HTML, SVG, or PNG format. In addition, HTML plotting scripts generated in Python are supported. For more information, see section "Plotting of external files" on page 746.

plotHeading

```
plotHeading(textString="", fontSize=0, fontName="",
lineColor[3]={0,0,0}, textStyle[:],
horizontalAlignment=TextAlignment.Center, id=0)
```

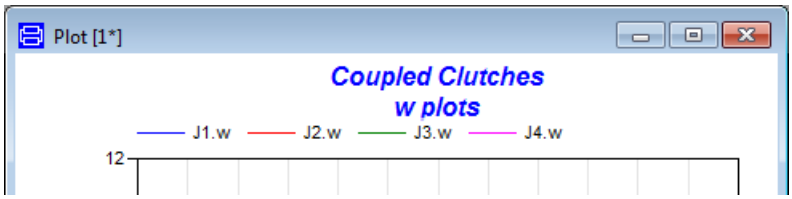
This function creates a heading in a plow window. An empty string as `textstring` removes the heading. `fontSize=0` means that the default base font size is used. For more about font size, and about `textStyle` and `horizontalAlignment`, see section "plotText" on page 969.

Example

The function call

```
plotHeading(textString="Coupled Clutches\nw plots",
fontSize=12, lineColor={0,0,255}, textStyle={TextStyle.Bold,
TextStyle.Italic});
```

will create the following heading:



plotMovingAverage

plotMovingAverage(variablePath, startTime, stopTime, intervalLength, order, id)

For more information about the Moving Average signal operator, see “Available signal operators” starting on page 722.

plotParametricCurve

```
plotParametricCurve(x[:], y[size(x,1)], s[size(x,1)], xName,
yName, sName, legend, id, color[3], pattern, marker, thickness,
labelWithS, erase, axis)
```

The function plots a curve defined by a parameter; the x(s) – y(s) plot. The corresponding GUI is described in section “Displaying parametric curve” on page 742.

`labelWithS` will present parameter labels in the curve if set to true, it corresponds to the context menu command Parameter Labels. See example in the above reference.

plotParametricCurves

```
plotParametricCurves(x[:, size(s,1)], y[size(x,1), size(s,1)],
s[:], xName, yName, sName, legends[:], id, colors[size(y,1),
3], patterns[size(y,1)], markers[size(y,1),
thicknesses[size(y,1)], labelWithS, axes[size(y,1)])
```

The function plots curves defined by x(s) – y(s). The function is an extension of the function above, covering multiple curves.

plotRowColumnLabels

```
plotRowColumnLabels(x[:], y[:], id)
```

By default, the header titles are blank. Any unspecified title is blank.

Calling the function with empty titles removes the titles, if any.

For an example, see “Setting header titles in 2D layout plot windows” on page 697.

plotScatter

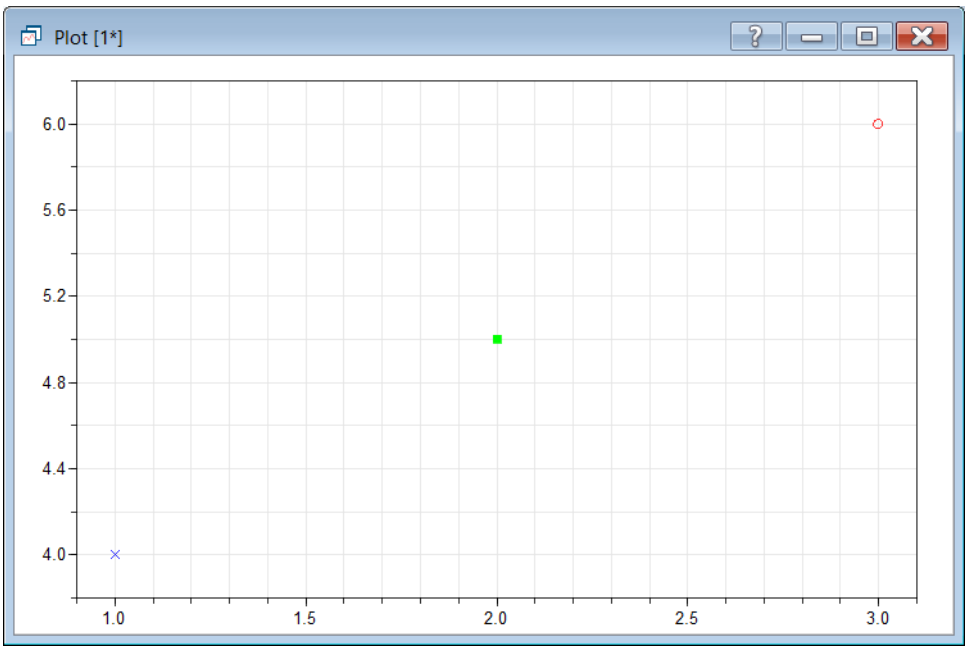
```
plotScatter(x[:], y[size(x,1)], colors[size(x,1), 3],
markers[size(x,1)], legend, axis, unit, erase, id)
```

Each point in the array can have its own color and/or marker style. The default color is the next available one in the subplot. In an empty subplot, the default color is blue. The default marker style is a filled circle.

As an example, the function call:

```
plotScatter(x={1,2,3}, y={4,5,6}, colors={{0,0,255}, {0,255,0},
{255,0,0}}, markers={MarkerStyle.Cross,
MarkerStyle.FilledSquare, MarkerStyle.Circle})
```

will produce the plot:



plotSignalDifference

```
plotSignalDifference(variablePath, startTime, stopTime, axis,
id);
```

The function plots the discrete difference of a signal, defined as y(i)=u(i)-u(i-1).

As an example, the function call corresponding to the signal operator curve in section “Inserting signal operators” starting on page 725, is:

```
plotSignalDifference("dsres[end].J1.w", 0, 1.2, 1, 1);
```

Note that by default the vertical legend axis is to the right.

(You can apply the plotSignalDifference function on for curves with other independent variables than time if the independent variable is rising over time.)

plotSignalOperator

```
plotSignalOperator(variablePath, signalOperator, startTime,
stopTime, id, result)
```

The function plots a signal operator in the active diagram of a plot window. The corresponding GUI handling is described in section "Displaying signal operators" starting on page 722. The following signal operators are presently available:

Signal operators:

```
SignalOperator.Min
SignalOperator.Max
SignalOperator.ArithmeticMean
SignalOperator.RectifiedMean
SignalOperator.RMS
SignalOperator.ACCoupledRMS
SignalOperator.SlewRate
```

Note that First Harmonic and Total Harmonic Distortion are not supported by this function, please see next function.

Concerning handling of FFT and signal operators for such plotted arrays, see example 2 below.

The `id` is the identity of the plot window, where "0" is the last window, -1 the second last etc.

The resulting signal operator is displayed in the plot, and the numerical result is output as `result`.

(You can apply the SignalOperator function also on curves with other independent variables than time (except for FFT) if the independent variable is rising over time.)

Example 1

```
plotSignalOperator("J1.a", SignalOperator.RectifiedMean, 0.8,
1.2, 1);
 = 5.075379430627545
```

Note that variable names with and without result file name and sequence name (in brackets) are supported. See comment on variables in the built-in function `createPlot` above.

Example 2: Handling of signal operators for plotted arrays like FFT (Fast Fourier Transform) in scripting.

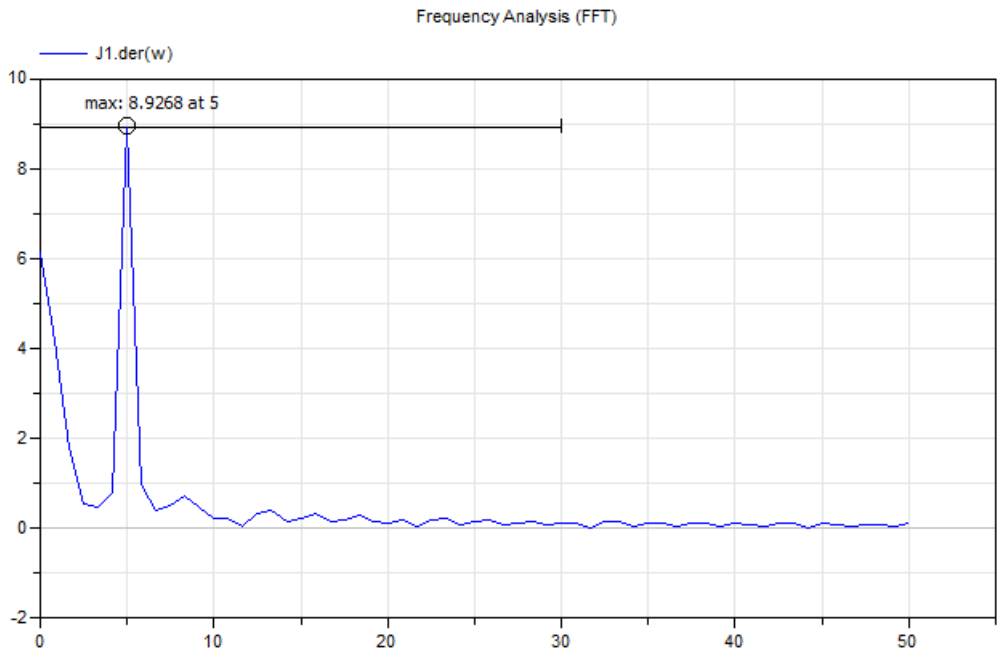
If a legend is specified for a plotted array, this legend can be used by the built-in function `plotSignalOperator` to plot signal operators for the plotted array.

As an example calculating max on an FFT, consider a script like (given that the demo Coupled Clutches is simulated before running the script):

```
createPlot(1, subPlot=2, erase=false, grid=true, leftTitleType=2, leftTitle="Amplitude",
   bottomTitleType=2, bottomTitle="Frequency (Hz)");
(freqPart, powers) = SignalOperators.Fourier.calculateFFT("CoupledClutches.mat",
   {"J1.der(w)"}, 0, 1.2, 501, 50, SignalOperators.Windows.Windowing.Rectangular);
plotArrays(freqPart, powers, legend={"J1.der(w)"}, id=1, title="Frequency Analysis
(FFT)");
plotSignalOperator("J1.der(w)", SignalOperator.Max, 0, 30, 0, 1);
```

(This script is actually created by creating the FFT and signal operator by GUI, and then making a script from the output in the commands window.)

The result is:



plotSignalOperatorHarmonic

```
plotSignalOperatorHarmonic(variablePath, signalOperator,
startTime, stopTime, period, intervalLength, window,
harmonicNo, id, result)
```

The function plots a signal operator in the active diagram of a plot window. The corresponding GUI handling is described in section “Displaying signal operators” starting on page 722.

Note, the package SignalOperators must be present in the package browser to be able to execute this function. The package can be opened by e.g. `import SignalOperators`.

The following signal operators are presently supported for this function:

```
SignalOperator.FirstHarmonic
SignalOperator.THD
```

Compare with previous function that supports other signal operators.

The `window` is the windowing function for FFT, it can be set to any of

```
SignalOperators.Windows.Windowing.Rectangular
SignalOperators.Windows.Windowing.Hamming
SignalOperators.Windows.Windowing.Hann
SignalOperators.Windows.Windowing.FlatTop
```

The `harmonicNo` is the relevant harmonic number.

The `id` is the identity of the plot window, where "0" is the last window, -1 the second last etc.

The resulting signal operator is displayed in the plot, and the numerical result is output as `result`.

Example

```
plotSignalOperatorHarmonic("J1.a",
SignalOperator.FirstHarmonic, 0.8, 1.2, 0.2, 1e-3,
SignalOperators.Windows.Windowing.Rectangular, 1);
 = 7.565026132645894
```

plotText

```
plotText(extent[2,2], textString, fontSize, fontName,
lineColor[3], textStyle[:], horizontalAlignment, id)
```

Insert a text object in the active diagram. The text is rendered using diagram coordinates.

"Null-extent" (both coordinates in the extent being the same) is possible; the text will be centered on the specific point.

If the fontSize attribute is 0 the text is scaled to fit its extents, otherwise the size specifies the absolute size. However, if a minimum font size is set; that size will be the smallest font size. This implies that to create a useful "null-extent" text, the minimum font size should be set. For setting of minimum font size, please see previous chapter, the command **Tools > Options**, the **Graphical Editor** tab, the setting **Restrict minimum font size**.

All installed fonts on the computer are supported.

Available `textStyle` values are (by default none of these are activated)

```
TextStyle.Bold
TextStyle.Italic
TextStyle.UnderLine
```

Available `horizontalAlignment` values are (by default the text is centered)

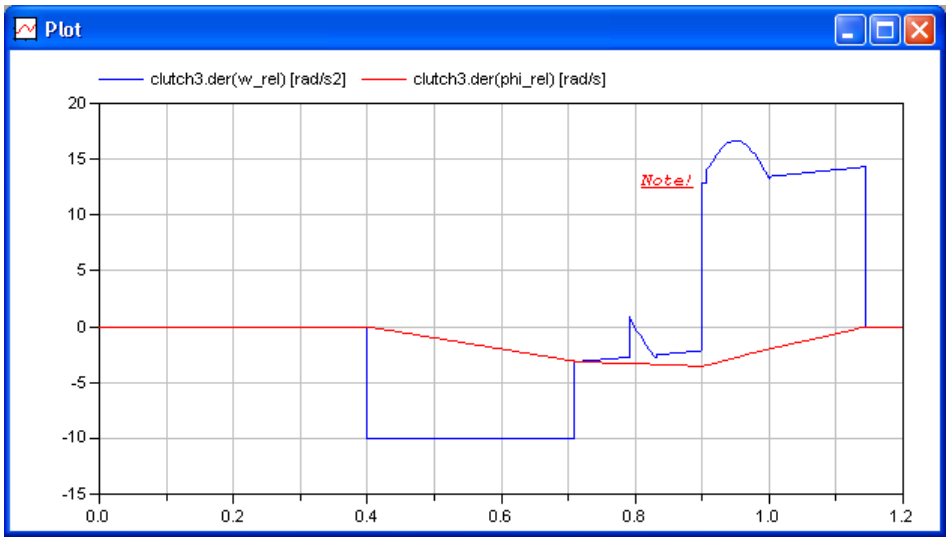
```
TextAlignment.Left
TextAlignment.Center
TextAlignment.Right
```

The text is vertically centered in the extent.

An example:

The text "*Note!*" has been inserted in the below plot using

```
plotText(extent={{0.85,13},{0.85,13}},textString="Note!",
lineColor={255,0,0},textStyle={TextStyle.Italic,TextStyle.
UnderLine},fontName="Courier");
```



The inserted text object corresponds to inserting such object using a command or button. The text can be moved, edited and deleted using interaction, as described in section “Insert and edit text objects in the plot” starting on page 751.

plotTitle

```
plotTitle(title, id)
```

- Add title (text in the title bar) to a plot window. Setting `title` to an empty string removes the title, if any. This feature is also available from the GUI, see “Printing the result.

Note that the argument `fileName` in the built-in function also supports URL´s, in addition to file paths.

Adding titles and documentation to plot windows” on page 748.

printPlot

```
printPlot(y[:], legends[size(y,1)], plotInAll,
colors[size(y,1),3], patterns[size(y,1)], markers[size(y,1)],
thicknesses[size(y,1), ], axes[size(y,1)])
```

Plot the variables and furthermore prints the resulting plot on the default printer. The parameters are the same as in the built-in function plot – see that function.

removePlots

```
removePlots(closeResults=true)
```

The function removes all plot windows. Optionally, if the input parameter `closeResults` is set to `true` (the default value), also all result files are closed. See next command for details about this.

removeResults

```
removeResults(results[:]=fill("",0))
```

Closes specified result files, removing them from the variable browser. If the result input is kept to its default value, all result files are closed. However, the initial values of the current model will still be present, to enable changing these values and do a resimulation of the current model with better parameter settings. If a new model is opened instead, the initial values will also disappear, since they are no longer relevant.

signalOperatorValue

```
signalOperatorValue(variablePath, SignalOperator, startTime=-
1e100, stopTime=1e100)
```

Return the value of a signal operator for a given variable.

Variable names with and without result file name and sequence name (in brackets) are supported. See comment on variables in the built-in function `createPlot` above.

The following signal operators are presently supported:

Signal operators:

```
SignalOperator.Min
SignalOperator.Max
SignalOperator.ArithmeticMean
SignalOperator.RectifiedMean
SignalOperator.RMS
SignalOperator.ACCoupledRMS
SignalOperator.SlewRate
```

Note that First Harmonic and Total Harmonic Distortion are not supported by this function.

`startTime` has default value `-1e100` and `stopTime` has default value `1e100`. If the `startTime` is before the simulated interval, the start of the simulation is used instead, and if the `stopTime` is after the simulated interval, the stop time of the simulation is used instead.

An example of a call:

```
signalOperatorValue("J1.w", SignalOperator.ArithmeticMean)
```

Trajectories

Function	Description
existTrajectoryNames	"Check if provided names exist in trajectory file"
getLastResultFileName	"Get file name of last result file"
interpolateTrajectory	"Interpolates a trajectory on a file"
readTrajectory	"Return all output points of trajectory"
readTrajectoryNames	"Return names in trajectory file"
readTrajectorySize	"Return number of output points of trajectory"
writeTrajectory	"Writes a trajectory file"

A number of examples where some of these functions are used can be found in section “Some examples of Modelica script files” starting on page 933.

Animation

Function	Description
animationSetup	"List current setup of animation window in the Commands window"
exportAnimation	"Export an animation file"
loadAnimation	"Load animation from result file"
RunAnimation	"Run animation"

exportAnimation

```
exportAnimation(path, width=-1, height=-1)
```

Exports an animation to file. Supported file formats are AVI, VRML, and X3D. The file format is determined by the file extension. Use wrl as file extension for VRML.

If there is more than one animation window, the last window is used.

`width` and `height` are only applicable for X3D files.

Matrix IO

For handling of CSV and MAT matrices, please see the package DataFiles. (The package can be accessed by typing the command **import DataFiles** in the command input line.)

Function	Description
readMatrix	"Read a matrix from a file"
readMatrixSize	"Read size of a matrix from a file"
readStringMatrix	"Read a String matrix from a file"
writeMatrix	"Write a matrix to a file"

Documentation

The Documentation group of commands support export of, for example, diagrams and equations, to create dynamic reports. For dynamic reports, please see the chapter “Simulation Environments”.

Function	Description
exportDiagram	"Export the diagram layer to file"
exportDocumentation	"Export the documentation for a model to an HTML file"
exportEquations	"Export the equations to file”
exportIcon	"Export the icon layer to file"
getClassText	"Return the Modelica text for a given model"

exportDiagram

```
exportDiagram(path, width=400, height=400, trim=true,
modelToExport="", evaluate=false)
```

Export the diagram layer to a file. Supported file formats are PNG and SVG. The file format is determined by the file extension. To export in SVG, the diagram layer must exist.

By default, the exported diagram is the one from the active model. However, `modelToExport` can be used to export the diagram of another model; if that is wanted, specify the path to that model using this string parameter.

To evaluate parameter default values before the export, set `evaluate` to `true`.

exportDocumentation

```
exportDocumentation(path, className)
```

Export the documentation for a model to an HTML file.

exportEquations

```
exportEquations(path)
```

Export the equations to file. Supported file formats are PNG and MathML. The file format is determined by the file extension. Use mml as file extension for MathML.

exportIcon

```
exportIcon(path, width=80, height=80, trim=true,
modelToExport="", evaluate=false)
```

Export the icon layer to a file. Supported file formats are PNG and SVG. The file format is determined by the file extension. To export in SVG, the icon layer must exist.

By default, the exported icon is the one from the active model. However, `modelToExport` can be used to export the icon of another model; if that is wanted, specify the path to that model using this string parameter.

To evaluate parameter default values before the export, set `evaluate` to `true`.

getClassText

```
getClassText(fullName, includeAnnotations=false,
formatted=false)
```

Returns the Modelica Text (as a string `classText`) for a given model, and if the model is read-only (`readOnly=true` if so). `fullName` is the name of the model, for example `"Modelica.Mechanics.Rotational.Components.Clutch"`. `includeAnnotations` specify if annotations should be included, `formatted` if the text should be returned as HTML (`formatted=true`) or as plain text.

To set Modelica text for a class, see section “setClassText” on page 947.

Others

Function	Description
DefaultModelicaVersion	"Set default Modelica version"
DymolaVersion	"Return version and date of Dymola"
DymolaVersionNumber	"Return version number of Dymola"
RequestOption	"Test if a certain license option is valid at a certain time"

RequestOption

```
RequestOption(option_name, license_date="")
```

The built-in function tests if a certain license option is valid at a certain time. If so, the output parameter license_valid returns true.

To check whether demo mode is used or not, `RequestOption("Free")` and `RequestOption("Standard")` can be used. The former returns true if in demo mode, and the latter returns true if in standard mode.

5.6 Debugging models

This section contains information about how to solve problems of different kind.

In this section we will assume that the model runs, at least for a short while, but that the results are incorrect. Before the simulation is complete or after a failed simulation one can examine the result by explicitly loading the result file dsres.mat (see “Simulation > Load Result” on page 829), and then plot and/or animate variables. This makes it possible to determine if the result is correct or not.

We will assume that you have followed the guidelines below; in particular, that you have not ignored any warnings and errors detected by the translator, and have tested each sub-model in isolation.

Note the tools for analyzing the model, for example, plotting dependencies and analyzing numeric integration. See section “Plot dependencies” starting on page 633 and “Analyzing numeric integration” starting on page 649.

5.6.1 Guidelines for model development

When constructing models they sometimes generate incorrect results, fail to simulate, fail to start, fail to translate or simply take too long to simulate. The ideal situation would be to avoid this problem altogether. The ideal is not achievable, but some simple rules that reduce the possibility of errors are:

- Use tested libraries, Modelica Standard Library, and available libraries for Hydraulics, PowerTrain, etc.
- Construct test examples for each model in isolation.

- Design models for testing, e.g. introduce variables for quantities that are easy to interpret, use 3D animation for 3D mechanics, and add assert statements for model assumptions.
- Use types and descriptions to document all variables.
- Use min, max to verify range of variables.
- Use assertions to check validity, but do not overdo it since this might imply to heavy constraints on changing the model etc.
- If unsure, verify structural parameters or external data before using them using the possibility to run functions before a component of the model is checked, translated or simulated. Please see the chapter "Advanced Modelica Support", section "Some supported features of the Modelica language", sub-section "Running a function before check/translation/simulation" for more information on this.
- Use standard notation and types.
- Use SI units and use unit checking to detect errors. (See previous chapter, section "Checking the model", sub-section "Unit checking and unit deduction in Dymola".)
- Regularly use the command **Graphics > Check / Text > Check** to check models and components to detect errors; see previous chapter, section "Editor command reference when developing a model", sub-section "Main window: Text tab", command "Text > Check".
- Have connectors of different types, e.g. Rotational, Translational, Electrical, since it allows early detection of mismatched connectors.
- Design packages along the lines of the Modelica Standard Library.
- Never ignore warnings and errors detected by the translator. For more information, please see next section.
- Do not overuse advanced features of Modelica, such as replaceable classes, inner/outer.
- Use any **Order** or **Rename…** command in the context menu for the class in package browser to move the class in package hierarchies. Use **New > Duplicate Class…** in the context menu for the class in package browser to copy a class in package hierarchies, and **File > Save > Save As…** to save to another location on disk. (**Cut/Copy/Paste** will not preserve the links to other classes etc.) It is a good idea to do a **Check** after any of these operations to check for any dangling references.
- Examine the start-values (x_0) and parameters to see that the start values are correct; see section "Setting parameters and initial conditions" on page 616.
- Try to provide good guess values for non-linear iteration variables. (For initial guess values for equations in the model during translation, see "Initial guesses for nonlinear equations in the model during simulation" on page 997.)
- Avoid algorithms in non-functions.
- Beware of graphical lines that look like connections. Connections are created directly by drawing from a connector, see the previous chapter, section "Basic model editing", sub-section "Connections".

- Large result files can be avoided for large models by using variable selections annotations. By default, variables not present in any such selection will not be stored to the result file, resulting in a decrease of the result file. This can considerably decrease the size of the result file, improving performance. Also check that the setting **All variables; ignore selections in model** is not checked in the **Output** tab of the **Simulation > Setup** command if you use this feature. *Note* that in this tab there is also an alternative to minimize the result file if you only need the result file at stop time; you can then tick **Write variables to result file only at stop time**.

Several of these steps are common with normal object-oriented development.

5.6.2 Error messages

Log files and error messages are displayed in the log window (see "Log window" on page 603 for more information).

There are two tabs of interest in the log window when it comes to simulaton of a model, the **Translation** tab and the **Simulation** tab.

More about information in the Translation tab of the log window

The translation tab of the log window contains the translation log. For a general description of the translation tab when translating a model without failure, see "Translation tab" starting on page 604.

More interesting here is the description of warnings and errors in the translation tab, see "Errors and warnings when translating" starting on page 753.

Error messages for type errors are grouped together, and start by giving the variable or equation that caused the error. The error messages contain links to the text of classes.

Assertions are evaluated early to improve diagnostics, and thus allow an assertion to guard against a structural error.

The translation log also includes statistics for the initialization problem.

Concerning the diagnostics for nonlinear system of equations (see also next section that deals more with this matter):

- Warnings are listed for iteration variables to be solved by a nonlinear solver, but not having an explicit start value.
- The start values for all iteration variables can be logged by setting the flag `Advanced.LogStartValuesForIterationVariables = true`
- If sparse solvers can be used, this is indicated in the Translation tab of the log window. See "Sparse solvers" on page 900.
- If additional information for failed differentiation of Jabobians may be displayed, this is indicated in the log window. See "Diagnostic message for failed differentiation of Jacobians" on page 1011.

More about information in the Simulation tab of the log window

The Simulation tab of the log window contains the simulation log. For a general description of the Simulation tab, see “Simulation tab” on page 608.

If DAE mode is used, information about this is included in the Simulation tab. See “DAE mode” on page 899.

Concerning the diagnostics for nonlinear systems of equations, note that the amount of information in the simulation tab can be controlled by a number of settings; see next section.

In some cases, direct links are available in the Simulation tab to variables in the model window, see “Direct link in the error log to variables in model window” on page 982.

5.6.3 Controlling the amount of diagnostics for nonlinear solvers in the simulation log

For the simulation log, displayed in the **Simulation** tab of the log window, a number of settings can control the amount of information given for nonlinear solvers after a simulation. The settings are available in the simulation setup, reached by the command **Simulation > Setup**, in the **Debug** tab. The tab is described in section “Debug tab” on page 812.

The settings are located under the **Nonlinear solver diagnostics** group. By default, they are all deselected.

In general, the first setting is specific for logging of failed nonlinear solutions. For the rest, except for the last setting, the idea is to start enabling the setting **Nonlinear soluton summary**, if this one does not give sufficient information, also enable the next setting, and so on. Notes:

- For the nonlinear solution summary, you can also plot these variables by activating an additional settting, see below.
- The simulation time and size of log file can be considerably larger by enabling some settings.

Detailed logging of failed nonlinear solutions

This setting is available from Dymola 2021x. If you activate this setting, you get the same detailed information about failed nonlinear solutions as in previous versions.

The setting is by default not activated, corresponding to the flag `Advanced.Debug.LogNonlinearFailedSolutions=false`. The setting is saved between sessions.

An example of the first part of a simulation log in Dymola 2021x with the default (not activated) setting:

```
Log-file of program ./dymosim
(generated: Wed Aug 26 10:49:02 2020)

dymosim started
... "NonlinearEquationSolve" simulating
... "dsin.txt" loading (dymosim input file)
... "NonlinearEquationSolve.mat" creating (simulation result file)

Integration started at T = 0 using integration method DASSL
(DAE multi-step solver (dassl/dasslrt of Petzold modified by Dassault Systemes))

Warning: Failed to solve nonlinear system using Newton solver.
  Time: 0.4996100837250252
  Tag: simulation.nonlinear[1]
  For debugging information enable
  Simulation/Setup/Debug/Nonlinear solver diagnostics/Detailed logging of failed nonlinear solutions.
Solver will attempt to handle this problem.

Warning: Failed to solve nonlinear system using Newton solver.
  Time: 0.4939881859511698
  Tag: simulation.nonlinear[1]
Solver will attempt to handle this problem.


Warning: Failed to solve nonlinear system using Newton solver.
  Time: 0.4911772370642422
  Tag: simulation.nonlinear[1]
Solver will attempt to handle this problem.
```

If the setting **Detailed logging of failed nonlinear solutions** is instead activated, the framed message above will instead be:

```
Warning: Failed to solve nonlinear system using Newton solver.
  Time: 0.4996100837250252
  Tag: simulation.nonlinear[1]

  Common causes:
   * The system of equations has no solution - the residual will be above zero.
     - In some cases the event-logic can cause this.
   * Starting values are too far from the solution.
     - In rare cases this could occur at events.
   * The equations are too discontinuous for the nonlinear solver - the residual will have knees.
     - Likely caused by over-using noEvent.

  To get more information consider the options:
   * Simulation/Setup/Translation/Generate listing of translated Modelica code in dsmodel.mof
   * Simulation/Setup/Translation/List non-linear iteration variables
   * The options under the group Simulation/Setup/Debug/Nonlinear solver diagnostics

  Jacobian inverse norm estimate: 516840
  Condition number estimate: 213378
  1-norm of the residual = 0.0133969

  Last value of the solution:
    der(z) = -0.358241
    der(x) = -1.09532
  Last value of the residual:
    { -0.00363467, -0.00976219 }

Solver will attempt to handle this problem.
```

Notes:

- From Dymola 2021x, messages about failed nonlinear solutions are displayed as warnings; previously they were displayed as errors.
- The log of the first failed attempt of solving nonlinear equations always contain tips on how to display more information.

Nonlinear solution summary

Enabling this setting, a nonlinear solver summary is included in the end of the simulation log, for example:

```
Nonlinear solver summary, accumulated amounts:
 Tag                          ,  Calls, Residues, Iterations, Jacobians
 initialization.nonlinear[1]:      1,        6,          6,         1
 simulation.nonlinear[1]    : 429665,   860615,     860615,    429913
 simulation.nonlinear[2]    :  99195,   199130,     199130,     99325
To also log used CPU-times enable Advanced.GenerateBlockTimers.
```

If the above setting is active, the nonlinear solution summary is also included in the message given in the simulation log in the following cases:

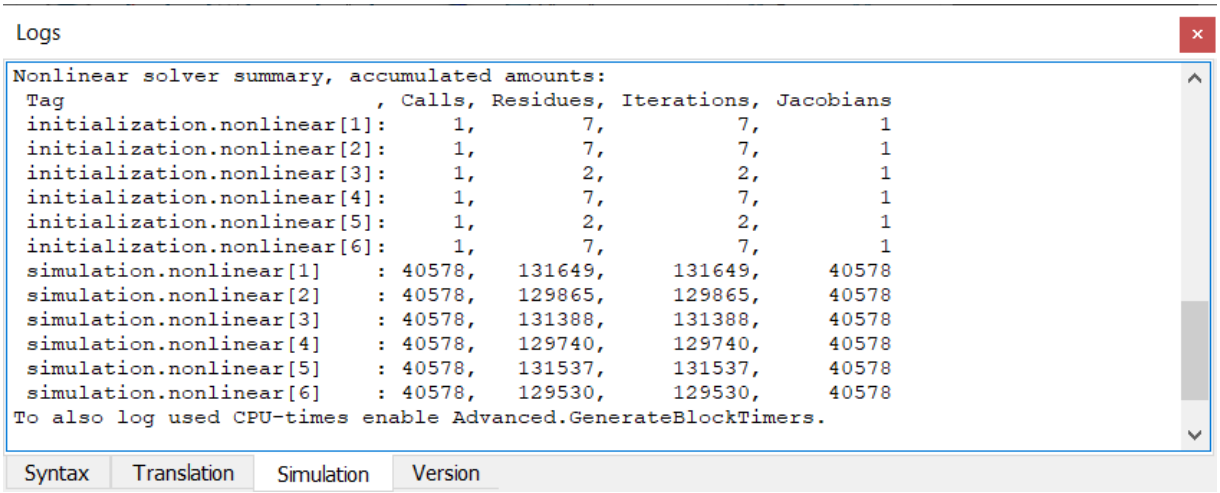
- When solving a nonlinear equation fails
- When initialization fails

- When using profiling – note the tip in the message about setting the flag `Advanced.GenerateBlockTimers` to `true` to activate profiling to log CPU times.

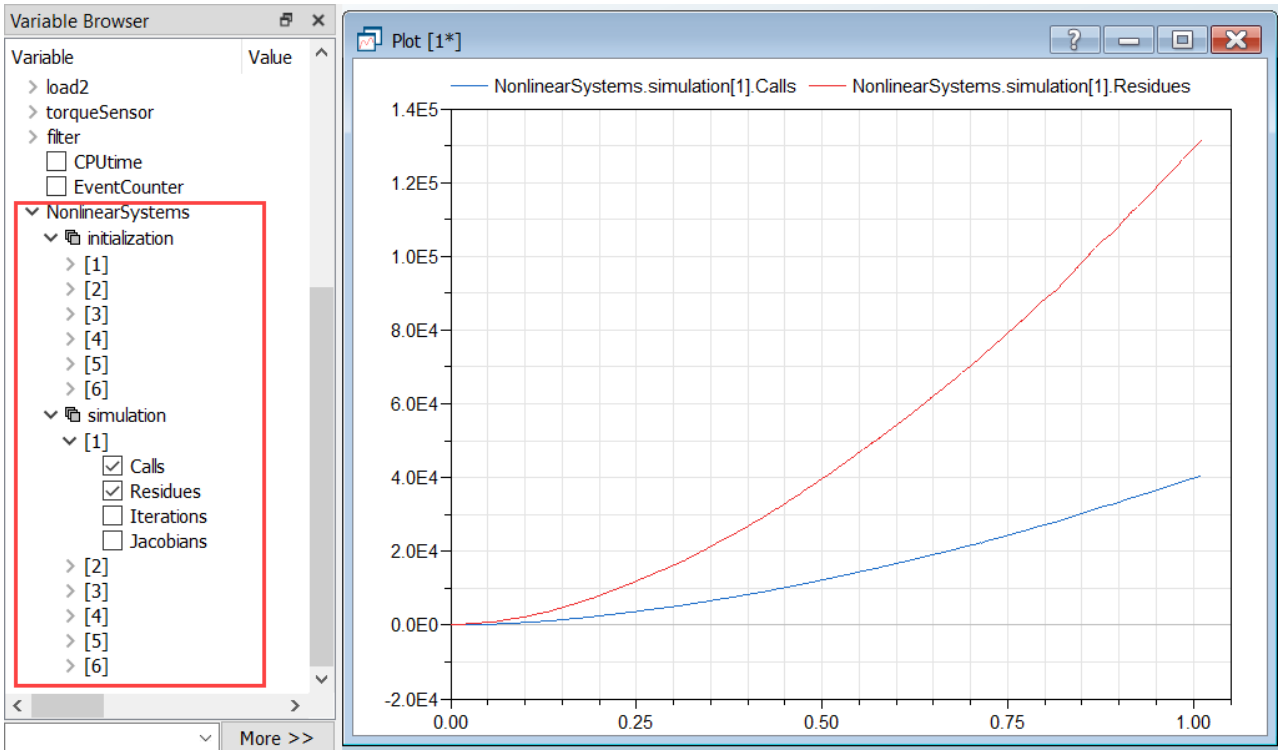
If you activate the setting **Include variables for CPU time, event counting, and nonlinear solver statistics** in the **Debug** tab of the simulation setup, you can also plot the variables in the statistics to see where the most effort is spent. For more information about the setting, see section “Debug tab” starting on page 812.

For the example Modelica.Mechanics.MultiBody.Examples.Loops.EngineV6:

The nonlinear solution summary in the log:



The corresponding variables in the variable browser, with two signals plotted:



We can see from the plot that the equation is called more often as the simulation proceeds; the number of residual calculations seems to be explained by the increased number of calls to the equation.

Nonlinear solutions

Enabling this setting, the solutions of nonlinear systems are displayed in the log. Note that the log can be large if this setting is activated, if the resulting simulation log file `dslog.txt` is larger than 10 MB, a message is displayed with the path to the file, and a recommendation to open this file outside Dymola. You can do this with a simple text file progam, for example, Notepad.

A small extract of the file in the case Dymola fails to solve a nonlinear system of equations during simulation might look like:

```
Log-file of program ./dymosim
(generated: Thu Mar 29 15:17:22 2018)

dymosim started
... "dsin.txt" loading (dymosim input file)

Solving nonlinear system of equations number = 1
Tag: initialization.nonlinear[1]
        iteratively at Time = 0
  Solution: Infinity-norm of residue = 2.69063E-13
```

```
            Acc. number of residue     calc.: 6
            Acc. number of symbolic Jacobian calc.: 1
der(x) = -1.10386
der(z) = -0.815554
Residual:
{ 2.21101E-13, 4.79616E-14 }
```

The tag in the message above facilitates to find the corresponding system in the file containing the C code, `dsmodel.c`, or, if flat Modelica is generated, in the file `dsmodel.mof`.

Nonlinear iterations

Enabling this setting, iterations of nonlinear systems are displayed in the simulation log.

Details

If this setting is enabled, the simulation log will contain details about the iterations of nonlinear systems.

Log singular linear

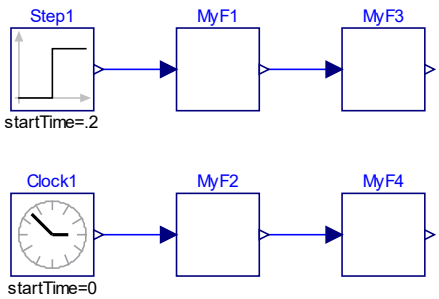
If this setting is enabled, the simulation log will contain information if linear systems are consistently over-determined.

5.6.4 Direct link in the error log to variables in model window

When an error occurs in a component or function call it is important to know in which instance of the function that caused the problem. In Dymola a direct link in the error log to variables in model window can be used to trace such problems. We will present an example and then look at how this example can be diagnosed.

A model example

Consider the following simple model.



Each of the blocks contain the same function call, and in case one of them fails it is important to understand which one. This problem can be even more pronounced in larger models.

```
model MyF
```

```
  extends Modelica.Blocks.Interfaces.SISO;
  function f
    input Real x;
    output Real y;
  algorithm
    assert(x<2, "Cannot be larger than 2");
    y:=x*x/(1+x);
  end f;
  Real x;
equation
  der(x)=u;
  y=f(x);
  annotation (uses(Modelica(version="2.2")));
end MyF;

model WhichOne
  import Modelica.Blocks.Sources;
  MyF MyF1 annotation (extent=[-20,20; 0,40]);
  annotation (Diagram, uses(Modelica(version="2.2")));
  Sources.Step Step1(startTime=.2, height=4)
    annotation (extent=[-60,20; -40,40]);
  Sources.Clock Clock1 annotation (extent=[-60,-20; -40,0]);
  MyF MyF2 annotation (extent=[-20,-20; 0,0]);
  MyF MyF3 annotation (extent=[20,20; 40,40]);
  MyF MyF4 annotation (extent=[20,-20; 40,0]);
equation
  connect(Step1.y, MyF1.u)
    annotation (points=[-39,30; -22,30]);
  connect(Clock1.y, MyF2.u)
    annotation (points=[-39,-10; -22,-10]);
  connect(MyF2.y, MyF4.u)
    annotation (points=[1,-10; 18,-10]);
  connect(MyF1.y, MyF3.u)
    annotation (points=[1,30; 18,30]);
end WhichOne;
```

This model fails to simulate because of the assertion in f, but which of MyF1.f, MyF2.f, MyF3.f and MyF4.f is violating the assertion?

The corresponding error in the simulation log

The log message in the **Simulation** tab of the log window contains

```
Assertion failed: x < 2
The following error was detected at time: 0.7000000000000001
Cannot be larger than 2
The stack of functions is:
MyF.f
MyF.f(MyF1.x)
Integration terminated before reaching "StopTime" at T = 0.7
```

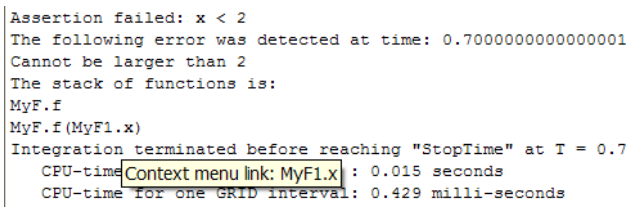
This clearly shows MyF1.f caused the error.

Note: Due to alias elimination, the variable in some cases will not be the exact same variable, but can be an identical variable in the same or a connected sub-system.

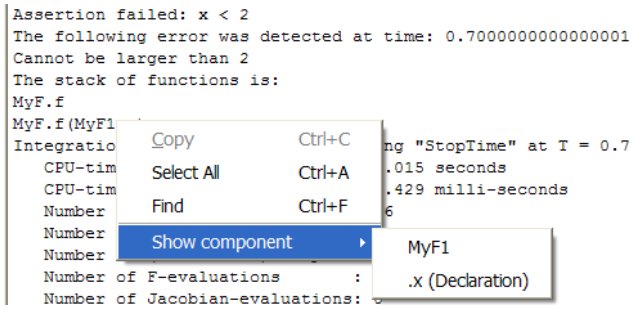
This is active as default but it is possible to de-activate this additional diagnostics, since it adds to the size of the generated C code and is only interesting if the model fails in some function.

The direct link in the error log to the variables in the model window

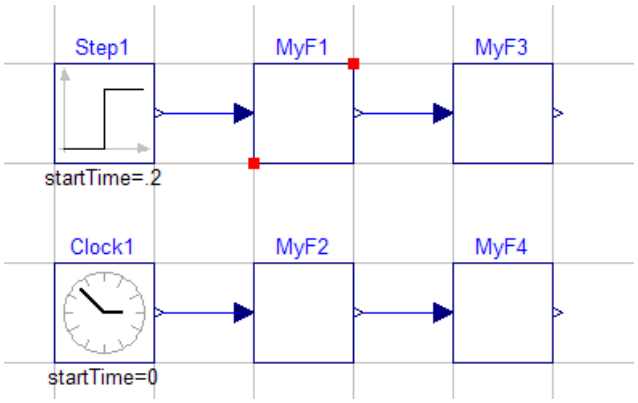
When looking at the error message for the WhichOne model in the simulation log there is a tooltip for the component as follows:



Right-clicking on MyF1.x brings up the context menu:



Selecting MyF1 in this menu highlights the MyF1 component in the diagram:



The intention of highlighting the component is that an error can be due to the component itself, the parameters, or the interaction with connected components. By highlighting the component, it is easy to investigate these.

Clicking on the last part (.x) brings up the text layer of the component, and searches for the declaration of 'x'.

The function part of the function call has a similar link to the function.

5.6.5 Saving periodic snapshots during simulation

To improve the possibility to restart a lengthy simulation form multiple time points, periodic snapshots can be saved during simulation. The reasons for such restarts could be some problems in the simulation that needs debugging, or external needs such as restart after each subtask in a training session.

The GUI for the option is available in the simulation setup, reached by the command **Simulation > Setup**, the **Output** tab, the **Complete result snapshots** group. Here the interval between printing of the simulation result file "dsfinal.txt" can be set, as well as if a timestamp should be added to that file. Please see section "Output tab" on page 809 for details.

The generated result snapshot can then be used in combination with the command **Simulation > Continue > Import Initial…** (or the function `importInitial`) to continue from the time of the chosen snapshot. For example, to perform parameter studies from that point.

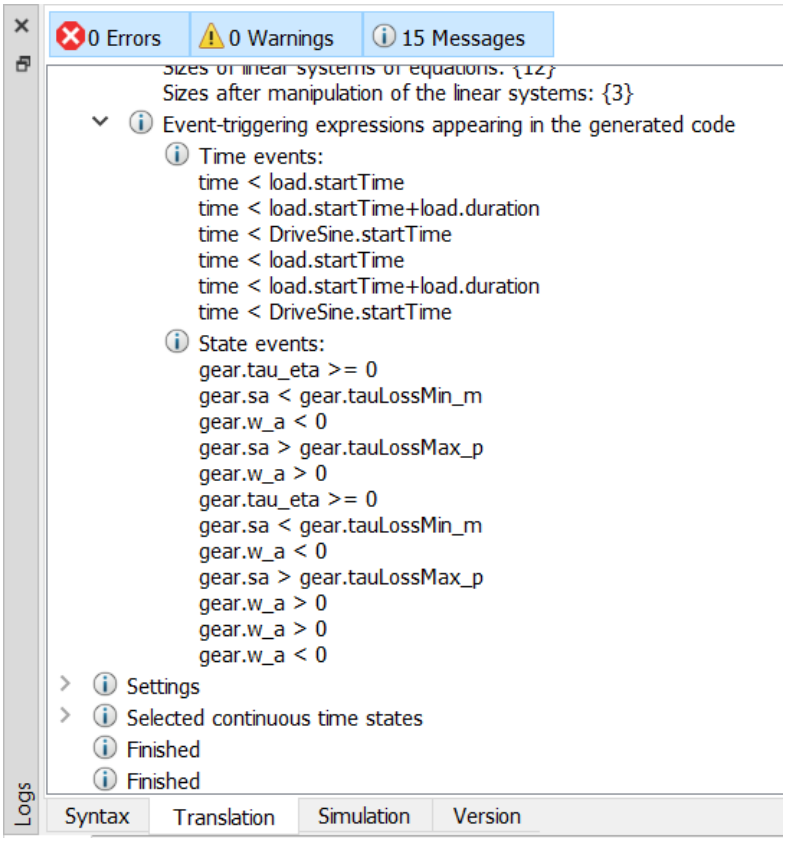
It is also possible to trigger the generation of snapshots at events. Calling the built-in function `Dymola.Simulation.TriggerResultSnapshot ()` at an event triggers the creation of a result snapshot after the event is fully resolved. Example:

```
when x > 0 then
       Dymola.Simulation.TriggerResultSnapshot();
end when;
```

5.6.6 Event logging

In order to determine if there is a problem with the discrete events one can turn on logging of events during a simulation. By activating the setting **Events during initialization and simulation** in the simulaton setup one will see a log where each expression causing an event is logged, see "Debug tab" starting on page 812. This makes it possible to track down errors in the discrete part of a hybrid system. Notes:

- What events are logged can be fine-tuned using flags. See see "Debug tab" starting on page 812.
- There is a GUI available to display the result of the event logging, see "Event logging user interface" starting on page 654.
- You can list all expressions that may trigger events by activating the setting **List event-triggering expressions** in the **Translation** tab of the simulation setup, see "Translation tab" starting on page 805. The expressions are listed in the translation log, under "Statistics". An example of the translation log from simulating `Modelica.Mechanics.Rotational.Examples.LossyGearDemo1` with this setting activated:



Note that the list in the translation statistics does not include input time events and dynamic state selection events.

Note also that the listed expressons only potentially may trigger events. It may be that only a subset actually triggers events during simulation. To check which event-triggering expressions actually did cause events during a particular simulation, consider the **Event Log** tab in the simulation analysis GUI; see the note about GUI above.

5.6.7 Model instability

The result file can be loaded even if the simulation fails.

Instabilities often cause the simulation to stop with unphysical values after some simulated time. In this case, it is advisable to load the simulation result even if the simulation failed, and also to store all simulated points by de-activating “Equidistant time grid”, see “Output tab” on page 809, since the integrator will store its steps and it generally uses more steps in problematic regions.

The first task is to determine which variable/subsystem becomes unstable first, since instability in one part can then spread to the entire system.

In almost all cases, instabilities are found among the states of the system and can be more easily found by only storing the states of the system and plotting them. This can be done using

the command **Simulation > Setup**, selecting the **Output** tab and then unchecking all entries in the **Store** group except **State variables**. Please see section "Output tab" on page 809 for more information.

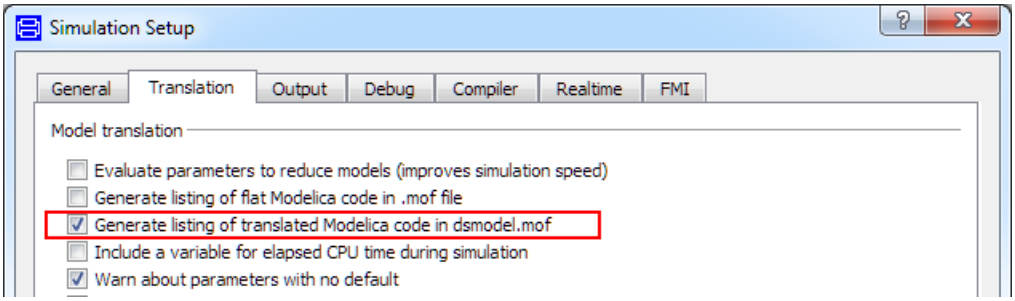
It is also possible to linearize around various operating points in order to determine the stability of the linearized system. A useful command is **Simulation > Linearize**. Please see page 827. By examining the eigenvalues, it is in general possible to track down the instability to some state.

Thus, the problem is localized to one part of the model, and the next step is to correct that sub-model. In many cases, the cause of the problem is a simple sign-error in one equation, either in a user-written equation or in a connection equation (due to not using the flow attribute correctly in the connectors of that model).

5.6.8 Output of manipulated equations in Modelica format

Description

The result of translating a Modelica model can be listed in a Modelica like representation. The listing is stored in the file dsmodel.mof and is meant to be a more readable version of dsmodel.c. (For more information about dsmodel.c, see the chapter "Simulation Environments", section "Code and Model Export".) The listing is enabled by selecting **Simulation > Setup** and then in the **Translaton** tab, ticking **Generate listing of translated Modelica Code in dsmodel.mof**.



The listing may be useful for users who want to investigate algebraic loops or for other debugging purposes. It gives the correct computational structure including algebraic loops. However, to make it more readable some optimization steps such as elimination of common sub expressions are not done.

It means that Jacobians of algebraic loops are by default not listed, because without common sub expression elimination those expressions may be very long. Listing of the non-zero Jacobian elements may be enabled by issuing the command

```
Advanced.OutputModelicaCodeWithJacobians = true.
```

Information on this is included in the listing if there are algebraic loops.

Listing of eliminated alias variables may also be long. Thus, the listing of these variables is not enabled by default. The listing of alias variables is enabled by setting the flag

```
Advanced.OutputModelicaCodeWithAliasVariables = true
```

Information on this is also included at the end of dsmodel.mof, if listing of alias variables is not enabled.

Note that you can specify how to present alias variables in the file by a flag, see section "Handling of alias names of variables in error messages and other outputs" on page 757.

Below some examples are given for illustration.

Examples

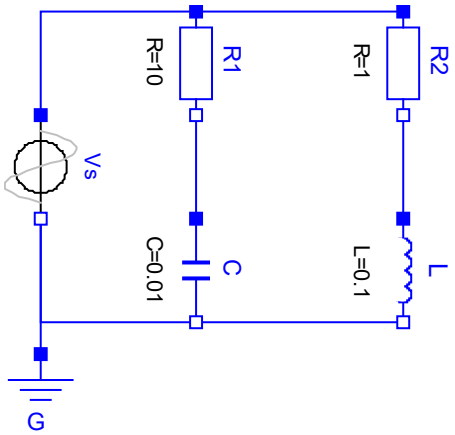
Below, some examples will be given to illustrate and to discuss the information given by dsmodel.mof.

- **Simple LC circuit** is a first example to discuss the organization of the manipulated equations and listing of alias variables
- **Two resistors connected in series** illustrates symbolic solution of a linear algebraic loop
- **A simple resistor network** illustrates a manipulated linear system for numeric solution.
- **Diode circuit with two valued resistance diode model** introduces mixed discrete/real algebraic loops
- **Diode circuit with diode exponential diode model** introduces nonlinear algebraic loops

Simple LC circuit

Consider a simple electric LC circuit with two resistors connected in series.

Simple LC circuit.



Translating the model produces a dsmodel.mof file with following contents.

```
// Translated Modelica model generated by Dymola from //
Modelica model
//  OutputModelicaCodeExamples.SimpleLC_Circuit
// ------------------------------------------------
// Initial Section
  Vs.signalSource.pi := 3.14159265358979;
  C.n.v := 0;
  G.p.v := 0;
  L.n.v := 0;
  Vs.n.v := 0;
// ------------------------------------------------
// Bound Parameter Section
  Vs.signalSource.amplitude := Vs.V;
  Vs.signalSource.freqHz := Vs.freqHz;
  Vs.signalSource.phase := Vs.phase;
  Vs.signalSource.offset := Vs.offset;
  Vs.signalSource.startTime := Vs.startTime;
// ------------------------------------------------
// Dynamics Section
  R2.p.v := Vs.signalSource.offset+
         (if time < Vs.signalSource.startTime then 0
          else Vs.signalSource.amplitude*
               sin(6.28318530717959*
                    Vs.signalSource.freqHz*
                   (time-Vs.signalSource.startTime)
               +Vs.signalSource.phase));
  R1.v := R2.p.v-C.v;

  // Linear system of equations
    // Symbolic solution
      /*  Original equation
```

```
      R1.R*C.n.i =  -R1.v;
      */
      C.n.i :=  -R1.v/R1.R;
    // Torn part
  // End of linear system of equations

  // Linear system of equations
    // Symbolic solution
      /*  Original equation
      C.C*der(C.v) =  -C.n.i;
      */
      der(C.v) :=  -C.n.i/C.C;
    // Torn part
  // End of linear system of equations
  R2.v := R2.R*L.i;
  L.v := R2.p.v-R2.v;

  // Linear system of equations
    // Symbolic solution
      /*  Original equation
      L.L*der(L.i) = L.v;
      */
      der(L.i) := L.v/L.L;
    // Torn part
  // End of linear system of equations
// -------------------------------------------------
// Conditionally Accepted Section
  Vs.p.i := C.n.i-L.i;
  G.p.i := Vs.p.i-C.n.i+L.i;
// -------------------------------------------------
// Eliminated alias variables
  // To have eliminated alias variables listed, set
  //   Advanced.OutputModelicaCodeWithAliasVariables
  //  = true
  // before translation. May give much output.
```

The manipulated equations are sorted into sections. First, there are calculations of constants and bound parameters. These parts are only executed at initialization. Then the calculation of outputs and the derivatives of the continuous time states follow. The output and dynamics sections are executed during continuous integration. In the general case there is then a section titled Accepted Section. It includes codes for detecting discrete event and updating discrete states that need not be evaluated at continuous integration. This section is executed at the end of each step to check for events. The output, dynamics and accepted sections are executed at event propagation also. Finally, there is the Conditionally Accepted Section. It includes calculation of variables which are not necessary to know when calculating derivatives or updating discrete states. This section is executed when storing values. In some situations for example when simulating on a HIL platforms where only states and outputs may be visible, the conditionally accepted section is not executed at all.

As shown by the listing, Dymola converts this problem symbolically to explicit ODE form (no algebraic loops to solve numerically). We can observe that Ohm's law of the resistor R1 is used to solve for the current through the resistor:

```
C.n.i :=  -R1.v/R1.R;
```

On the other hand, Ohm's law of the resistor R2 is used to solve for the voltage drop across the resistor :

```
R2.v := R2.R*L.i;
```

The sorting procedure of Dymola automatically finds which variable to solve for from each equation.

Connections between non-flow connectors result in simple equations, v1=v2. A connection between two flow connectors gives v1+v2=0. Dymola exploits such simple equations to eliminate variables. The listing of these variables is not enabled by default as indicated above because it may give much output. If we enable the listing of alias variables by setting the flag

```
Advanced.OutputModelicaCodeWithAliasVariables = true
```

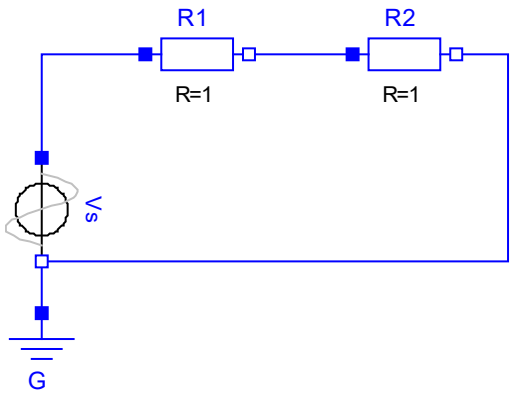
the last part of the listing becomes

```
// Eliminated alias variables
  R2.p.i = L.i;
  R1.i =  -C.n.i;
  R2.n.v = L.v;
  R1.p.i =  -C.n.i;
  R2.n.i =  -L.i;
  L.p.v = L.v;
  Vs.n.i =  -Vs.p.i;
  R1.n.i = C.n.i;
  C.i =  -C.n.i;
  R2.i = L.i;
  R1.n.v = C.v;
  R1.p.v = R2.p.v;
  Vs.v = R2.p.v;
  C.p.v = C.v;
  Vs.i = Vs.p.i;
  C.p.i =  -C.n.i;
  L.p.i = L.i;
  Vs.p.v = R2.p.v;
  Vs.signalSource.y = R2.p.v;
  L.n.i =  -L.i;
```

Two resistors connected in series

Consider a simple electric circuit with two resistors connected in series.

Two resistors connected in series.



After elimination of alias variables, the problem has a linear algebraic loop with three unknowns. Dymola solves this symbolically as seen from the following excerpt from the dsmodel.mof .

```
// Linear system of equations
  // Symbolic solution
    /*  Original equation
    R1.R*R1.i = R1.v;
    */
    R1.i := Vs.v/(R1.R+R2.R);
  // Torn part
    R2.v := R2.R*R1.i;
    R1.v := Vs.v-R2.v;
// End of linear system of equations
```

The equation

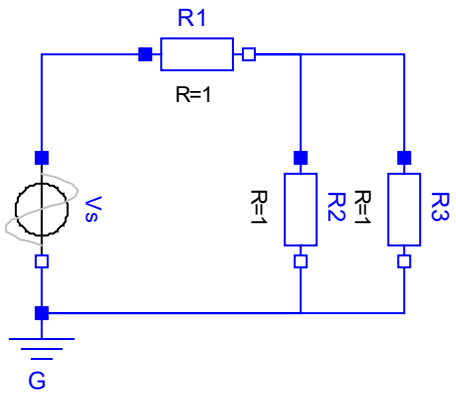
```
R1.i := Vs.v/(R1.R+R2.R);
```

reveals that the resistance of two resistors connected in series is the sum of the resistances of the two resistors. Dymola “discovered” this law automatically.

A simple resistor network

Consider a simple resistor network.

A simple resistor network.



After elimination of alias variables, the problem has a linear algebraic loop with five unknowns. Dymola reduces it to a linear problem with two unknowns as seen from the following excerpt from the dsmodel.mof.

```
// Linear system of equations
  // Matrix solution:
    /* Original equations:
    R1.R*R1.n.i =  -R1.v;
    R2.R*R2.p.i = R3.v;
    */
    // Calculation of the J matrix and the b vector,
    // but these calculations are not listed here.
    // To have them listed, set
    //   Advanced.OutputModelicaCodeWithJacobians =
    //    true
    // before translation. May give much output,
    // because common subexpression elimination is
    // not activated.
    x := Solve(J, b); // J*x = b
    {R3.p.i, R2.p.i} := x;
  // Torn part
    R1.n.i :=  -(R2.p.i+R3.p.i);
    R3.v := R3.R*R3.p.i;
    R1.v := Vs.v-R3.v;
// End of linear system of equations
```

To make the listing more readable, some optimization steps such as elimination of common sub expressions are not done. It means that Jacobians of algebraic loops are by default not listed, because without common sub expression elimination those expressions may be very long. As described above the listing of the non-zero Jacobian elements is be enabled by issuing the command

```
Advanced.OutputModelicaCodeWithJacobians = true
```

The manipulated linear system is now output.

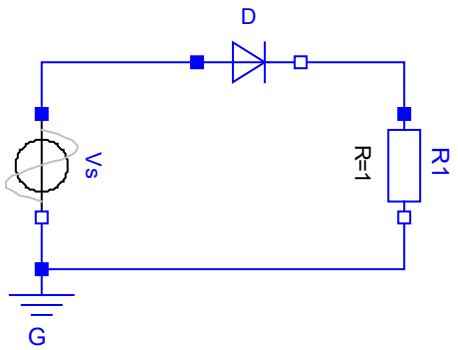
```
// Linear system of equations
  // Matrix solution:
    /* Original equations:
    R1.R*R1.n.i =  -R1.v;
    R2.R*R2.p.i = R3.v;
    */
    J[1, 1] :=  -(R1.R+R3.R);
    J[1, 2] :=  -R1.R;
    J[2, 1] :=  -R3.R;
    J[2, 2] := R2.R;
    b[1] :=  -Vs.v;
    x := Solve(J, b); // J*x = b
    {R3.p.i, R2.p.i} := x;
  // Torn part
    R1.n.i :=  -(R2.p.i+R3.p.i);
    R3.v := R3.R*R3.p.i;
    R1.v := Vs.v-R3.v;
// End of linear system of equations
```

Please, note that all element of the J matrix are non-literal expressions. Elimination of variables must not introduce divisions by zero. If we would like to use the first remaining equation to solve for the first unknown, R3.p.i, we need to divide by J[1, 1] := -(R1.R+R3.R). Since it cannot be guaranteed that this expression always is non-zero, it is not a good idea to use this equation to eliminate R3.p.i. Thus to use an equation to eliminate a variable safely, its coefficient must be a non-zero numeric value. Since the Jacobian above has no numeric elements, it is not possible to eliminate variables further. We need to invert the matrix. It is indeed possible to do that for a two by two matrix and Dymola does it in some situations when generating simulation code for real-time and HIL simulation. However, in normal cases Dymola generates code for numeric solution, because it is allows better support of singular systems.

Diode circuit with two valued resistance diode model

Consider a simple electric circuit with a diode and a resistor connected in series.

Diode circuit with two valued resistance diode model.



Let the diode be modeled by the model

```
Modelica.Electrical.Analog.Ideal.IdealDiode.
```

It models the diode characteristic as a conductance in off mode and a resistance in leading mode. Thus in each of the two modes the problem is linear. There is an algebraic loop with five unknowns of which one of them, namely D.off, is a Boolean variable. The algebraic loop is a mixed system with one Boolean equation and four real equations.

```
// Mixed system of equations
  // Linear system of equations
    // Symbolic solution
      /*  Original equation
      D.v = Vs.v-R1.v;
      */
      D.s := (Vs.v-(R1.R*D.Goff*D.Vknee+D.Vknee)) /
             (R1.R*(if D.off then D.Goff else 1)
             +(if D.off then 1 else D.Ron));

    // Torn part
      D.i := D.s*(if D.off then D.Goff else 1)
             + D.Goff*D.Vknee;
      R1.v := R1.R*D.i;
      D.v := D.s*(if D.off then 1 else D.Ron)
             + D.Vknee;
  // End of linear system of equations

  // Torn discrete part
    D.off := D.s < 0;
// End of mixed system of equations
```

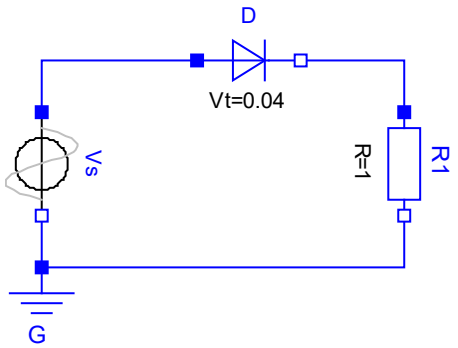
Dymola solves such a system by iterating. Assuming the Boolean variable to be known and removing the Boolean equation, the rest is a linear problem with four real unknowns. As seen, Dymola solves the linear part symbolically. During simulation, Dymola iterates, if the assignment D.off := D.s < 0 changes the value of D.off.

Diode circuit with exponential diode model

Let us revise the diode circuit above to use the diode model

```
Modelica.Electrical.Analog.SemiConductors.Diode
```

Diode circuit with exponential diode model.



Its diode characteristic is nonlinear, with exponential terms. There is a nonlinear algebraic loop with three unknowns.

```
// Mixed system of equations
    // Nonlinear system of equations
      // It depends on the following parameters:
      //   D.Ids
      //   D.Maxexp
      //   D.R
      //   D.Vt
      //   R1.R
      // It depends on the following timevarying
      // variables:
      //   Vs.v
      //   discreteexpr_0.
      // Unknowns:
      //   R1.v(start = 0)
    algorithm // Torn part
      D.v := Vs.v-R1.v;
      D.i := (if discreteexpr_0.then
                D.Ids*(exp(D.Maxexp)*
                       (1+D.v/D.Vt-D.Maxexp)-1)
                +D.v/D.R
              else D.Ids*(exp(D.v/D.Vt)-1)+D.v/D.R);
    equation // Residual equations
      0 = R1.R*D.i-R1.v;
      // Non-zero elements of Jacobian
        J[1, 1] := (-1)-
          (if discreteexpr_0.then
              D.Ids*exp(D.Maxexp)/D.Vt+1/D.R
           else D.Ids*exp(D.v/D.Vt)/D.Vt+1/D.R)*R1.R;
    // End of nonlinear system of equations
    // Torn discrete part
      discreteexpr_0.:= D.v/D.Vt > D.Maxexp;
// End of mixed system of equations
```

Dymola reduces it to a nonlinear problem with one iteration variable, namely

```
R1.v(start = 0)
```

The start value is included in the listing, because the nonlinear solver will use it. The diode model includes also an if-then-else expressions, where the condition, D.v/D.Vt > D.Maxexp, refer to one unknown, D.v, of the algebraic loop. Dymola introduces an auxiliary variable named "discreteexpr_0." for the condition. This transformation removes a possible discontinuity from the real part of the problem.

5.6.9 Initial guesses for nonlinear equations in the model during simulation

A translated Modelica model may contain nonlinear equations which are solved several times during integration. To enable fast and robust solution Dymola integrators provide initial guesses for these nonlinear equations. The guesses are dynamically updated during integration to incorporate the most recent values of the iteration variables of the nonlinear equation system.

The user can control the computation method for initial guesses, in particular for nonlinear equations in the accepted sections (`Accepted Section` and `Conditionally Accepted Section`) of the model equations.

(To display the output of manipulated equations in readable form to see the different equations sections, you must generate the output of manipulated equations in Modelica format and investigate the file `dsmodel.mof`. For more information about this, and examples, see previous section.)

The inter-/extrapolation method to be used to compute initial guesses can be controlled by using the flag

```
Advanced.Define.InitialGuessInterpolation
```

The flag can have any of the following values:

- 0 results in constant inter-/extrapolation in all sections
- 1 (default) meaning linear inter-/extrapolation is being used to define initial guesses in all sections except accepted sections
- 2 results in linear inter-/extrapolation only in accepted sections
- 3 results in linear inter-/extrapolation being used to define initial guesses in all sections

Linear extrapolation typically performs better if the iteration variables vary smoothly as functions of time, otherwise constant extrapolation may perform better.

Note

Note that only some solvers support linear inter/extrapolation. The solvers supporting it are Dassl, Radau IIa, Esdirk*, Dopri*, Sdirk34hw, and Cerk*.

Enabling the flag

```
Advanced.MoveEquationsToDynamics
```

moves all nonlinear equations in the accepted sections to the dynamics section. This may be useful if problems with nonlinear equations in the accepted sections are experienced, especially if few output points are used. It forces the integrator to solve the nonlinear equations each integrator step and thereby it also updates the initial guesses more often.

5.6.10 Checking for unconnected physical connectors

A number of flags are available to warn for unconnected physical connectors:

To warn for all unconnected physical and input connectors, you can set `Advanced.Translation.WarnAboutUnconnected = true`. (The default is `false`.)

To warn for all unconnected physical connectors for selected domains, you can set, as an example for unconnected physical electrical and mechanical connectors (the call is a comma-separated list): `Advanced.Translation.UnconnectedDomainsRequiringWarning = "Modelica.Electrical, Modelica.Mechanics"`. (The default is an empty string `""`.)

To warn for any unconnected connector with stream (like in Modelica.Fluid), you can set `Advanced.Translation.WarnAboutUnconnectedStream = true`. (The default is `false`.)

Finally, to log (note, only log) all unconnected physical connectors, you can set `LogDefaultConnect = true`. (The default is `false`.)

5.6.11 Checking for structural singularities

Dymola has extended support for checking for structural singularities of the model equations.

When using a model for simulation it is a basic requirement that there are the same number of equations and number of unknown variables and that there is for each variable an equation from which the variable can be solved. This check is now done in more detail as it is done separately for each of the four basic data types Real, Integer, Boolean and String. This supports better checking. For example, let r be a real variable and let i be an integer variable and consider the equation r = i. This equation is a real equation, since we need to use it to solve for r. We cannot use it to solve for i. It may be remarked that the checking for structural singularities of the initialization problem has had this more detailed checking from the start. The simulation problem is more complex since it may have high DAE index which means that it is by definition singular if only the derivatives, der(x), and algebraic variables, v, are considered to be the unknown variables. It is necessary to also consider the appearances of x.

Using the check command enforces checking for structural singularities. No user is happy when the error message at translation says missing equations or too many equations. It may be that the components are used in a wrong way, for example a component is missing, but it may also be that a used component model is wrong. Dymola has extended the checking of non-partial models and block components to include checking for structural singularities in order to give component or library developers better support. To get a non-trivial result, Dymola puts the component in an environment that should reflect a general use of the model.

- All inputs of the model are considered to be known.

- Flow variables that are not connected will at translation be set to zero. However, checking a flow source (components that defines the flow variable) in such a way would make it singular. Such models are not intended to be used in that way. At checking Dymola instead generates for each flow variable a fictive equation referring all variables of all the connectors of the model component. The aim is to create the most general variable dependence that may be generated by connections.
- The equations of the model component may depend on the cardinality of its connectors (whether a connector is connected or not). The Check operation of a component considers all the connectors of the component to have connections on their outer sides.
- Overdetermined connectors for example in the MultiBody Library are dealt with in the following way. If an overdetermined connector of the model component is part of a connected set that has a root or potential root candidates everything should be fine. Otherwise, Dymola specifies the connector as a potential root and if it is selected as a root, Dymola adds fictive equations referring to all of the variables of the overdetermined connectors to compensate for the missing redundancy.

Checking of model components is done recursively. As indicated above a structural singularity may be caused by improper use of components or come from singular components. When trying to pinpoint a source of singularity we cannot assume that the components are correct because we have checked all model classes. First, the checking of a model component assumes a general use, however, when actually using a component, the environment are more specific and singularities may then show up. Secondly, modifiers with re-declare may imply drastic changes of a component and there may not be an explicit class to make relevant checks on. Thirdly, this is even more accentuated when there are class parameters to set. Fourth, dimension parameters may take other values then assumed when checking the component model. When Dymola finds a component to be singular, it makes a recursive check of the components. Dymola then tries to set up an environment that mimics the real environment in the best way. For example, a connector not being connected, the generic equations for the flow variables of that connector are not generated, but zero default values are used. For example, a flow source will then be diagnosed as singular. In addition, the cardinality is preserved. The error diagnosis output exploits the results. If a component is found to be singular, this is reported. If no component is found to be singular, the error message focuses on the use of the components.

The extended structural checking is enabled by default, but can be disabled by setting the flag `Advanced.ExtendedStructuralCheck = false`

If a model is found to be singular at translation, the components are checked recursively. This can be disabled by setting the flag `Advanced.ExtendedStructuralDiagnosis = false`

Connectors that are neither physical (matched flow and non-flow) nor causal will assume a suitable number of external conditions on them. If this corrects the problem, no recursive check is performed.

Models that by design are non-partial and structurally singular can use the `annotation(__Dymola_structurallyIncomplete);` This has been added to e.g. Modelica.Blocks.Math.TwoInputs, and to the base classes of MediaModels.

Component models of partial classes inhibit the structural check. However, the top-level of a model containing components of partial classes at top-level can be checked by setting the flag `Advanced.StructuralCheckOfTopLevel = true`. (The flag is by default false.) The check assumes that sub-models are correct and can be used to:

- Check a template-model (using partial models for replaceable components at top-level).
- Quickly finding some common errors at the top level for any other model. In this case, the check does not add any new information, but changes the order of the check; the top-level is checked first instead of last.

Note the limitation of this flag that "input connectors" of media are not handled correctly when checking Modelica.Media.Examples and Modelica.Fluid (primarily Sources).

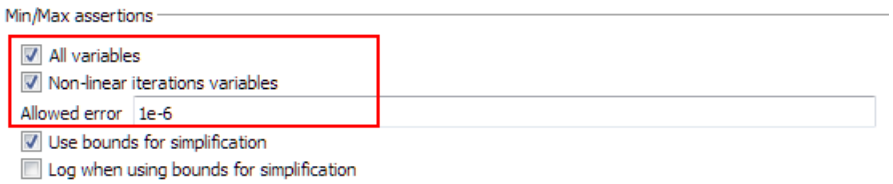
5.6.12 Bound checking for variables

Bounds checking for variables can be used to ensure that the solution is not only a numerical solution to the equations, but also satisfies additional bounds to and thus is physically correct.

Consider the following model where a length constraint should be satisfied and the length shall be positive:

```
model LimitProblem
  Real length(min=0);
equation
  length^2-length=1;
end LimitProblem;
```

This equation has two solutions: -0.62 and 1.62. However, the solution at -0.62 does not satisfy the min value. By using the command **Simulation > Setup**, selecting the **Debug** tab and then enabling **Min/Max assertions** (and allowing an error of 1e-6) it is guaranteed that an unphysical solution is not found and instead the physically correct solution at 1.62 is found. (Please see section "Debug tab" on page 812 for information about the Debug tab.)



Note that types in Modelica.Units.SI contain min values, and thus by using Modelica.Units.SI some min values are automatically applied. However, based on the actual model it might make sense to add stricter limits.

For the use of the two last settings in the image above, see "Simplifying if-then-else expressions by using min/max of variables" on page 1009.

5.6.13 Diagnostic message for inline handling

The flag `Advanced.Translation.Log.FailureToInline` can be set tor true to enable warnings to be written into the translation log whenever Dymola fails to inline a function with one of the annotations `Inline=true`, `LateInline=true`, or `InlineAfterIndexReduction=true`.

The flag is false by default.

5.6.14 Online diagnostics for non-linear systems

When a simulation is slow, it can be due to the non-linear systems of equations in the model. This is especially the case if the simulation is not yet started and the problem occurs for the initial equations.

As a contrived example, consider:

```
model SlowNonLinear
  Real x[10];
  function multiplySlowly
    input Real x[:];
    output Real y[size(x,1)];
  algorithm
    y:=x;
    for i in 1:1000000 loop
      y:=x+y;
    end for;
  end multiplySlowly;
equation
  multiplySlowly(x)=ones(size(x,1));
end SlowNonLinear;
```

Running this and pressing **Ctrl+C** at the start gives:

```
Log-file of program dymosim
(generated: Wed Dec 21 11:59:03 2005)

dymosim started
... "dsin.txt" loading (dymosim input file)

In non-linear solver after 11 function evaluations:
x[10] = 0
x[9] = 9.99999E-007
x[8] = 9.99999E-007
x[7] = 9.99999E-007
x[6] = 9.99999E-007
x[5] = 9.99999E-007
x[4] = 9.99999E-007
x[3] = 9.99999E-007
x[2] = 9.99999E-007
x[1] = 9.99999E-007
Integration status probed at T = 0
   CPU-time for integration      : 2.55 seconds
   Number of result points       : 0
```

```
Number of GRID   points       : 1
Number of (successful) steps  : 0
Number of F-evaluations       : 0
Number of Jacobian-evaluations: 0
Number of (model) time events : 0
Number of (U) time events     : 0
Number of state    events     : 0
Number of step     events     : 0
Minimum integration stepsize  : 0
Maximum integration stepsize  : 0
Maximum integration order     : 0
```

By pressing **Ctrl+C** twice instead of once the simulation is stopped and the user is additionally prompted for further commands:

```
Enter command: continue(c), quit(q), stop non-linear with
diagnostics(s), log event(le), log norm(ln), log
singular(ls)=log & allow singular systems log
iterationsnonlinear(li), log debugnonlinear(ld), log
resultnonlinear(lr)
Logging command syntax: log event true, log norm false, and log
norm reset
```

The last three log commands correspond to the setup in **Simulation > Setup**, the **Debug** tab, the **Non-linear solver diagnostics** group with (iterationsnonlinear=Nonlinear iterations, debugnonlinear=Details, resultnonlinear = Nonlinear solution). To log all subsequent non-linear iterations write:

```
log iterationsnonlinear true
continue
```

To stop the simulation, give the command 'quit' instead. The quit command generates the same diagnostics as if the non-linear system of equations did not converge.

5.6.15 Diagnostics for stuck simulation

When a simulation is stuck (or progressing very slowly) it is important to provide some form of simple diagnostics.

As an example consider the following example:

```
model StuckSimulation
  Real x(start=0.5);
equation
  der(x)=if x>0 then -1 else 1;
end StuckSimulation;
```

This model is a classical example of chattering after 0.5 seconds (when x has reached zero) since the derivative of x is -1 for positive x and +1 for negative x.

Such models can occur when manually writing e.g. friction elements such as clutches without a stuck mode (use Modelica.Mechanics.Rotational.Components.Clutch). Adding 'noEvent' around the if-expression is not a solution.

Diagnostics for example

Running this with lsodar/dassl gives a very slow progress, and it thus seems best to press **Stop**. This terminates the simulations and gives a log with:

```
Integration started at T = 0 using integration method DASSL
(DAE multi-step solver (dassl/dasslrt of Petzold modified by
Dynasim))
Integration terminated before reaching "StopTime" at T = 0.5
  WARNING: You have many state events. It might be due to
chattering.
  Enable logging of event in Simulation/Setup/Debug/Events
during simulation
   CPU-time for integration      : 9 seconds
```

Re-running and pressing **Ctrl+C** in the dymosim window gives a message:

```
Integration status probed at T = 0.5004871181
  WARNING: You have many state events. It might be due to
chattering.
  Enable logging of events by pressing ctrl-c twice and then:
log event true
continue
```

Enabling the logging in one of these ways gives a large number of messages of this type:

```
Expression x > 0 became false ( (x)-(0) = -3.93213e-013 )
Iterating to find consistent restart conditions.
      during event at Time :  0.5000000000003932
Expression x > 0 became true ( (x)-(0) = 1e-010 )
Iterating to find consistent restart conditions.
      during event at Time :  0.5000000001007865
Expression x > 0 became false ( (x)-(0) = -1.0025e-010 )
Iterating to find consistent restart conditions.
      during event at Time :  0.5000000003010365
```

The ‘x’ variables in the log are links as described in the section “Direct link in the error log to variables in model window” on page 982.

By using a fixed-step solver, Euler, the simulation runs to completion, but there is still chattering and thus the warning message is given at the end of the successful simulation.

Fast sampling

Another cause for slow simulations is that the model contains sampling at a high speed.

As an example consider

```
model RapidSampling
  Real x;
equation
  when sample(0, 1e-6) then
    x=pre(x)+1;
  end when;
end RapidSampling;
```

Running this with dassl/lsodar generates the diagnostics:

```
Integration terminated successfully at T = 1
  WARNING: You have many time events. This is probably due to
fast sampling.
  Enable logging of event in Simulation/Setup/Debug/Events
during simulation
   CPU-time for integration      : 57.1 seconds
```

The simulation does not stop because of this, but especially for a larger system, it will be slower than normal, and can be a cause for concern. If the simulation speed is acceptable, there is no need to enable the logging for further investigation.

Using fixed-step-size solvers such as Euler does not generate any diagnostics, since running a sampled system with a step size corresponding to the sampling rate is normal and not a cause for concern.

5.7 Improving simulation efficiency

We will in this section assume that the model runs, generates the correct results, but runs too slowly. Before the simulation is complete one can examine the result by explicitly loading the result file, `dsres.mat`, and then plot and/or animate variables. This makes it possible to determine if the result is correct or not.

We will assume that you have followed the guidelines for model development (see section "Guidelines for model development" on page 974); in particular that you have not ignored any warnings and errors detected by the translator, and have tested each sub-model in isolation.

A slow simulation can be caused by either to too much work spent on each time-step, each step is too short (and thus too many steps are taken), or because of the overhead with storing the result to a file.

A first step is plotting the CPU-time to determine if the problem occurs at some particular time, due to the model behavior at that point.

5.7.1 Time of storing result

Storing the result takes time.

One important aspect of a simulation in Dymola is to generate results in a file. In some cases the storing of the result (large result files) and not the integration is the cause of the "slow simulation".

For large systems the actual writing of the file can take substantial time, this can be checked by using the command **Simulation > Setup**, selecting the **Output** tab and deselecting storing of **State variables**, **Derivatives**, **Output variables** and **Auxiliary variables** (see section "Output tab" on page 809) and then re-running the simulation. In this case, no result file will be generated. By looking at the log file, it is possible to compare the time to normal simulations.

A large number of events (due to e.g. chattering, see the next section) will also influence the time of storing results. This can be checked using the command **Simulation > Setup**, selecting the **Debug** tab and deselecting **Events during initialization and simulation** in the **Simulation analysis** group (see section "Debug tab" on page 812) and then re-running the simulation.

For very smooth solutions, the interpolation of the solution in order to generate the result file can also take substantial time. This can be reduced by decreasing the number of output points or the increasing the interval length, see section "General tab" on page 800. If using an integrator with fixed step-size it is necessary to enter a non-zero value for the **Fixed Integrator Step** in this case.

5.7.2 Simulation speed-up

Multi-core support

Parallel model execution

Dymola can parallelize the evaluation of model equations for calculation of the derivatives for continuous-time integration.

This feature is activated by setting the flag

```
Advanced.ParallelizeCode = true
```

Dymola automatically inquires the number of cores.

Notes:

- Hyper-threading is included in this number, i.e. a dual-core processor is seen as 4 cores, and a quad-core as 8 cores.
- The Dymola calculated number of cores is seen in the command log (see below).

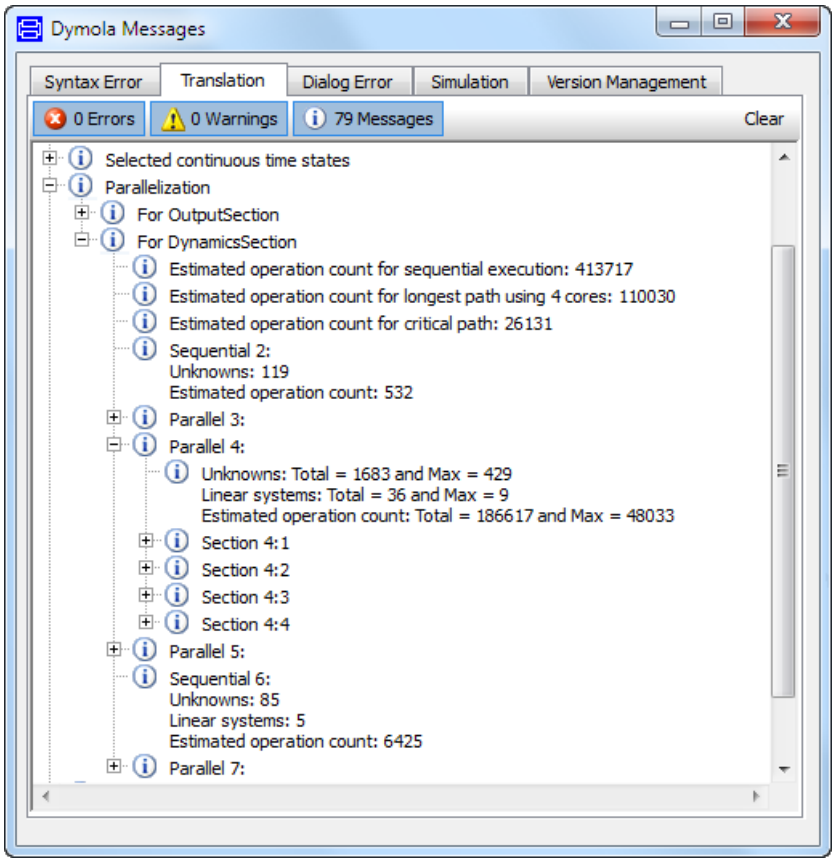
If a certain number of cores are to be used (including hyper-threading as above), the flag `Advanced.NumberOfCores` can be set to any wanted (positive) number of cores, this value overrides the automatically calculated value. An example of using this flag is when you want to create code for a machine with a higher number of cores than the machine you work on. To go back to using the automatically calculated number of cores instead, set this flag to 0. (This is also the default value of the flag.)

Notes:

- The compiler used must support OpenMP. For Visual Studio this means that you must use Visual Studio Professional 2012 or later, or Visual Studio Express 2012 or later. On Windows, the GCC versions that Dymola supports, also support OpenMP. This is the case also for the Intel compiler that Dymola supports.
- Multi-core simulation is currently not supported when using DLL or embedded server (DDE).
- On Windows, when using MinGW GCC compiler for 32-bit multi-core simulation, currently none of the following algorithms are supported: Radau, Esdirk23a, Esdirk34a, Esdirk45a, and Sdirk34hw.
- When using external functions in parallel code it is assumed that the functions are not thread-safe; if they are, see below.
- For source code generation the generated C-code will contain standard OpenMP-pragmas that are supported by many compilers (see their documentation).
- For some general tips about parallelization, see section “General tips about parallelization” below.

At translation, Dymola finds out which equations that can be executed in parallel. The result is a sequence of layers, where the layers have sections that can be executed in parallel.

The translation produces a log to be found in the log window.



The log reports that the calculation of the derivatives (the DynamicsSection) includes 413717 operations, while for the parallelization using 4 cores the longest path is 110030 operations, which means an estimated speed-up of 3.76.

The critical path is estimated to have 26131 operations, i.e., the Amdahl speed-up factor is 413717/26131 = 15.3, which indicates an upper limit of what could be obtained having many cores and neglecting overhead.

The log then reports the structure of the parallelization obtained. First there is a sequential part calculating 119 unknowns followed by 3 parallel layers, a sequential part and finally a parallel layer. The log for the parallel layer 4 is opened up and it reports that there are 4 parallel sections.

Dymola supports profiling. It is activated by setting `Advanced.GenerateBlockTimers = true` as usual. If parallelization has been activated at translation, the profiling result will also include timing results for the sequences and the parallel layers as well as for the individual sections. These are identified by Seq[i], Par[i] and Sec[i:j] where i and j are the numbers given in the log above.

The method for parallelization is described in the paper: H. Elmqvist, S.E. Mattsson and H. Olsson: “Parallel Model Execution on many cores”, Proceedings of the 10th Internal Modelica Conference:

https://www.modelica.org/events/modelica2014/proceedings/html/submissions/ECP14096363_ElmqvistMattssonOlsson.pdf

This paper includes successful uses from the thermodynamic and the electrical domains giving speed-ups of 2.9-3.4 on a quad-core machine.

It should be noted that for many kinds of models the internal dependencies don’t allow efficient parallelization for getting any substantial speed-up.

Parallel Jacobian update

In addition to setting the flag `Advanced.ParallelizeCode = true` to activate parallelization, you can also, in addition, activate parallel Jacobian update by setting the flag

```
Advanced.Translation.ParallelizeJacobianSystems = true
```

(The flag is by default `false`.) This flag is in particular useful when you work with models with large equation systems, using inline integration with any implicit solver.

Handling of external functions in parallel code

When using external functions in parallel code it is from Dymola 2016 FD01 by default *not* assumed that the functions are thread-safe; this means that all external function calls are seen as a critical region, and only one external function will execute at the same time.

This avoids unpredictable results, at the cost of efficiency. If you are *certain* that a specific external function is thread-safe, you the override the above for this function, with the following annotation in the external function:

```
annotation(__Dymola_ThreadSafe=true);
```

If the external functions are in a read-only library, you can set the flag

```
Advanced.ExternalFunctionCritical = false;
```

to indicate that the external functions are by default thread-safe – and to get the same behavior as in previous versions of Dymola (Dymola 2015 FD01 and earlier).

In addition, you can use the annotation

```
annotation(__Dymola_CriticalRegion="CLanguageIdentifier");
```

This indicates that the function is not thread-safe, and will create a named critical region. This ensures that only one external function with the same identifier will execute at the same time (but other functions may execute in parallel) - regardless of the setting of `Advanced.ExternalFunctionCritical`.

General tips about parallelization

In general, use equations to the greatest extent possible, not alogrithms. A tool such as Dymola has much greater freedom to process equations to produce efficient code.

Try to ensure that there is some form of dynamics separating the different parts of the model.

If you run a parallel code simulation outside of Dymola by using, for example, the command prompt, you can speed up the parallel simulation by setting the flag `OMP_WAIT_POLICY=PASSIVE` before running the simulation.

If it is not possible to naturally treat the different parts of a model in parallel, you might use the decouple operator (look at the `dymola\Modelica\Library\DecoupleBlocks.mo` package for more information). Note: Be careful to validate the results first.

Note the difference between running the simulation code of one model in parallel and running a number of simulations of the same model in parallel (to speed up for example the built-in function simulateMultiExtendedModel). Multiple simulations of the same model in parallel is by default activated, by the flag `Advanced.ParallelSimulations` being `0`. You should not mix the two ways of parallelizations; if you run a number of simulations of the same model in parallel, the flags `Advanced.ParallelizeCode` and `Advanced.Translation.ParallelizeJacobianSystems` should be `false`.

Simplifying if-then-else expressions by using min/max of variables

The min/max of variables can be used to simplify if-then-else expressions, such as `if x>=0 then … else` when x has a min-value of 0 (or higher), or a max-value below 0.

There are two settings available in the simulation setup (reached by the command **Simulation > Setup**, the **Debug** tab) for this feature:



(For information about the Debug tag, see “Debug tab” starting on page 812.)

The feature can be disabled by unchecking the setting **User bounds for simplification**. This corresponds to setting the flag `Advanced.UseMinMaxToSimplify=false`. (The option is by default activated.)

The simplifications can be logged by activating the setting **Log when using bounds for simplification**. This corresponds to setting the flag `Advanced.LogMinMaxSimplification=true`. (The option is by default not activated.)

Since this can cause unexpected behavior, assertions are automatically added ensuring that these variables are assigned values within the min/max limits (there exists a corresponding setting for all variables, **All variables**, the first one in the image above, note that this setting does not influence the min/max handling described here, for more information about this setting see the reference below). The automatic addition of assertions can be disabled by the setting (only available as flag):

```
Advanced.AutoAssertMinMax=false;
```

There is also a safety margin for time-varying variables, only simplifying when the limit is a bit beyond the min/max-limit – or it is the special case of simplifying semiLinear for non-reversible flow. This can be changed by changing the value of the setting (only available as flag):

```
Advanced.MinMaxSimplificationSafety
```

The default value of this setting is 0.1.

For simplifying semiLinear in general, there are no bounds, but an additional "allowed error". This is by default 0.5 and can be changed by the flag

```
Advanced.AssertInsideMinMaxNonReversibleLimit
```

The flag is by default 0.5. The setting is only available as flag.

For information about the three first setting in the image above, see "Bound checking for variables" on page 1000.

More efficient event handling

By default, the flag

```
Advanced.EfficientMinorEvents = true;
```

This activates more efficient handling of "minor" events.

These are events that either do not influence the continuous model at all (these events will be skipped; this is likely to occur when sub-sampling signals in synchronous models), or events that do not require a new Jacobian (sampled input to a continuous model; not implemented yet for lsodar and dassl).

The efficient event handling is not approximate in itself, but due to fewer Jacobians and changes in the simulation intervals for the numerical solvers, the results may change within the tolerance.

Using analytic ODE Jacobians

General

If the setting **Generate Analytic Jacobian for the ODE problem** is set in the simulation setup (reached by the command **Simulation > Setup**, the **Translation** tab), analytic ODE Jacobians are used during simulations, and also for the linearization. This is intended to improve the simulation speed, by computing the Jacobians more efficiently, but it generates a large amount of C-code that needs to be compiled. Note that the speed improvement may not be as large as assumed since the case where the flag is false (default value) has also been optimized. (There is a corresponding flag `Advanced.GenerateAnalyticJabobian` available for scripting.)

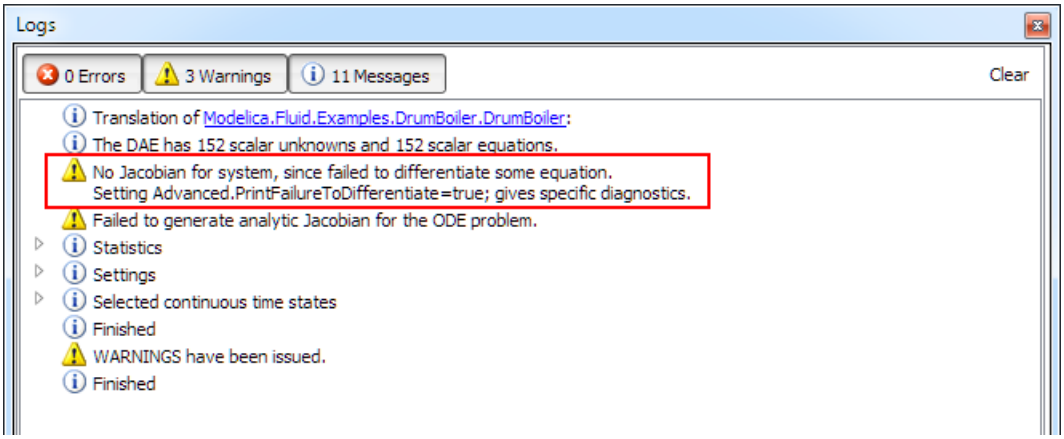
Sparse solver usage

If sparse solver usage is activated by the flag `Advanced.SparseActivate=true` these Jacobians will be stored in sparse matrix format during simulation and will be subject to sparse linear algebra.

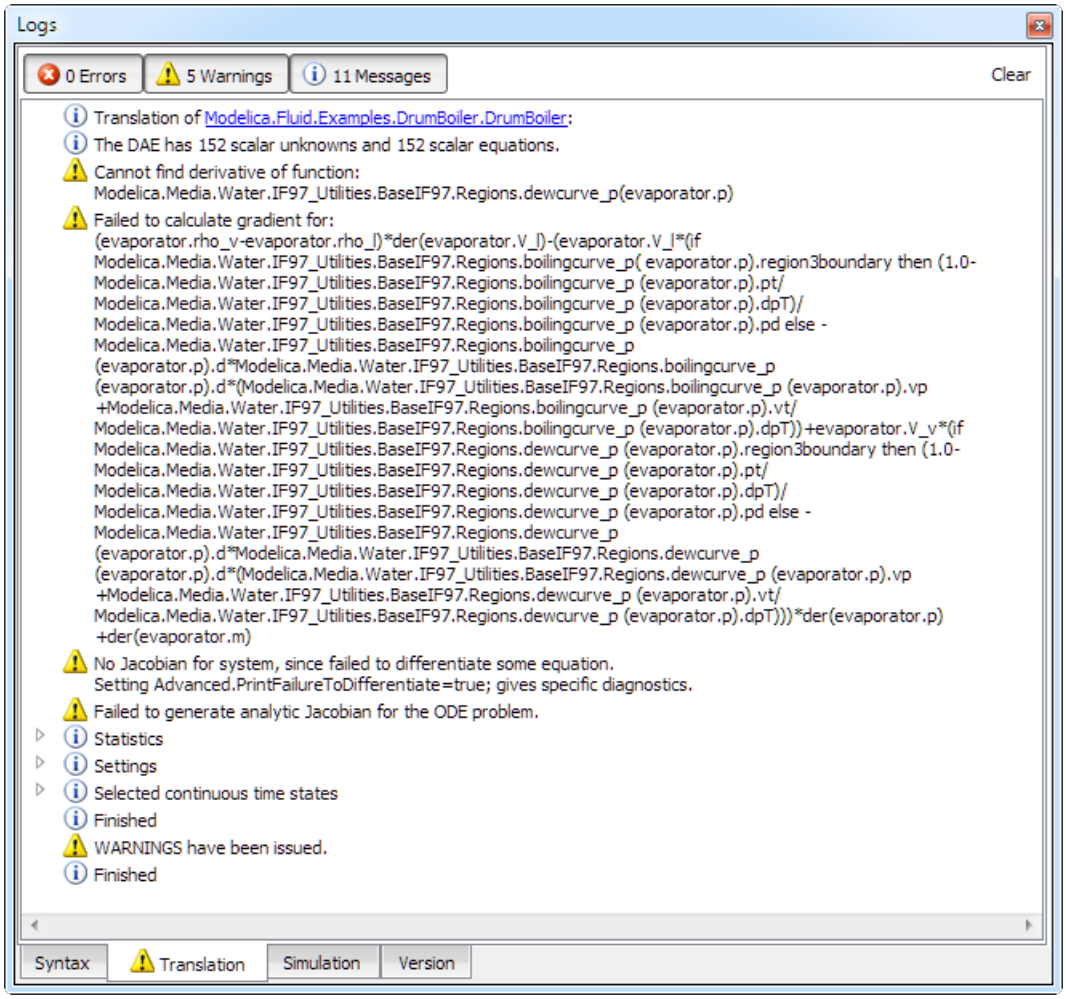
Diagnostic message for failed differentiation of Jacobians

Analytic Jacobian requires that functions can be differentiated. The flag `Advanced.PrintFailureToDifferentiate` allows displaying the analysis message of failed differentiation of Jacobians to the command log.

The use of the flag is indicated by the system when relevant. For example, setting the flag, opening the demo Modelica.Fluid.Examples.Drumboiler.Drumboiler, and then simulating it gives the following message:



Setting `Advanced.PrintFailureToDifferentiate=true` and trying to simulate again gives:



The flag is false by default.

5.7.3 Events and chattering

The first step to ensure that events are efficient is to examine the number of events in the log of the simulation. If the number of events is small compared to the number of steps, the simulation is not slow because of events.

A large number of state events can be caused by chattering, e.g. solving the model

```
model Chattering;
  Real x(start=0.3);

equation
  der(x) = -sign(x-0.2*time);
end Chattering;
```

After 0.25 seconds, the sign-function will switch back and forth between positive and negative. Activating 'Output Debug Information', see "Output tab" on page 809, will generate a log where the expression `x-0.2*time>0` switches between true and false.

```
Expression x-0.2*time > 0 became true ((x-0.2*time)-(0) =
0.0016)
Iterating to find consistent restart conditions.
      during event at Time :  0.252

Expression x-0.2*time > 0 became false ((x-0.2*time)-(0) = -
0.0008)
Iterating to find consistent restart conditions.
      during event at Time :  0.254

Expression x-0.2*time > 0 became true ((x-0.2*time)-(0) =
0.0008)
Iterating to find consistent restart conditions.
      during event at Time :  0.256

Expression x-0.2*time > 0 became false ((x-0.2*time)-(0) = -
0.0016)
Iterating to find consistent restart conditions.
      during event at Time :  0.258
```

In this case, a fixed step-size solver was used; an adaptive integrator would normally have had the events at almost identical times.

The solution may be to rewrite the model.

The result is a clear indication of chattering and the solution is to rewrite the model. Exactly how depends on the physical system, e.g. friction models usually have a stuck state in order to avoid similar problems, and bouncing balls can be modeled by an elastic impact zone.

If the events are time events it is not chattering, and only indicate fast sampling.

In order to reduce the simulation time if there are many step one can select an explicit solver, see "Specify simulation run" on page 616.

In some cases, it is possible to replace a detailed model generating many events by an averaging mode with a smaller number of events, e.g. a fast sampled controlled system can be replaced by an equivalent continuous-time controller, and a pulse-width modulated control signal by an average signal.

5.7.4 Debug facilities when running a simulation

When a simulation is running it is possible to open the result file to plot and animate the result using certain commands in order to see that the simulation is progressing correctly.

It is also possible to activate these actions individually before the simulation is started using the "Debug tab" on page 812 or by using the debug monitor to give certain commands when the simulation is running.

Debug monitor.

In order to determine the roots of some problems it is possible to enter a simple debug monitor for the simulation in order to determine how the simulation is progressing and to determine which variable is slowing down the simulation.

This requires that **DDE server** is **not** selected as compiler since the commands are entered in the DOS-window created when using the other compilers. Please see also "Compiler tab" on page 817.

Restore the DOS-window that will be present in the Windows lower toolbar during the simulation and press **Ctrl+C** to get the end-of-simulation statistics (where the simulation is, number of steps, number of function evaluations, etc.) Since parts of this information are stored internally by the integrators, the information might underestimate some of the statistics.

Debug commands.

By pressing **Ctrl+C** twice in rapid succession you will enter a debug monitor (pressing **Ctrl+C** once more will terminate the simulation). The following commands are available:

For continue/stop simulation two commands are available:

Command name (abbreviation)	Action
continue (c)	continue simulation.
quit (q)	stop simulation.

A number of commands are available for logging. Each command below (except the first one) represents a group of commands.

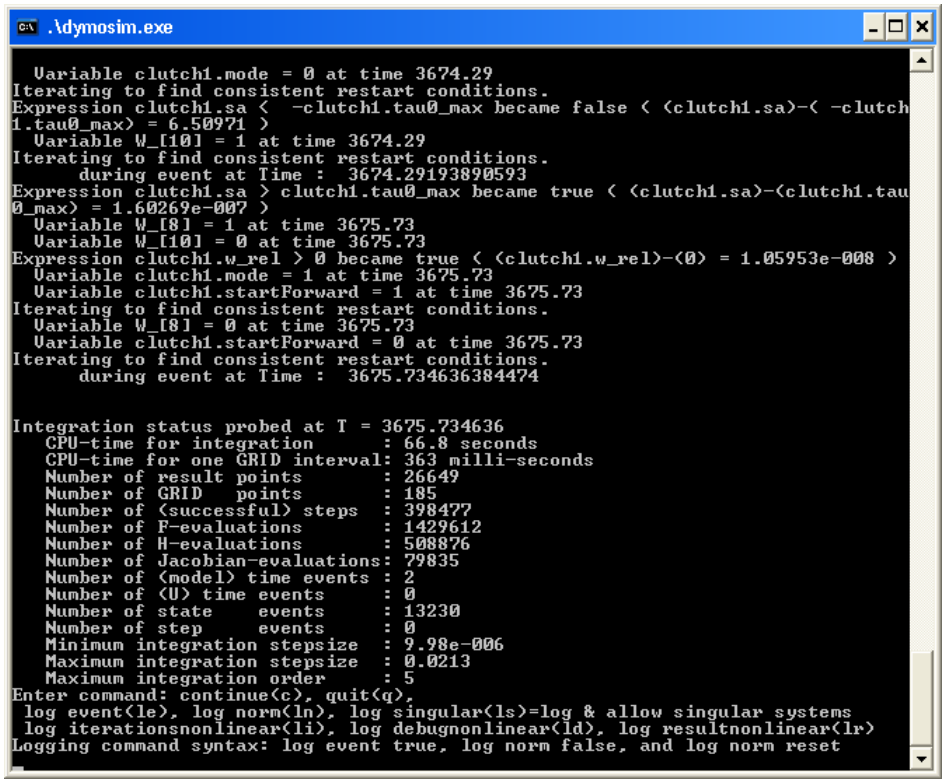
Command name (abbreviation)	Action
log (l)	give help for commands starting with l (all commands below)
log event true (le t)	activate logging of events
log norm true (ln t)	activate logging of dominating components
log singular true (ls t)	log and continue if singular system
log iterationsnonlinear true (li t)	log non-linear iterations
log debugnonlinear true (ld t)	log debug information for nonlinear systems
log resultnonlinear true (lr t)	log results of solving nonlinear systems

Each of the commands in the table above (except the first one) represents a group of four commands depending on the syntax. As an example, the log event true (le t) command represents the following commands;

Command name (abbreviation)	Action
log event true (le t)	activate logging of events
log event false (le f)	deactivate logging of events
log event reset (le r)	reset event logging to default (false)
log event (le)	displays status of logging of events and a clarification of the command

Of the log-commands **log event true** and **log norm true** are the most important. It is also possible to use **log event false** to turn it back off. Setting **log event true** makes it possible to activate logging of events at the right time and without seeing the log for non-linear system of equations. It can be used to determine if there are any problems with chattering.

Example of listing of log events.

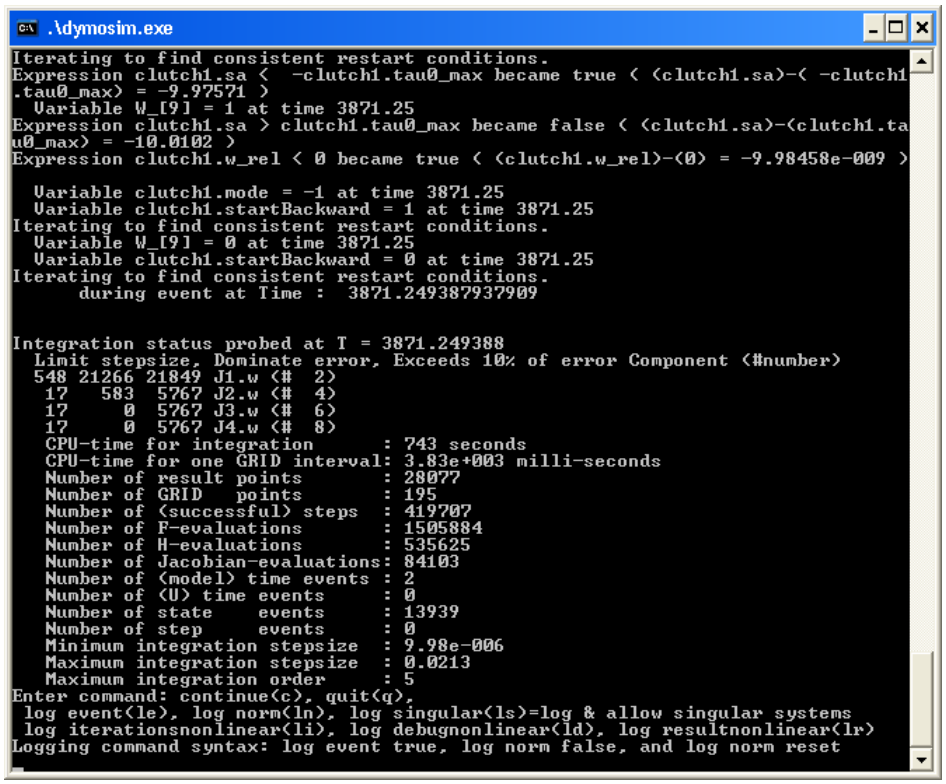


Integrator error statistics.

Setting **log norm true** makes it possible to determine which variable is causing an adaptive integrator, e.g. Dassl, to be slow. In the end-of-simulation statistics it will include statistics for each variable indicating

Column	Indicates
Limit stepsize	How often the error in the component has exceeded the tolerance and forced the integrator to reduce the step size.
Dominate error	How often the component has dominated the error estimate.
Exceeds 10 % of error	How often the error contribution from the component has exceeded 10 % of the total error estimate.
Component	Name of the state.
(#number)	State number.

Example of listing of integrator error statistics.



Hopefully a few states will dominate this statistics, and this should have localized the problem to those few variables. By plotting those variables one can see if it is caused by undamped oscillations, in which case extra damping can be a solution, or merely due to highly complex behavior, in which case a simplified model can be more appropriate.

5.7.5 Profiling

The previous sections makes it possible to find out if the code is using too many steps, in some cases the number of steps seems correct, but the cost per step is too large. If the time per step is too large one must determine where the time is spent in evaluating the model equations.

Precsion timing

To be able to use profiling, precision timing is needed. Precison timing is by default always activated when timers are used (that is, when any of the flags `Advanced.GenerateBlockTimers`, `Advanced.GenerateFunctionTimers`, or `Advanced.GenerateTimers` are set to `true`). This also means that if no timers are generated, the time reported in the normal simulation statistics in the simulation log does not use precision timing.

The precision timing functionality uses the frequency counter of the processor to provide high-resolution time measurements.

(The frequency of the processor can be specified (in Hz) using the flag `Advanced.Define.UseRDTSCFrequency`. However, it is recommended to keep this flag to the default value of 0.0 meaning that the frequency is automatically read from the text description of the processor.)

(In Windows, precision timing can also be accomplished by first selecting **DDE server**, since it has access to highly accurate timers (accuracy in microseconds or nanoseconds), see "Compiler tab" on page 817. Selecting other compilers will only generate timers with millisecond resolution, which often is too inaccurate. However, note that if timers are used, precision timing is available for all compilers.)

(In Dymola versions up to and including Dymola 2022 x, the flag, `Advanced.Define.PrecisionTiming` controlled if precision timing was used. From Dymola 2023, the value of the flag is by default always `true`, but precion timing is only activated when any timers are generated.)

Storing and presentation of timer values

In the simulation log

When timers are activated and a simulation is performed, you can see the timer values as a table in the end of the simulation log file. (Note that it might be necessary to open the log file (`dslog.txt`) in an editor to view the entire table.)

In the timers.txt file

When the flag `Advanced.Debug.WriteTimerResultsToFile` is set to `true`, the results of any timers will be printed, as a semi-colon separated list, to the file `timers.txt`. The flag is by default `true`. Currently, the text file is updated once per second when simulating, if timers are generated.

In the Timers tab in the Simulation Analysis dialog

If timers are activated, and timer values are written in the `timers.txt` file (see previous section), the timer values are displayed in the Simulation Analysis dialog, in the **Timers** tab. This tab has two subtabs, **Block timers**, and **Function and FMU timers**. For more information about this dialog, see section "Analysing timers" starting on page 666.

In the dsmodel.mof file

Most block timers are printed to `dsmodel.mof` if the following flags are set:

```
Advanced.GenerateBlockTimers = true;
Advanced.OutputModelicaCode = true;
```

(By default, these flags are `false`.)

The block numbers match those in the simulation log (see next section).

Basic profiling

To turn on profiling write the following at Dymola's command input

```
Advanced.GenerateBlockTimers=true
```

(The flag is by default `false`.) Note that you can set the flag by the setting **Generate function and FMU timers** in the simulation setup, the Debug tab. See section "Debug tab" starting on page 812.

After having set the flag/setting, then translate the model, and run the simulation. At the end of the log file, a table contains the profiling information.

We exemplify the usage of block timers by simulating the model `Modelica.Mechanics.MultiBody.Examples.Loops.EngineV6_analytic` (which takes about 0.1 second to simulate). Below follows a timing of such a simulation.

Note that from Dymola 2023 you can see the table in the **Timers** tab in the Simulation Analysis dialog. See section "Analysing timers" starting on page 666.

```
Profiling information for the blocks.
Estimated overhead per call 0.01[us] total 0.001[s]
the estimated overhead has been subtracted below.
Name of block              , Block, Total CPU[s],  Mean[us]  ( Min[us] to Max[us] ),  Called
Entire model               : 0,            0.139,      10.88 ( 0.53    to   464.91),   12740
Event handling             : 1,            0.003,      13.59 ( 11.18   to    49.33),     252
Empty timer                : 2,            0.000,       0.01 ( 0.00    to     0.05),   12744
Outside of model           : 3,            0.034,       2.68 ( 0.07    to  2140.39),   12742
Parameters                 : 4,            0.000,     203.73 ( 203.73  to   203.73),       1
   Parameter code          : 5,            0.000,     197.46 ( 197.46  to   197.46),       1
   Parameter code          : 6,            0.000,       6.12 ( 6.12    to     6.12),       1
InitialSection             : 7,            0.000,     292.15 ( 292.15  to   292.15),       1
   Initial code            : 8,            0.000,     232.99 ( 232.99  to   232.99),       1
   initialization.linear[1]: 9,            0.000,      57.24 ( 57.24   to    57.24),       1
   Initial code            : 10,           0.000,       0.07 ( 0.07    to     0.07),       1
OutputSection              : 11,           0.123,      10.26 ( 8.29    to   463.70),   11990
   Output code             : 12,           0.044,       3.67 ( 2.92    to   265.53),   11990
```

```
   simulation.linear[1]    : 13,          0.078,       6.46 ( 5.07    to   458.66),   11990
   Output code             : 14,          0.000,       0.02 ( 0.01    to     0.73),   11990
DynamicsSection            : 15,          0.001,       0.06 ( 0.03    to    16.61),   11991
   Dynamics code           : 16,          0.000,       0.01 ( 0.00    to     0.12),   11991
AcceptedSection2           : 17,          0.007,       1.62 ( 1.21    to    28.02),    4337
   Auxiliary2 code         : 18,          0.007,       1.58 ( 1.15    to    27.94),    4337
```

The first lines state that we have estimated the overhead of the timers to 0.01 microseconds and subtracted them from all timing estimates, thus making it easier to find the problematic block. The total overhead is also included for comparison.

It is then necessary to examine the file dsmodel.c in order to determine what block number corresponds to what variable(s). The start of block 6 is marked by DymolaStartTimer(6) and the end by DymolaEndTimerName(6). The first blocks are special:

Block #	**Task**
0	The total time for all model evaluations.
1	The total time for model evaluations during event handling.
2	An empty block, included just for comparison.
3	The total time spent between model evaluations.

For each of these blocks we have only subtracted the overhead for its own timer and not for additional timer calls that occur during its call. Thus the total time for all model evaluations should more accurately be around 0.139 s - 0.001s (total overhead) = 0.138 s.

In this example, the event iterations are insignificant compared to the rest, even though there are 125 event iterations.

The remaining blocks are code-blocks, either system of equations (including a trailing part of torn equations) or blocks of non-equations.

Among these blocks, the most time is spent in block 13. This linear system of equations accounts for about 0.08 s of the models total 0.14 s. It represents the global kinematic loop that is defined by the six pistons through their connection to the crankshaft. However, note that each cylinder in the `EngineV6_analytic` model contains a loop breaker joint that analytically solves all kinematic loops. Thus, `simulation.linear[1]` is entirely solved during translation. During simulation, it only consists of assignment statements (as can be seen under Statistics in the Translation log, where the equation´s size is listed as zero).

Note that block 8 has the largest average time of the normal blocks, but it has no significant influence on the total CPU-time. This block consists of variables that are parameter-dependent and thus are constant during continuous integration, and is thus evaluated only once.

One should remember to reset the flag

```
Advanced.GenerateBlockTimers=false
```

in order to avoid the overhead for the next model. This flag is not cleared by **Clear** or **Clear All** commands.

The timer overhead also affects the total simulation time. Also, note that the accurate timers measure wall-clock time and not CPU-time in the simulation process. Thus, simple statements

can have large maximum times if another process interrupts them between the start of the timer and the stop of the timer. This should not affect the average and minimum times.

Fine grained profiling

In some cases, the profiling indicates that one non-equation block is the cause of the problem. In order to determine what is the cause within that block it is possible to turn on a finer grained profiling by writing the following at Dymola's command input

```
Advanced.GenerateTimers=true
```

Then translate the model, and run the simulation. At the end of the log file, a large table contains the profiling information. Note that it might be necessary to open the log file (dslog.txt) in an editor to view the entire table.

Note that from Dymola 2023 you can see the table in the **Timers** tab in the Simulation Analysis dialog. See section "Analysing timers" starting on page 666.

By running the `EngineV6_analytic` model again with fine-grained profiling, one gets results such as

```
Profiling information for the blocks.
Estimated overhead per call 0.01[us] total 0.001[s]
the estimated overhead has been subtracted below.
Name of block             , Block, Total CPU[s],  Mean[us]  ( Min[us] to  Max[us] ),  Called
Entire model              : 0,           0.538,      42.22 ( 1.33    to    569.62),   12743
Event handling            : 1,           0.017,      65.65 ( 51.94   to    465.13),     252
Empty timer               : 2,           0.000,       0.01 ( 0.00    to      0.13),   12744
Outside of model          : 3,           0.150,      11.74 ( 0.13    to 100523.00),   12743
…
OutputSection             : 3550,        0.413,      34.47 ( 28.34   to   567.45),    11990
…
   Output code            : 3648,        0.000,       0.01 ( 0.00    to     0.13),    11990
   Output code            : 3649,        0.002,       0.20 ( 0.10    to    30.42),    11990
   Output code            : 3650,        0.000,       0.01 ( 0.00    to    19.31),    11990
…
   Output code            : 3748,        0.000,       0.01 ( 0.00    to     0.14),    11990
   Output code            : 3749,        0.002,       0.17 ( 0.10    to    24.16),    11991
   Output code            : 3750,        0.000,       0.01 ( 0.00    to    16.59),    11991
…
   simulation.linear[1]   : 4157,        0.088,       7.30 ( 5.70    to   288.06),    11990
      Lin sys total       : 4158,        0.087,       7.26 ( 5.65    to   287.97),    11990
      Lin sys torn        : 4163,        0.026,       2.18 ( 1.64    to   264.76),    11990
```

The dominating block 4157 is the equation block found already with the coarser grained profiling. Note that more details about the linear system are given here. However, since the system is solved already during translation, no time is spent during simulation to construct right-hand-side vectors and system matrices. Nor is any time spent solving the resulting linear equations. Thus only the time for the torn assignments are listed above.

The blocks 3649 and 3749 (as well as 3849, 3949, 4049, and 4146) are complex equations giving the gas force in each cylinder. Since these equations have event generating '>' and '<' one might try to use the noEvent operator. When using noEvent one **must** first check that the equation is sufficiently smooth. Since part of the equation is a piece-wise polynomial, we

evaluate both pieces at the switch and observe that they **do not** give the same result. Thus, noEvent **must not** be used here. Additionally, the **Event Log** tab of the **Simulation Analysis** command tells us that these events only happen once during simulation. They are therefore not a big issue. (For information about the Event Log, see section "Event logging user interface" starting on page 654.)

On the other hand, rewriting the polynomials with Horner's rule decreased the average from about 0.2 to 0.05 microseconds. This had marginal influence on the total CPU-time in this example.

One should remember to reset the flag

```
Advanced.GenerateTimers=false
```

in order to avoid the overhead for the next model. Note that this flag is not cleared by **clear** or **Clear All** commands.

Profiling of function calls

It is possible to activate profiling of function calls. This can be used to, for example, investigate external function calls, especially FMUs. Another usage can be to find time-consuming media functions.

To activate profiling of function calls, set the flag

```
Advanced.GenerateFunctionTimers = true
```

(The flag is by default `false`.) Note that you can set the flag by the setting **Generate function and FMU timers** in the simulation setup, the Debug tab. See section "Debug tab" starting on page 812.

This profiling can be used together with profiling described in previous sections, but can also be used on its own.

5.7.6 Inline integration

In order to increase the simulation speed, in particular for real-time simulation, Dymola provides inline integration. Inline integration is a combined symbolic and numeric approach to solving differential-algebraic equations systems. Discretization expressions representing the numerical integration algorithm are symbolically inserted into the differential algebraic equation model. The symbolic power of the Dymola translator is then exploited to transform the equations into a representation that is efficient for numeric simulation. The method of inline integration was presented in Elmqvist *et al.* (1995).

Inline integration

Consider an ordinary differential equation on explicit state space form

$$\dot{x} = \mathrm{f}(x,t); \qquad x(t_0) = x_0$$

where x is the vector of state variables and t denotes time. It is straightforward to solve this problem by using an explicit integration algorithm. In the simplest case, using the Euler

forward algorithm, the derivative of the state vector is approximated by a forward difference formula:

$$\dot{x}(t_n) = \dot{x}_n \approx \frac{x_{n+1} - x_n}{h}$$

where $x_{n+1} = x(t_{n+1})$ is the unknown value of x at the new time instant $t_{n+1} = t_n + h$, $x_n = x(t_n)$ is the known value of x at the previous time instant t_n, and h is the chosen step size. Inserting the discretization expression into the model equations leads to the recursion formula:

$$x_{n+1} = x_n + h \cdot \mathrm{f}(x_n, t_n); \quad x_0 \text{ is known}$$

which is used to "solve" the ODE.

Unfortunately, explicit integration algorithms are not well suited if systems are stiff. These are systems with dynamically fast and highly damped components. If an explicit algorithm is used to integrate such systems, the step size is limited due to stability problems of the integration algorithm. If the step size is too large, then the computed trajectory starts to oscillate and diverge. The standard cure is to use implicit algorithms. However, this leads to a non-linear equation system that has to be solved at each step. The simplest example of an implicit algorithm is Euler backward. The derivative of the state vector is approximated by a backward difference formula:

$$\dot{x}(t_n) = \dot{x}_n \approx \frac{x_n - x_{n-1}}{h}$$

leading to the discretized problem

$$x_{n+1} = x_n + h \cdot \mathrm{f}(x_{n+1}, t_{n+1})$$

which at each time-step this has to be solved for x_{n+1}.

The idea of inline integration is to let the Dymola translator manipulate the discretized problem to reduce and simplify the problem that has to be solved numerically at each step. A set of differential-algebraic equations can be written as

$$0 = g(t, x, \dot{x}, v)$$

where x is the vector of variables appearing differentiated and v is the vector of unknown algebraic variables. Normally, Dymola manipulates this problem to solve for v and $\dot{x}$, and a numerical integration algorithm is used to integrate for x. When inline integration is used the Dymola translator first reduces the DAE index to one and then adds the discretization equations. Assume that the original problem is index one, then using Euler backward turns the problem into

$$0 = f(t_n, x_n, \dot{x}_n, v_n)$$

$$x_n = x_{n-1} + h \cdot \dot{x}_n$$

with x_n, $\dot{x}_n$ and v_n being the unknowns to be solved for. The Dymola translator manipulates the problem to facilitate the numerical solution work.

A number of inline integration algorithms can be used in Dymola, please see next section.

Inline integration in Dymola

A number of inline integration algorithms/methods for real-time simulation can be used in Dymola, either as settings or as a built-in variable.

Algorithm	Built-in variable value
Inline integration algorithm not used	0
Explicit Euler	1
Implicit Euler	2
Trapezoidal method	3
Mixed explicit/implicit Euler[16]	4
Implicit Runge Kutta	5
Explicit Runge Kutta	6
Rosenbrock	7

Selecting the algorithm using menu settings

Use the **Simulation > Setup** menu, select the **Realtime** tab. Here the algorithm can be selected using a drop-down menu. The Runge Kutta order can be selected between 1 and 4 for the three last methods in the table. (Selecting 1 for Implicit/Explicit Runge Kutta means using Implicit/Explicit Euler.)

Optimize code for step size can be selected. For more information, please see section “Realtime tab” on page 820. Please note the corresponding settings that must be done in the General tab!

Selecting the algorithm using a built-in variable

As an example the Trapezoidal algorithm should be used. This will be obtained by setting the built-in variable:

```
Advanced.InlineMethod:=3;
```

If implicit Runge-Kutta of order 4 should be used two variables have to be used:

```
Advanced.InlineMethod:=5;
Advanced.InlineOrder:=4;
```

Prerequisites

Please note that some settings have to be made in the **Simulation > Setup…** menu, the **General** tab. Which settings are described in the section “Realtime tab” on page 820.

[16] The mixed explicit/implicit Euler is particularly useful for mechanical systems.

Diagnostics for inline handling

See section “Diagnostic message for inline handling” on page 1001.

More information and advanced options

Real-time simulation in other simulation environments is described in the chapter “Simulation Environments”. Some advanced options are also described here.

More information about real-time simulation can also be found in the report "Dymola Application Note HILS".

References

Elmqvist, H., M. Otter, and F. E. Cellier (1995): “Inline Integration: A New Mixed Symbolic/Numeric Approach for Solving Differential-Algebraic Equation Systems”, In Proceedings of ESM'95, SCS European Simulation MultiConference, Prague, Czech Republic, pp. xxiii-xxxiv.

5.8 Initialization, steady state, and start values

Please also see, in the chapter “Introduction to Modelica”, section “Initialization of models”.

5.8.1 The continue command

Please note that when using the command **Simulation > Continue > Continue** or **Simulation > Continue > Import Initial...** the methods below are not used, they only apply to the first time start values are specified.

5.8.2 Over-specified initialization problems

At translation Dymola analyses the initialization problem to check if it is well posed by splitting the problem into four equation types with respect to the basic scalar types Real, Integer, Boolean and String and decides whether each of them are well-posed. If such a problem is over-specified, Dymola outputs an error message indicating a set of initial equations or fixed start values from which initial equations must be removed or start values inactivated by setting fixed=false.

Dymola stops translation when an over-specified problem is found. It may be the case that this sub problem also has under-specified parts, i.e., variables that cannot be uniquely determined. Moreover, the sub problems for the other data types may also be ill posed. To find this out, try to correct the reported problems by removing initial conditions and retranslate to check for additional diagnostics.

5.8.3 Discriminating start values

The initialization problem often involves nonlinear algebraic loops. When solving a nonlinear system of equations, it is important that the iteration variables have good start values to

guarantee convergence as well as convergence to the desired solution in case the nonlinear system has several solutions. Dymola decreases the size of a nonlinear system by eliminating variables by tearing. It is thus a good idea to eliminate variables without start values or variables having less confident start values and keep those variables having more confident start values as iteration variables.

Old text, may be moved to new subsection according to Hans

Models are typically built by composing components. The components may include settings for start values. If the top model sets parameters or start values of the components, the idea is to improve the model, not make it worse. When building the top model, we usually have better and more specific information than when the component models were built. Thus it is reasonable to have more confidence in start values being set from higher hierarchical levels of the model. This approach gives both the model developers and the users good control possibilities. Changing a start value is expected to have an immediate effect.

A model has a hierarchical component structure. Each component of a model can be given a unique *model component hierarchy level number*. The top level model has a level number of 1. The level number increases by 1 for each level down in the model component hierarchy. The model component hierarchy level number is used to give start values a *confidence number,* where a lower number means that the start value is more confident. Loosely, if the start value is set or modified on level *i* then the confidence number is *i*. If a start value is set by a possibly hierarchical modifier at the top level, then this start value has the highest confidence, namely 1 irrespectively on what level, the variable itself is declared.

The feature can be disabled by setting
`Advanced.UseConfidenceNumberForStartValues = false`

Let us consider the situations where Dymola can select between start values. The very first situation is elimination of alias variables. A connection between non-flow connectors gives an equation v1= v2. A connection between two flow connectors gives v1+v2=0. Dymola exploits such simple equations to eliminate variables. In this elimination procedure, Dymola keeps the start value.

The start-value priority also considers the parameter-level if the start value is bounded to a parameter.

(This change can be disabled by setting the new flag `Advanced.Translation.UseParameterLevelForStartValue = 0`. The default value of the flag is 2, meaning that parameter-level is considered everywhere. You can also set the flag to 1, that means only use parameter-level for non-iteration, trusting literals more.)

You can now also get an overview of the prioritized conflicting start-values setting the new flag:

```
Advanced.Translation.LogStartValuePriority = true
```

(The flag is by default `false`.)

The sorting procedure of Dymola finds the minimal loops, which means that the sorting is unique and such a loop cannot be made smaller by sorting the variables and the equations in another way. It means that the set of intrinsic unknowns of an algebraic loop is well defined.

In order to obtain efficient simulation, it is very important to reduce the size of the problem sent to a numerical solver. Dymola uses a tearing approach to "eliminate" variables. The numerical solver is only made aware of the remaining variables, the iteration variables, call them z. A numerical solver provides values for the z variables and would like to have the residuals of the remaining equations calculated. The tearing procedure has produced a sequence of assignments to calculate the eliminated variables, v, assuming z to be known. The start values of the eliminated variables have no influence at all. An aim is of course to make the number of components of z as small as possible. It is a hard (NP-complete) problem to find the minimum. However, there are fast heuristic approaches to finding good partitions of the unknowns into v and z. In order to get good start values for the numerical solver, Dymola tries first to eliminate variable with less reliable start values.

As for usual or iteration variables of nonlinear system of equations, the plot browser provides support for setting start values interactively. If the start value of the unknown is bound to a parameter expression, then setting any of the parameters appearing in the expression will of course influence the start value. If no start value is given or if it is a literal number, then it is possible to set it interactively. In the variable browser, click on **Advanced**, and then click the button labeled **v0≈**, and the interactively setting is enabled.

Setting

```
Advanced.LogStartValuesForIterationVariables = true;
```

before translation, will make Dymola produce a listing of all iteration variables and their start values.

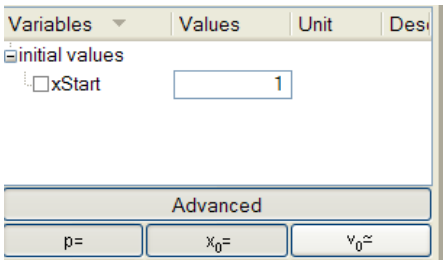
Example

Consider the following example intended to illustrate both the improved heuristics and setting start values:

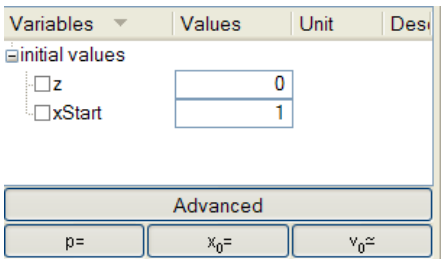
```
model NonLinear
  Real x(start=xStart);
  Real y;
  Real z;
  parameter Real xStart=1;
algorithm
  x:=(y+z)+time;
algorithm
  x:=(y+z)^2;
equation
  0=y-z;
end NonLinear;
```

Note: Using algorithm in this way for actual models is not good since Dymola manipulates algorithms less and thus algorithms often lead to harder numeric problems (larger system of equations, no analytic Jacobian, no alias elimination between and y and z). Rewriting them as equations would be a good idea.

Translating this model gives an input field for initial values:



Enabling guess values ($v_0 \approx$) gives the prompt



Setting 'z' to 1 generates another solution for this non-linear system of equations.

5.8.4 Debugging failed initialization

Using plot dependencies after failed initialization

The feature plot dependencies can be used also after failed initialization, if the setting **Store variables after failed initialization** is activated before the simulation. This setting is available in the simulation setup, which can be reached by the command **Simulation > Setup**, the **Debug** tab.

The feature is available in the same way as usual, note however that the presented dependencies are for the simulation problem, the dependencies for the initialization problem may differ, cf. dsmodel.mof. For using plot dependencies, see section "Plot dependencies" starting on page 633.

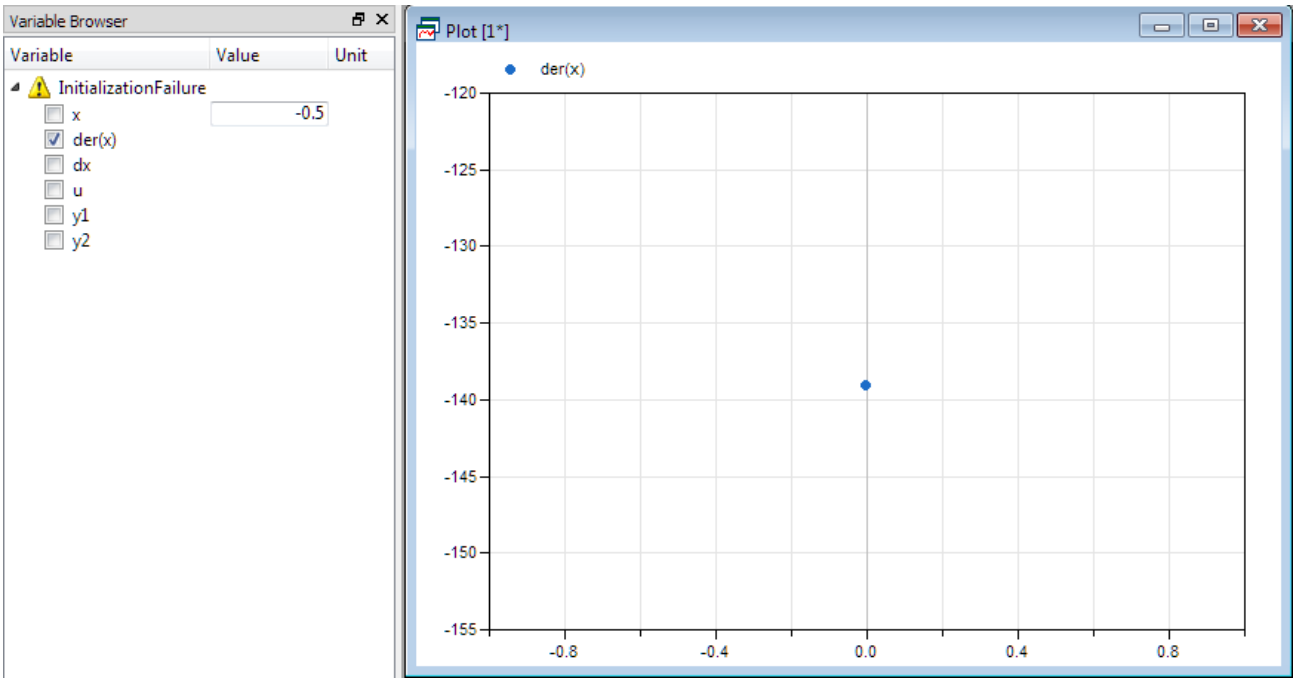
Plotting the results of the failed initialization using the variable browser

If you have activated the setting **Store variables after failed initialization** as described in previous section before simulation, all variables, ready for plotting, are now available in the Variable Browser after a failed initialization. The value of each variable depends on how far the initialization process came before failing. If the variable was initialized the result of that initialization is presented. If the variable was not initialized its start value is instead presented. The simulation log contains information on what made the initialization fail. The execution order can be found in e.g. `dsmodel.mof` (to generate this file, activate the setting **Generate listing of translated Modelica code in dsmodel.mof** before translation, the setting can be reached by the command **Simulation > Setup**, in the **Translation** tab).

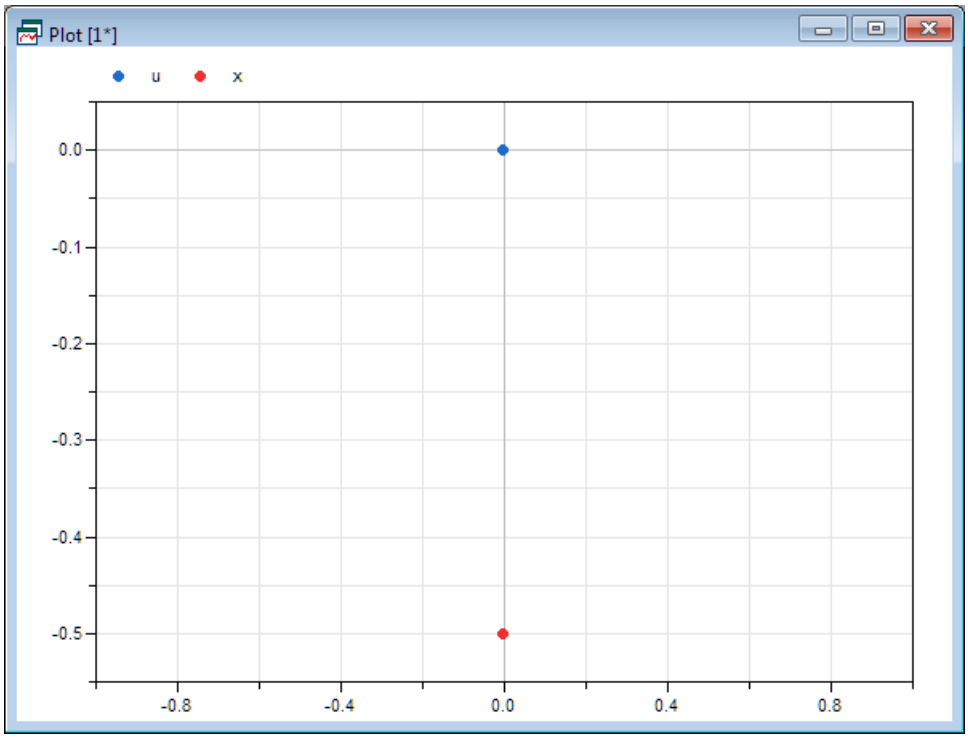
Consider the example model:

```
model InitializationFailure
  Real x(start=-0.5, fixed=true);
  Real dx(start=-0.9);
  input Real u;
  output Real y1(start=-1);
  output Real y2(start=-1);
equation
  der(x) = dx;
  u*dx + exp(dx) = x;
  y1 = x^3;
  y2 = der(x)^3;
end InitializationFailure;
```

To compute the variable `der(x) = dx` Dymola will, during translation, generate a nonlinear equation in this variable. When simulating the model Dymola complains that it cannot solve this nonlinear equation and the initialization fails. Plotting the iteration variable `der(x) = dx` gives the last attempted value in the Newton iteration.



By a closer look at the model one can see that the nonlinear equation depends on the variables u and x. (This information is also readily available in the translation log if you enable the setting **List non-linear iteration variables** before translation (the setting can be reached by the command **Simulation > Setup**, in the **Translation** tab) or in `dsmodel.mof` (see above how to generate this file).) Plotting these variables reveals that `u = 0` and `x = -0.5`.

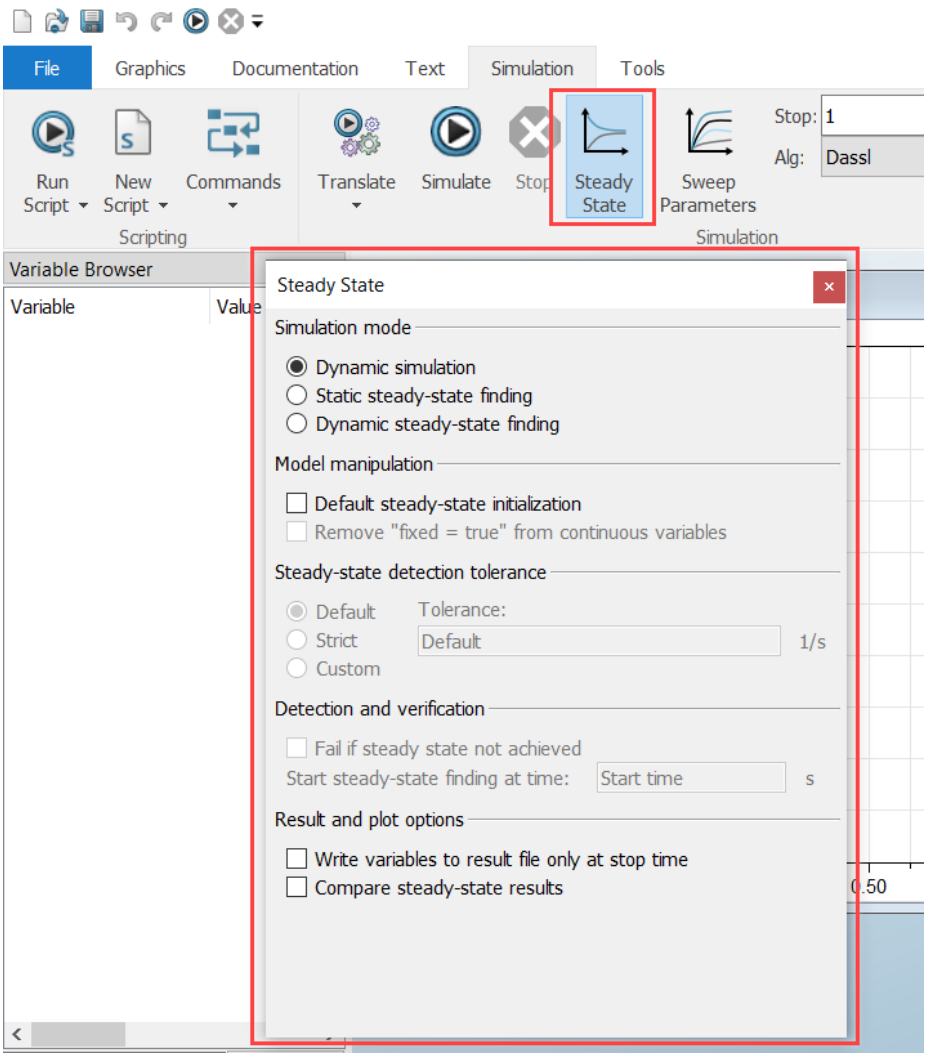


Thus, the nonlinear equation reduces to `exp(dx) = -0.5`, which lacks solution. The source of the initialization problem has been isolated.

Also note that a look into dsmodel.mof reveals that y1 is computed before the attempt to solve the nonlinear equation. The variable y2 is computed after. This agrees with the values presented when plotting these variables: `y1 = -0.125` and `y2 = -1`. The former variable has it computed value `x^3`, whereas the latter variable has its start value -1.

5.8.5 Steady state solver interface

Steady state handling in Dymola is in general handled by the command **Simulation > Steady State**:



The command displays a dialog with steady-state simulation options. The dialog is dockable like the Sweep Parameters dialog. These two dialogs may be used together to perform batch runs of steady-state simulations.

The main selection in the Steady State dialog is selecting the simlulation mode.

Simulation mode

You can select any of the following three simulation modes:

Dynamic simulation

This is the default selection.

The flags corresponding to this default selection are:

```
Advanced.Simulation.SteadyState = false
Advanced.Simulation.SteadyStateTermination = false
```

For more information about these flags, see the steady state alternatives below.

Static steady-state finding

Selecting this option sets the stop time for the simulation equal to the start time. Thus, a purely static simulation will be performed. When simulating with this option Dymola will also tag the result as a steady-state result. See "Plot options" below.

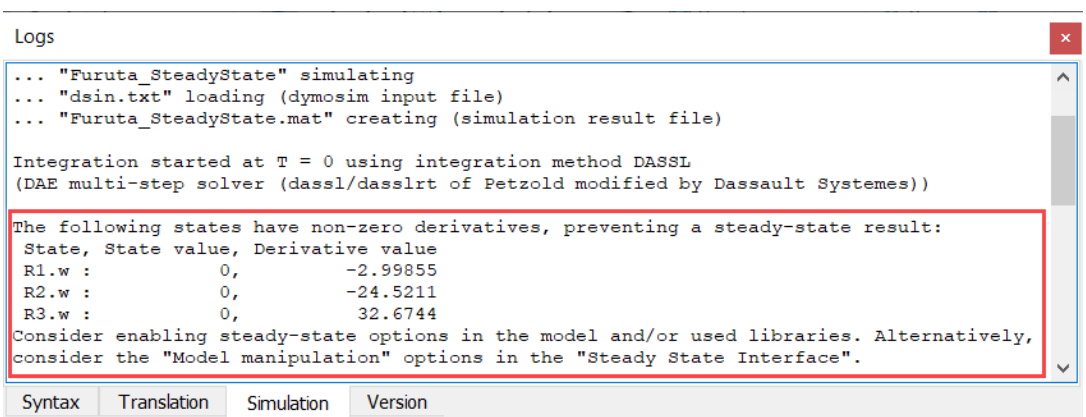
The algorithm Dassl is also forced. During purely static simulation, the integration algorithm isn´t actually called, but selecting Dassl in Dymola enables better-optimized static simulations.

Important! Selecting this option causes the simulation to be static. However, by itself, it does not generate equations to put the model in a steady state. Modelica libraries often come equipped with steady-state alternatives. Using such options, when available, is preferred.

Alternatively, in addition to selecting this simulation mode, any of the two options **Default steady-state initialization** and **Remove fixed = true from contionuous variables** can be tried. (For more information about these options, see the group "Model manipulation" below.) There is no guarantee that these options will actually put the model in a steady state. For example, there may be equations or initial equations in the Modelica model of the type "der(x) = 1". These cannot be automatically removed but must be handled by modifying the model.

Note that (by default) after a static simulation a posteriori check is made to determine if a steady state was actually achieved. See the option **Fail if steady state is not achieved** in the group "Detection and verification" below.

If a steady state is not reached, a message is given in the simulation log, telling which state derivatives that were far from zero. An example:



The flags corresponding to this selection are:

```
Advanced.Simulation.SteadyState = true
Advanced.Simulation.SteadyStateTermination = false
```

Setting `Advanced.Simulation.SteadyState = true` marks the simulation result file as steady-state and activates the logging of states that have derivatives that are not zero.

For the second flag, see the next steady state alternative below.

In addition, when selecting this steady-state alternative, the flag `Advanced.Simulation.FailIfSteadyStateNotReached` is by default set to `true`.

Dynamic steady-state finding

When enabling this option, simulations will stop as soon as a steady state is reached (for more about the condition for this, see below). Unless the end time is reached first, in which case the simulation is failed (by default). When simulating with this option, Dymola will also tag the result as a steady-state result. See the group "Result and plot options" below.

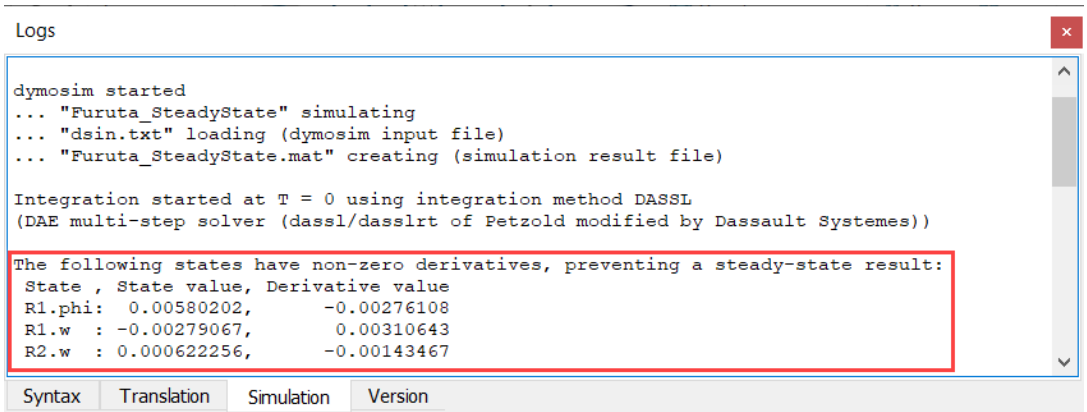
You can set the end time of the simulation by the option **Stop time** in the simulation setup, or the command **Simulation > Stop**.

Finding a steady state by dynamically simulating to it can be a good option:

- When the static steady-state is difficult to formulate, e.g. due to initial equations; or
- When the static steady-state problem is difficult to solve, e.g. due to the nonlinear solver not converging

However, dynamic steady-state finding is usually slower than static steady-state findings, and requires that the model be properly damped.

If steady state is not reached, a message is given in the simulation log telling which state derivatives that were far from zero. An example:



The flags corresponding to this selection are:

```
Advanced.Simulation.SteadyState = true
Advanced.Simulation.SteadyStateTermination = true
```

Setting `Advanced.Simulation.SteadyState = true` marks the simulation result file as steady state and activates the logging of states that have derivatives that are not zero.

Setting `Advanced.Simulation.SteadyStateTermination = true` tells the integrator to stop as soon as a steady state has been reached.

In addition, when selecting this steady-state alternative, the flag `Advanced.Simulation.FailIfSteadyStateNotReached` is by default set to `true`.

(Formally the test if steady state is reached for each state x_i is:

```
|der(x_i)| <= tolerance * (|x_i| + |nominal_i|) / 2.0
where
..tolerance = Advanced.Simulation.SteadyStateTerminationTolerance
if this flag is set, or
  tolerance = 0.02 / min(stopTime – startTime,
500*interval_length). When "Strict" is selected, 0.02 is changed to
0.0001.
```

For unbounded variables, the term |x_i| is remove from the test.

Note that the feature is implemented by checking if the state derivatives are close to zero. This means that the following cannot be detected:

- Periodic steady states
- Steady states where a subset of the states are still varying)

Model manipulation (requires retranslation)

The following two options may help a simulation to initialize in a steady state. It is most useful for static steady-state finding. Note that these options do not guarantee that a steady state will be achieved; cf. the posteroiri check **Fail if steady state is not reached**. Also, consider using any steady-state option in the model or used library.

Default steady-state initialization

When this option is enabled, each state x that do not have a fixed start value will be assigned the steady-state condition `der(x) = 0`. You can check the translation log to see which state derivatives are not put to zero.

The option is by default not active. This corresponds to the flag `Advanced.DefaultSteadyStateInitialization = false`.

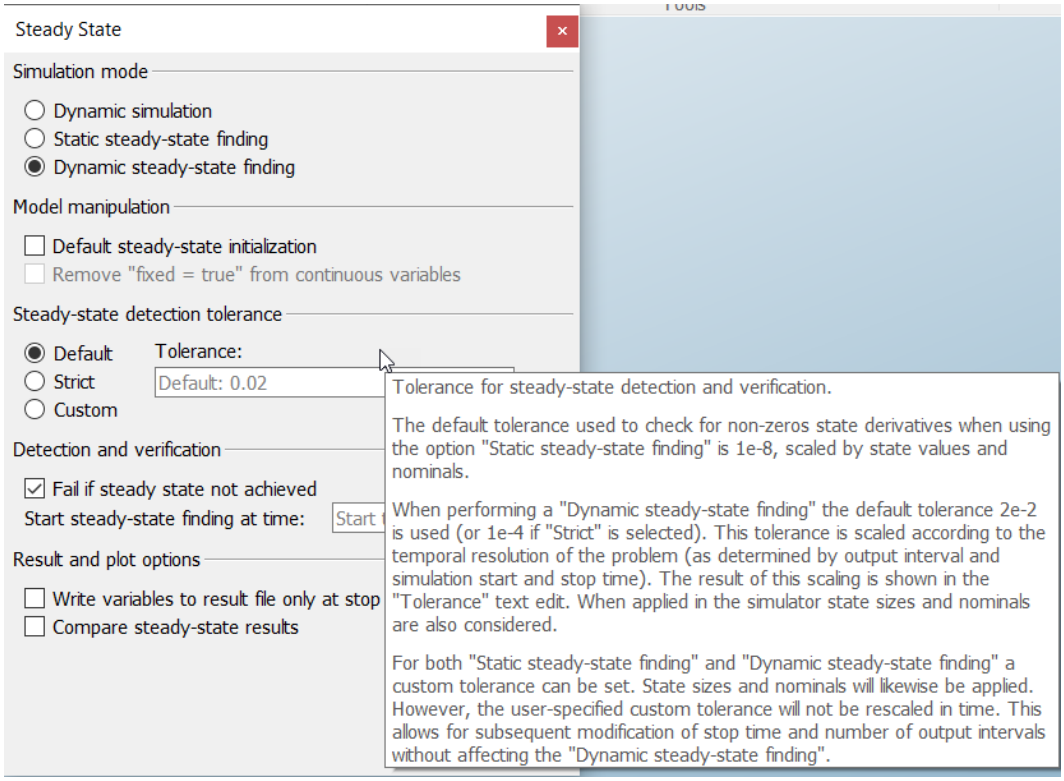
Remove fixed = true from continuous variables

This option is only available if the previous option is activated. Activating this option removes `fixed = true` from contiuous variables. The default is to keep them. By enabling this option, more variables are assigned the steady-state condition `der(x) = 0` as fixed initial conditions are ignored. Note that initial conditions are still in effect and may cause a non-steady-state static simulation. By default, any such simulation will be failed by the a posteori check discussed in **Fail if steady state is not achieved** below.

The option corresponds to setting the flag `Advanced.Translation.DefaultSteadyStateInitializationLevel = 1`.

Steady-state detection tolerance

Three options are available for steady-state detection tolerance. Please note the extended tooltip for this group:



Default

When performing a **Static steady-state finding**, the **Default** tolerance used to check for non-zeros state derivatives is 1e-8. This value can be seen in the **Tolerance** field, preceeded by the text `Default:`. During simulation, the value is scaled by the state variables (the size of the states) and nominals.

When performing a **Dynamic steady-state finding**, the **Default** tolerance used to check for non-zeros state derivatives is 2e-2 (or 1e-4 if **Strict** is selected, see next option). The tolerance is then scaled according to the temporal scale of the problem (derived from the output interval and the simulation start and stop time). This allows for quick enabling of **Dynamic steady-state finding** out-of-the-box. The result of this time scaling can be seen in the **Tolerance** field, preceeded by the text `Default:` or `Strict:` to indicate if **Default** or **Strict** tolerance has been selected. During the simulation, also the size of states and their nominal are scaling the value, for both selections.

Strict

When you perform a Dynamic steady-state finding, you have an option **Strict** to use a stricter tolerance. As described in previous section, the default tolerance of **Strict** is 1e-4. Selecting **Strict** corresponds to setting the flag

```
Advanced.Simulation.SteadyStateTerminationStrict = true
```

Changing to this strict tolerance does not require re-translation. It also works with FMI export.

Custom

For both **Static steady-state finding** and **Dynamic steady-state finding**, you can set a **Custom** tolerance. During simulation, also the size of states and their nominals are scaling the value, but the value is not rescaled in time. This allows for subsequent modification of stop time and number of output intervals without affecting the **Dynamic steady-state finding**. For example, it allows for setting a large stop time to minimize the risk of hitting the stop time before a steady state is reached.

The custom tolerance can be seen and changed in the **Tolerance** field, with no preceding text. The value corresponds to the value of the flag `Advanced.Simulation.SteadyStateTerminationTolerance`. (If custom tolerance is not used, which is the default, the value of the flag is 0.)

Notes:

- Using and changing the custom tolerance requires re-translation.
- Custom tolerance also works for FMI export.
- If you click the **Tolerance** field and start entering a value, you automatically change the tolerance mode to **Custom**.

Detection and verification

Two options are available for detection and verification, which of them are available depend on the selection of the simulation mode.

Fail if steady state is not achieved

This option can only be used if any of the steady-state modes (**Static steady-state finding** or **Dynamic steady-state finding**) is selected. By default, this option is activated for these alternatives. For Static steady-state finding, a check is made after the simulation is done. If any of the state derivatives is non-zero the simulation is failed. For Dynamic steady-state finding, the simulation is failed if the stop time is reached before a steady state could be detected.

The feature corresponds to the flag `Advanced.Simulation.FailIfSteadyStateNotReached = true`.

Start steady-state finding at time:

This option can only be used for the **Dynamic steady-state finding** mode. This option sets a start time for the dynamic steady-state finding. That is, Dymola will not start to look for a

steady state during simulation until the specified time is reached. This can be useful if a model starts in steady state, but an event triggers dynamic behavior at a time later than the start time of the simulation. By default, Dymola will start the steady-state detection from the beginning of the simulation. The setting corresponds to the flag `Advanced.Simulation.SteadyStateTerminationStartTime`.

Result and plot options

Two result and plot options are available:

Write variables to result files only at stop time

This option can be used to minimize the size of the result file if the model is large and only the result at stop time is needed. Notes:

The output points, as defined in the **General** tab of the simulation setup, are still used internally by the integrator.

This operation can be seen as a complement to variable selection.

The option is also available from the **Output** tab of the simulation setup.

Compare steady-state results

This option compares the results opened in the variable browser. This is similar to the existing Compare results in the variable browser, but plots signals over simulation number. This option is deactivated by default. If activated, it is only applied when plotting results of a steady-state simulation (**Static steady-state finding** or **Dynamic steady-state finding**). As an example, consider the simple linear model with one state x.

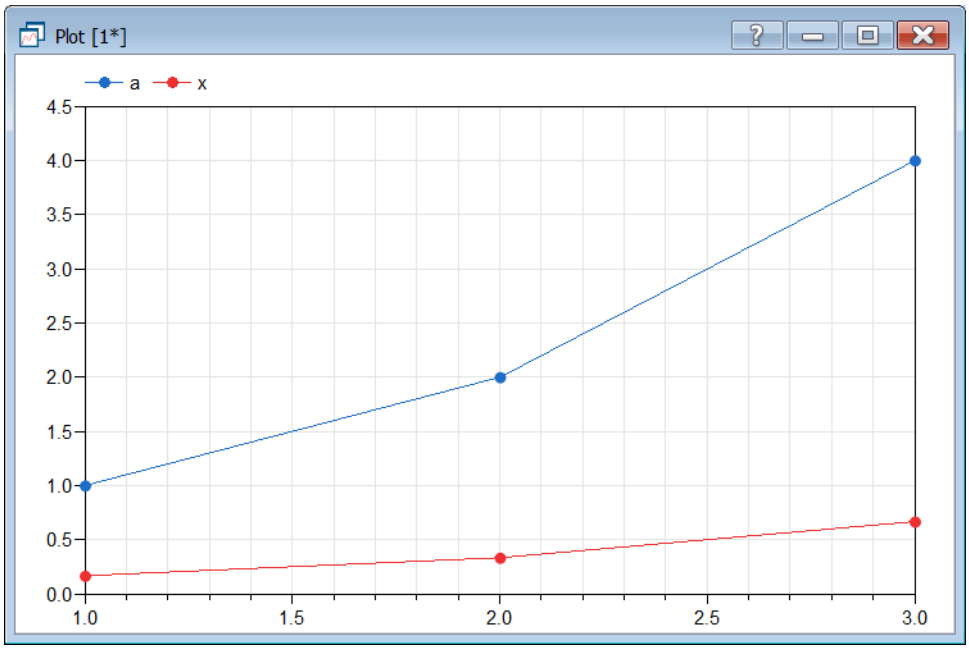
```
model LinearEquation
Real x(start = 0.1);
parameter Real a = 1.0;
parameter Real b = -6.0;
equation
der(x) = a + b*x;
end LinearEquation;
```

First select **Static steady-state finding** and activate the option **Default steady-state initialization**. (Or just activate **Dynamic steady-state finding**, or even alternate between the two options.) Then enable **Compare steady-state results**.

Perform three simulations for different values of the parameter a. Notes:

- To see all three simulations in the variable browser, you can change the value Number of results to keep to 3.
- The above means that you first simulate with a=1.0, and then, in the variable browser, change a to 2.0 and simulate again, and finally change a to 4.0 and simulate again.
- Perform no sweeping.

Now, you can expand any of the three result files and plot a and x. This will give the following plot:



The three steady states are compared. For dynamic steady-state finding, the time variable is hidden and only the result of the final time (the steady state) is plotted. This means that the results are plotted the same way regardless if they come from a static steady-state finding or from a dynamic steady-state finding.

5.8.6 Steady state initialization

General

Steady state initialization can be easily achieved in Dymola by setting state derivatives to zero (see, in the chapter "Introduction to Modelica", section "Initialization of models"). This can either be done explicitly in the model using initial equations or it can be done by setting the flag

```
Advanced.DefaultSteadyStateInitialization = true
```

discussed below.

For pure steady state simulations, where the dynamic behavior of a model is of no interest, it is recommended to set the simulation end time equal to the simulation start time for faster simulation. Note that it is also recommended to use any of the solvers Lsodar, Dassl, Euler, or any Rkfix solver; those solvers give the fastest simulation in this case. See also the chapters "Model Experimentation" and "Model Calibration" for more functionality empowering experimentation with steady state simulations.

Default steady state initialization

When starting a simulation, the states must be initialized. This requires additional information beyond the differential equations that describe the dynamics. The additional information can be provided in two ways:

- Start values for variables defined with `fixed=true;`
- Initial equations and initial algorithms

See, in the chapter "Introduction to Modelica", section "Initialization of models" for details.

If initial conditions are missing, Dymola makes automatic default selection of initial conditions. The approach is to select continuous time states with inactive start values and make their start values active by turning their fixed attribute to true to get a structurally well-posed initialization problem. A message informing about the result of such a selection can be obtained.

The logic for automatic selection of default initial conditions can be changed by setting the flag:

```
Advanced.DefaultSteadyStateInitialization = true
```

(The flag is by default `false`.) With this flag activated the initialization problem is instead completed by setting state derivatives to zero, effectively adding initial equations like `der(x)=0`. Consider, for example, the simple model:

```
model DefaultSteadyStateInitialization
  Real x(start=1);
equation
  der(x) = 8-x^3;
end DefaultSteadyStateInitialization;
```

If the flag `Advanced.DefaultSteadyStateInitialization` is false, the continuous state x is initialized to its prescribed start value 1. If the flag is true, then instead the initial equation `der(x)=0` is added, resulting in the initial value 2 for x.

Note that the flag `Advanced.DefaultSteadyStateInitialization` does not change any initial conditions that are already defined in the model. Therefore, setting the flag does not automatically mean that the whole model is initialized in steady state. For example, in the above example model, consider the case when the start value of x is defined with `fixed=true`:

```
Real x(start=1, fixed=true);
```

The variable x will be initialized to the value 1 regardless of the value of the flag `Advanced.DefaultSteadyStateInitialization`. This flag only affects the process of completing an incomplete initialization problem by automatic initial conditions selection.

However, to make it easier to apply the above flag to models that were not intended for steady-state initialization, you also set the flag

```
Advanced.Translation.DefaultSteadyStateInitializationLevel = 1;
```

This means that all start values in the model are ignored. In general, `Advanced.Translation.DefaultSteadyStateInitializationLevel` means that all

start values at that level or higher are ignored; where start-values directly in the actual model have level 1, and start-values directly in component models have level 2. (The default value of the flag is 0, meaning that the flag is not used.)

Undetermined square systems

For some models the steady state initialization problem is underdetermined even though enough initial conditions have been specified according to to the description in the chapter "Introduction to Modelica", section "Initialization of models", subsection "Continuous time problems" (explicitly and/or by default selection). That is, even though there is the same number of unknown variables as there are initial conditions. This can, for example, happen in closed fluid loops, that could initialize at a range of pressure values.

In these cases, Dymola will try to automatically localize and handle the issue by removing unnecessary equations and complementing the steady state initialization problem using non-steady state start values. A warning message is given in the translation log with details on which start values are used.

To choose between non-steady state start values Dymola uses the normal start value level priorities (as for non-steady state initialization), see section "Discriminating start values" starting on page 1024.

Using the flag for default steady state initialization in models

As describe above Dymola can automatically handle many difficult issues arising when doing steady state initialization. However, to enable more control to the user there is also the built-in function `steadyStateInitialization()`. It returns the value of the flag `Advanced.DefaultSteadyStateInitialization`.

This built-in function enables using the value of the flag in the Modelica code, which simplifies the programming of steady state thandling. An example of a steady state initialization code section could be:

```
initial equation
   if steadyStateInitialization() then
         der(x)=0;
   else
         x=x_start;
   end if;
```

In general, this built-in function can be used in models to introduce special settings that are only enabled during steady state simulations. Applications include loop breaking components, which may only be needed during steady state initializations.

6 MODEL EXPERIMENTATION

6 Model Experimentation

6.1 Introduction

Dymola provides the Experimentation package as a feature of the Design package. The main purpose of this package is to allow the user to vary parameters of the system to get an intuitive knowledge of the behavior of the model. Some of the functionalities of this package are related to other functions of the Calibration package. Please see the chapter "Model calibration" in this manual.

The main difference is that those are coupled to the calibration setup, while the functions in Experimentation are independent and can be used to illustrate phenomena of the system. One of the functionalities of Experimentation package is essentially different: Monte Carlo simulation.

Important!

In Dymola 2019 FD01 the GUI for sweeping parameters was improved. The new way of sweeping parameter is described in the first section below: 6.2 "Sweeping parameters using new GUI". Note that when sweeping this way you don´t need to open the Design package.

The explicit use of the Design.Experimentation package, including the old GUI for sweeping parameters, that still can be used, is described starting from section 6.3 "Varying parameters of a model" starting on page 1064, and the following sections in this chapter.

To increase simulation speed, all functions in the Design.Experimentation package, and the command **Simulation > Sweep Parameters** for the new GUI, by default run on multiple cores on the CPU. By default all cores are used, but if needed the number of parallel runs can be specified by the flag

```
Advanced.ParallelSimulations
```

The default value is 0, meaning using the number of available cores.

6.2 Sweeping parameters using new GUI

6.2.1 General

The new GUI for sweeping parameters has the following advantages compared with the old, still applicable, way described in later sections of this chapter:

- **Easier selection of signals:** In a dockable window "Sweep Parameters" you specify what signals may be used for sweeping, by, for example, dragging them from the Variable Browser
- **Easier configuring of each signal, making it easy to change what signals to use, and how, for a sweeping:**
    - In the dockable window, you can for each signal select if it should be swept, plotted or not used by the moment. You can also, for each signal, select to remove it.
    - For signals selected for sweeping, you can, in the same window, for each signal, click a button to specify values for a grid.
- **Easier integration setup:** The integration setup is now by default the current simulation setting, you don´t have to redo it for the sweeping.
- **Easier to switch between plotting trajectories and plotting endpoints:** The dockable window has one button for each type of sweeping, just to click one or the other.
- **Easier to switch between sweeping one parameter, two parameters, or more parameters:** The GUI is the same; the difference is only that depending on your selection, different built-in functions are called when executing your selection.

Notes:

- The functionality of the package Design.Experimentation is not changed, there are no changes in the built-in functions used, and you can actually also still use the previous way to setup sweepings (described later in this chapter).
- If you sweep parameters manually, you may not want each new simulation result file that you get to be automatically kept in the Variable Browser. You can control this by the setting **Keep result files when signal is plotted** in the plot setup, reached by the command **Plot > Setup**, the **Options** tab.

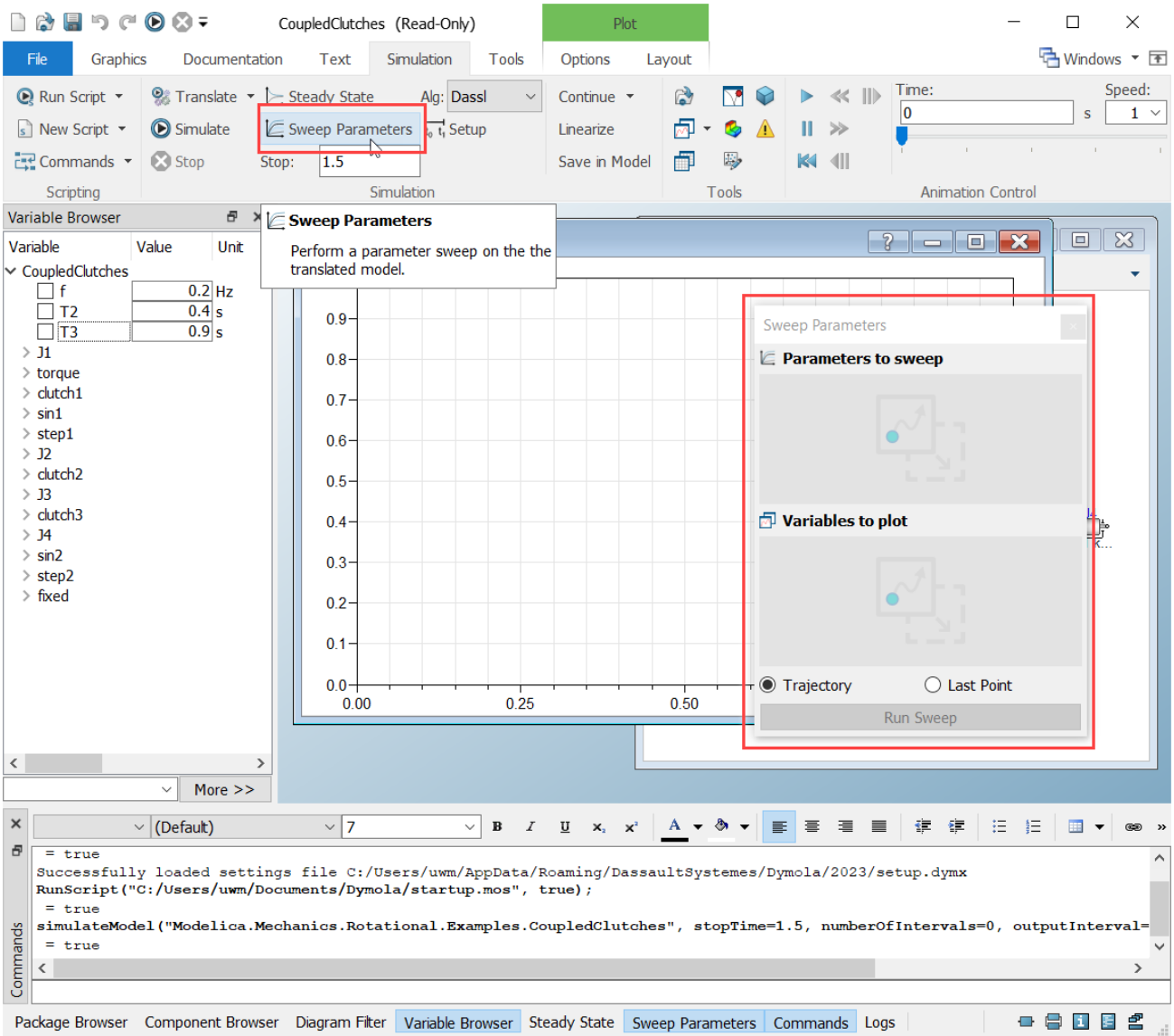
6.2.2 Sweeping one parameter

This example shows sweeping, with the new GUI, the parameter J1.J to see how J1.w and J2.w are changed in the demo Coupled Clutches.

Open the demo Coupled Clutches by the command **File > Demos > Coupled Clutches**. Simulate the model by the command **Simulation > Simulate**.



To open the dialog for sweeping, select **Simulation > Sweep Parameters**. This will open a Sweep Parameters window for selecting variables:

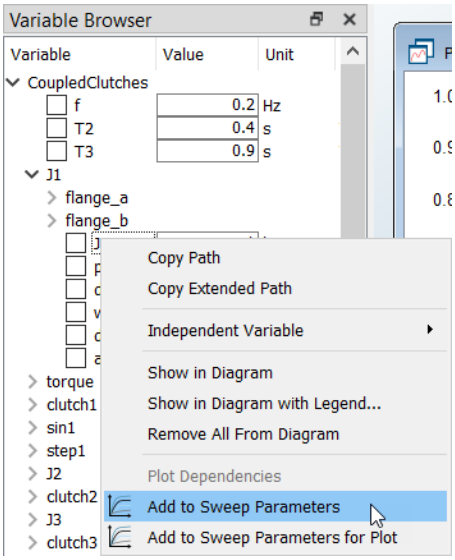


(You can also display the window by the **Windows > Docked Windows > Sweep Parameters** command.)

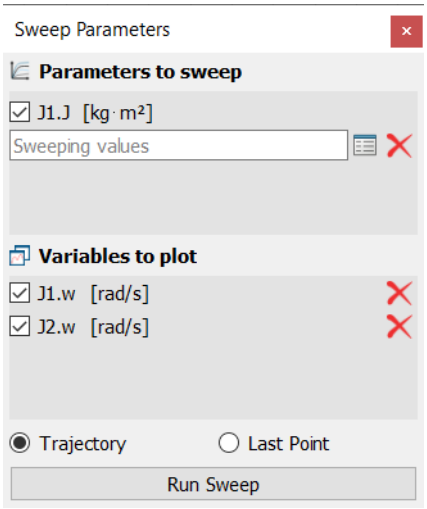
The Sweep Parameters window has two panes, one for parameters to sweep and one for variables to log.

We want to sweep the parameter J1.J and log J1.w and J2.w.

In the Variable Browser, drag J1.J from the Variable Browser to the Sweep Parameters window, to the **Parameters to sweep** pane. (You can also right-click the signal in the Variable Browser and select **Add to Sweep Parameters** to add it in to the pane **Parameters to sweep** in the Sweep Parameters window.)

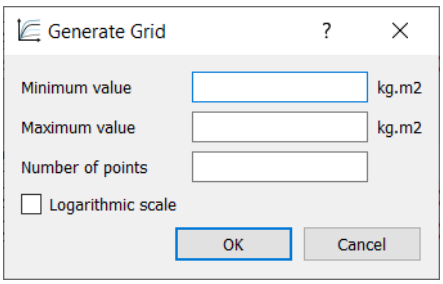


Do the same For J1.w and J2.w, drag them to the **Variables to plot** pane in the Sweep Parameters window, or right-click them in the variable browser and select **Add to Sweep Parameters for Plot**. The result will be:



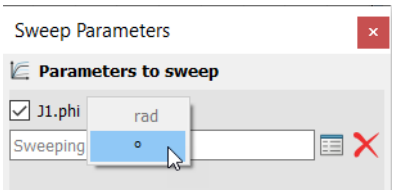
For each signal in the **Parameters to sweep** pane, you have the following alternatives/options:

- **Generate points in an interval** – click to display the following dialog:



  The values are empty be default, units are however displayed. The display units can be changed, if possible. Alternatively, and array can be typed into the corresponding textbox that is enabled in the Sweep Parameters window for that signal. Note the option to generate a grid with logarithmic scale by activating **Logarithmic scale**.

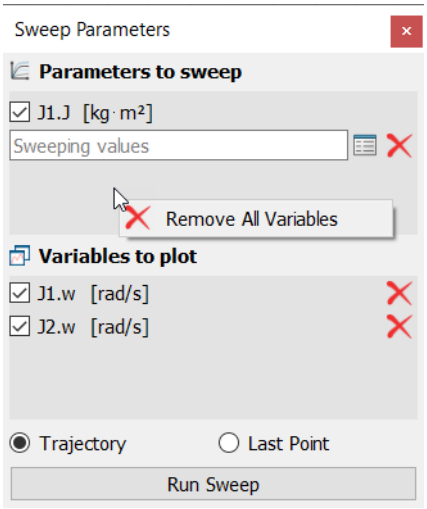
- If possible, you can change display unit of both swept parameters and plotted variables, an example of the former:



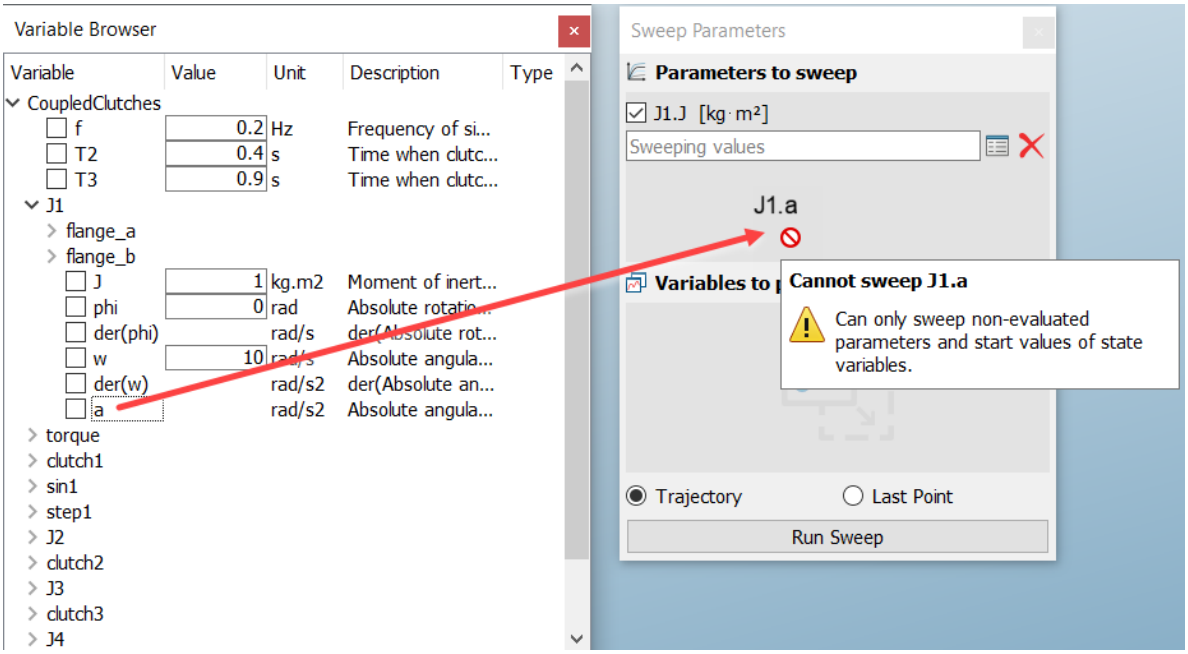
- **Remove variable** – click to remove this variable from the **Parameters to sweep** pane.

For each signal in the **Variables to plot** pane, you can click **Remove variable** to remove this variable from that pane.

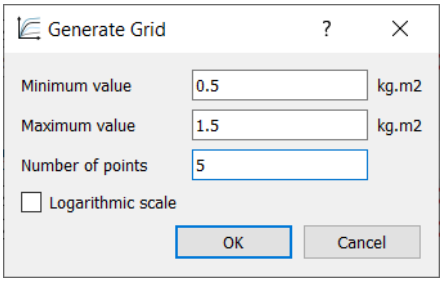
To remove all variables from both panes, you can right-click anywhere in the Sweep Parameter window and select **Remove All Variables**.



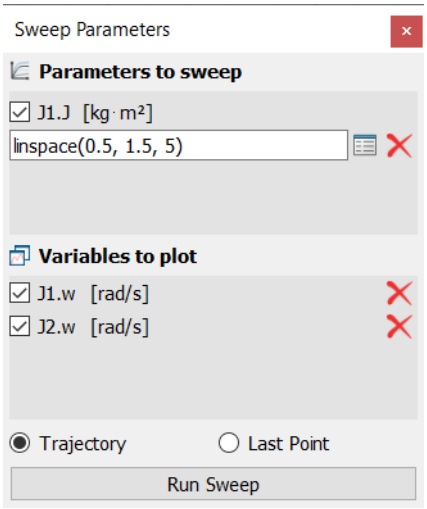
Note that if sweeping has no meaning for a variable, you are not allowed to add it in the **Parameters to sweep** pane. For example, trying to add J1.a for sweeping will give:



To continue with the example, for J1.J, click **Generate points in an interval** and specify, in the dialog, 0.5, 1.5, and 5:

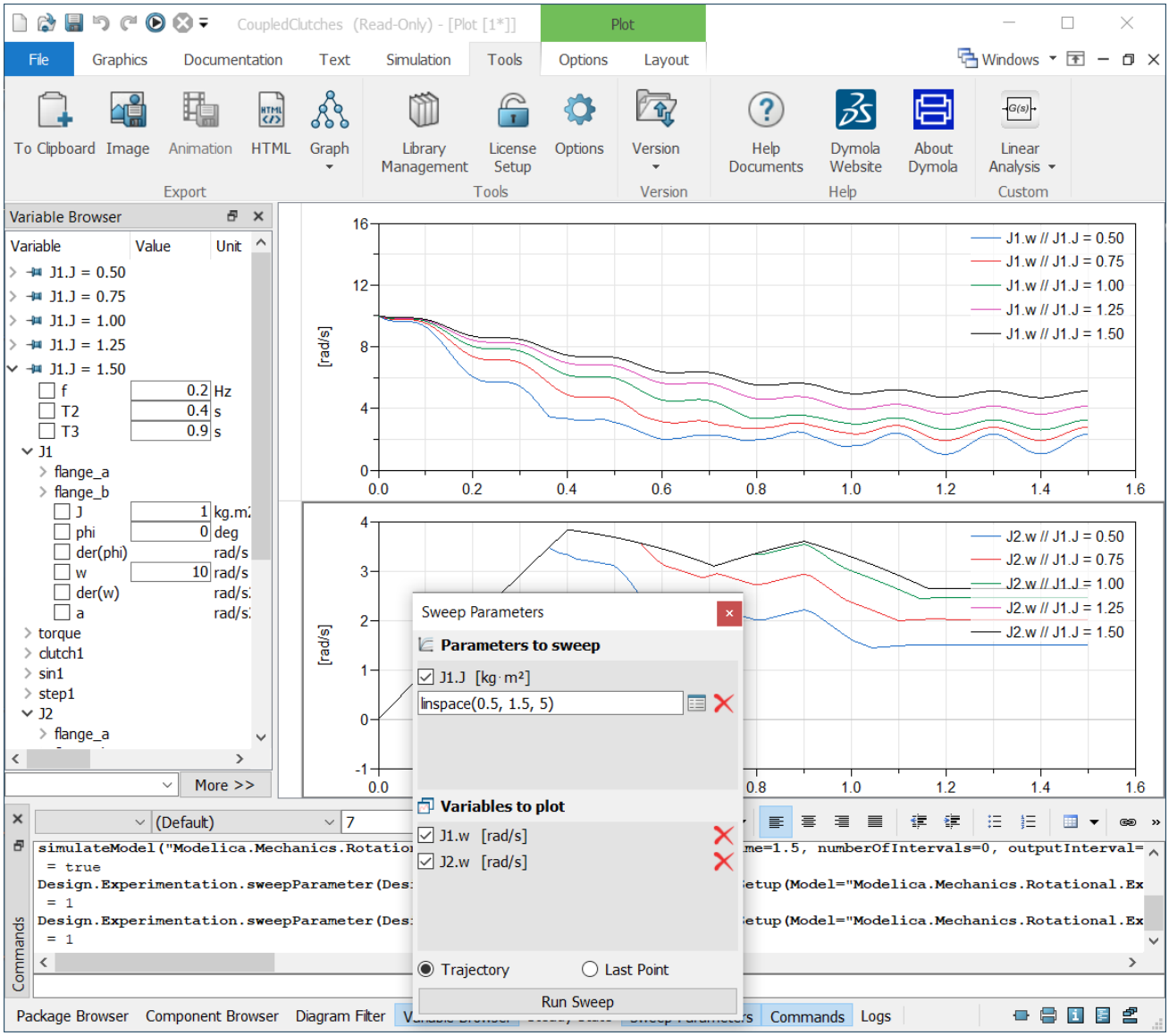


The resulting dialog is:

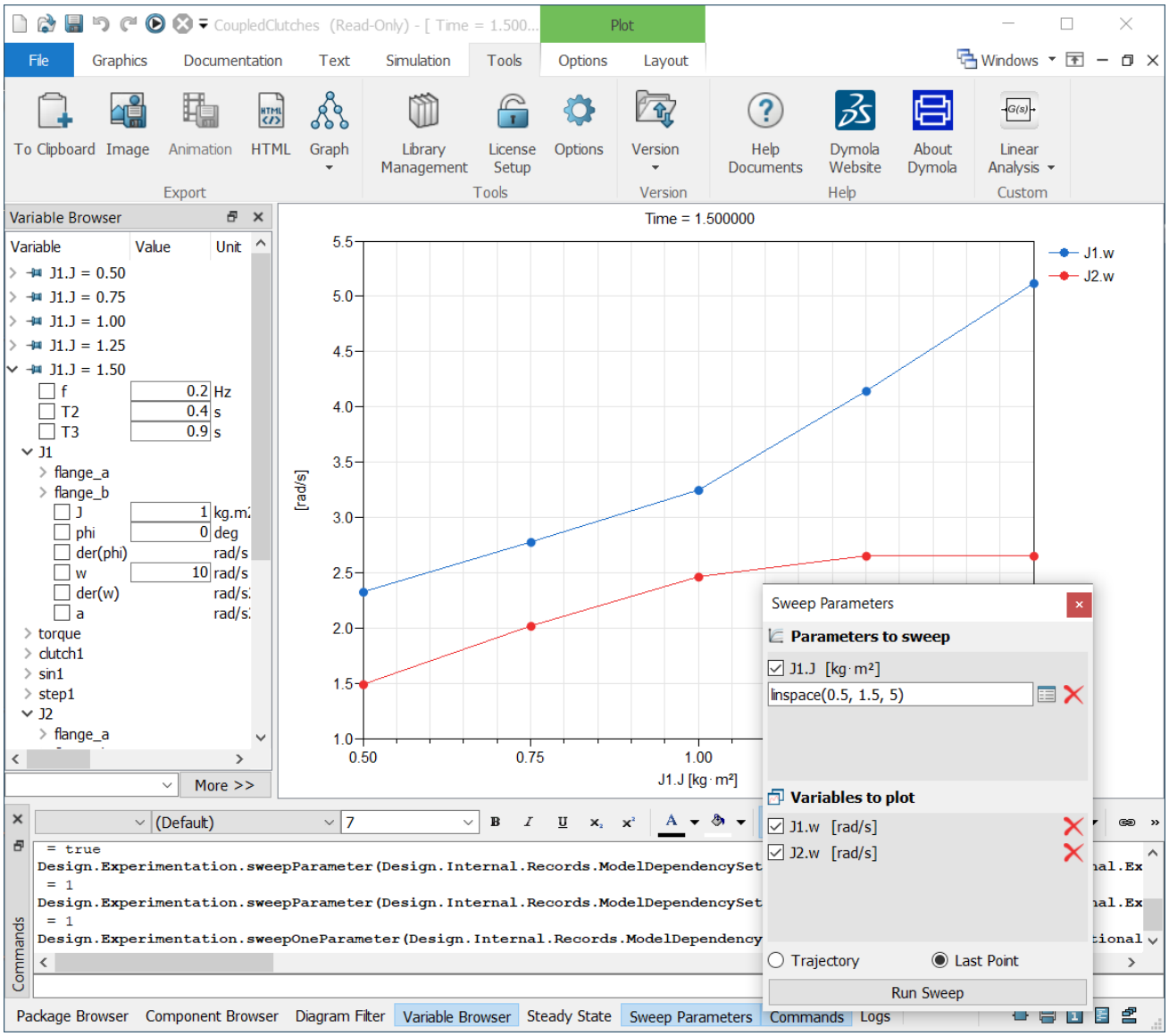


Now, before clicking **Run Sweep**, select if trajectories should be plotted, or if last points, by the command buttons **Trajectory** and **Last Point**.

Selecting the default **Trajectory** and then clicking **Run Sweep** gives:

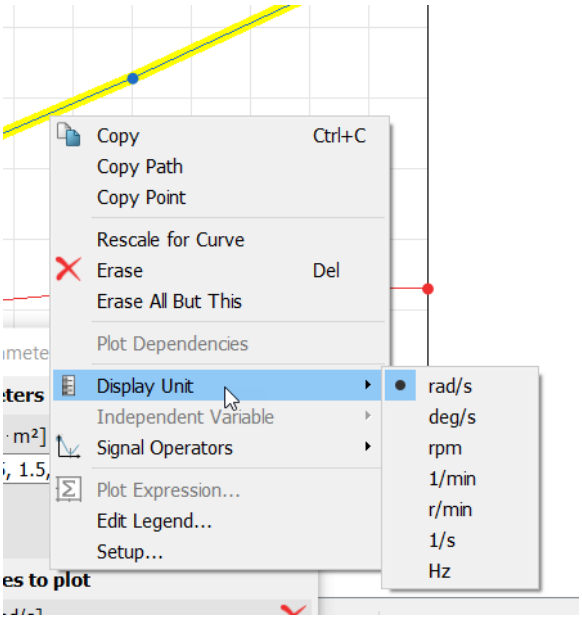


Selecting instead **Last Point** and then **Run Sweep** gives:

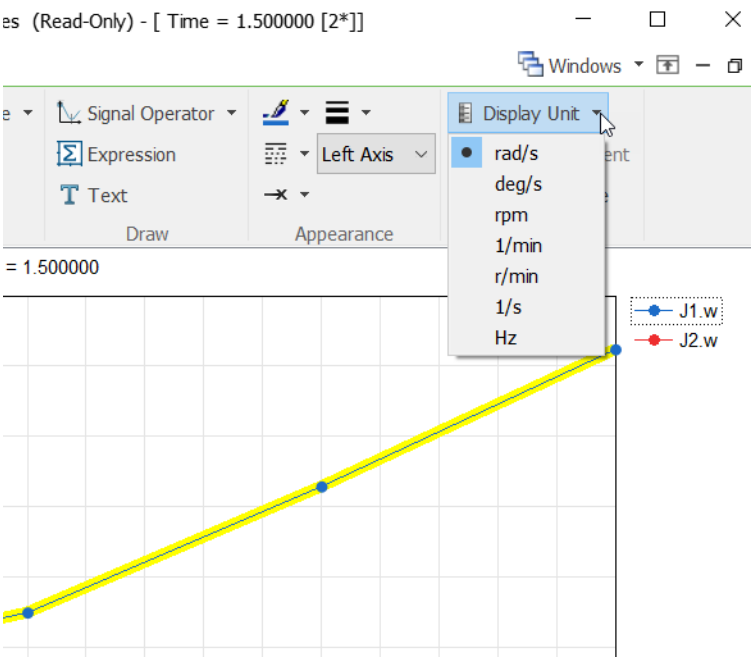


You can change the display units after the sweep, both for the variables plotted and the parameter swept.

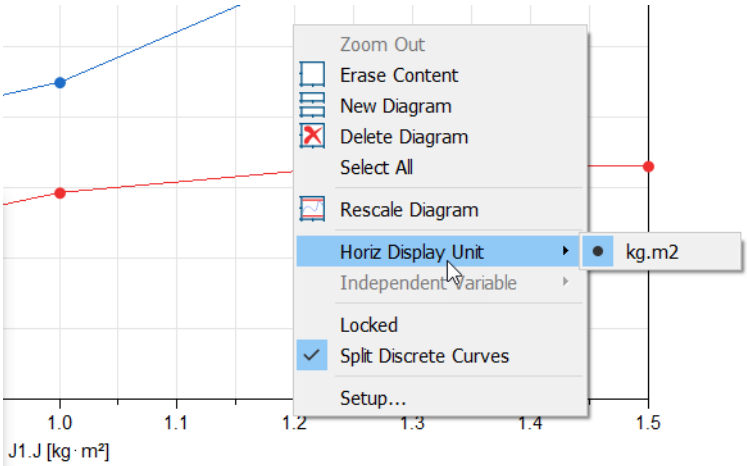
To change the display unit for the variables plotted, you can right-click the corresponding curve and select **Display Unit**:



As an alternative, you can use the **Display Unit** button in the ribbon:



To change the display unit of the parameter swept, you can right-click an empty space in the plot, and select **Horiz Display Unit**:



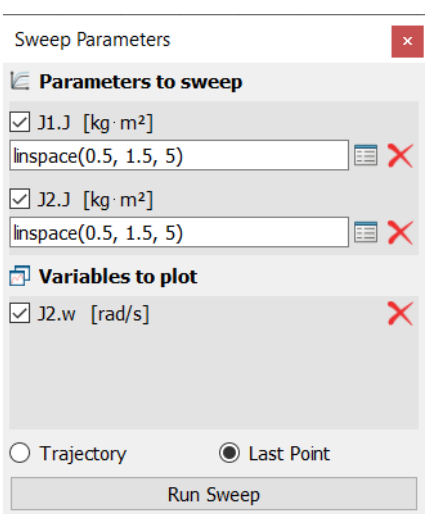
6.2.3 Sweeping two parameters

This example shows sweeping, with the new GUI, the parameters J1.J and J2.J to see the last point of J2.w for all combinations of J1.J and J2.J, in the demo Coupled Clutches.

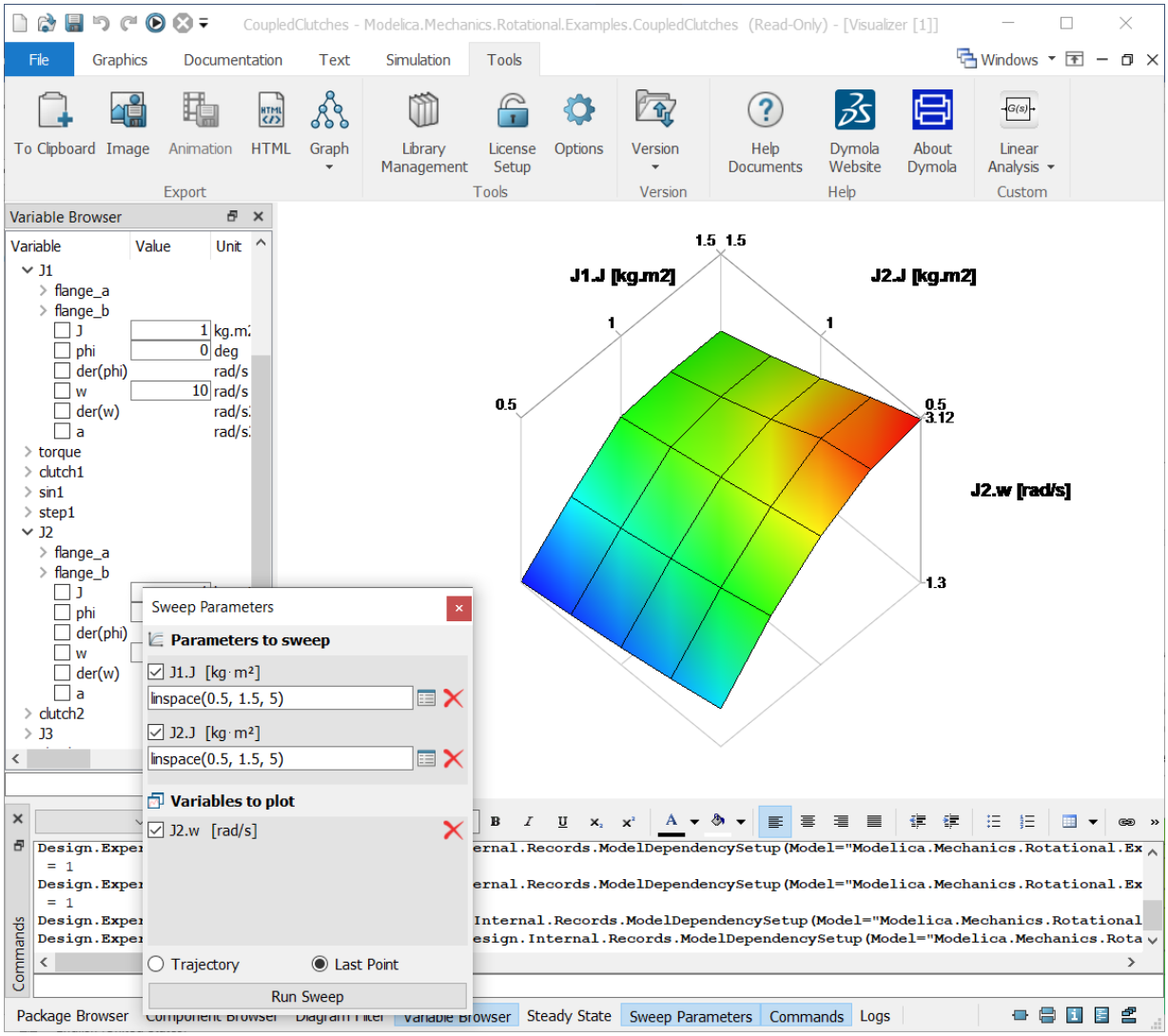
The easiest way to run this example, given you have run the previous example, is to just:

- Select **Last Point**.
- Drag the missing J2.J from the Variable Browser to the **Parameters to sweep** pane and enter values for equidistant grid like for J1.J.
- Delete J1.w from the **Variables to plot** pane.

The resulting dialog is:



Now select **Run Sweep**. The result is:

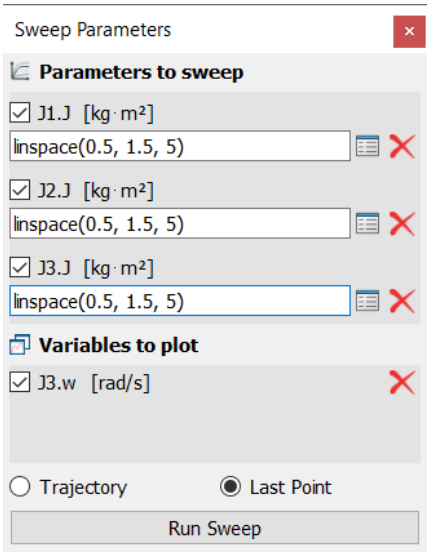


Note: You cannot change any display unit after the sweep.

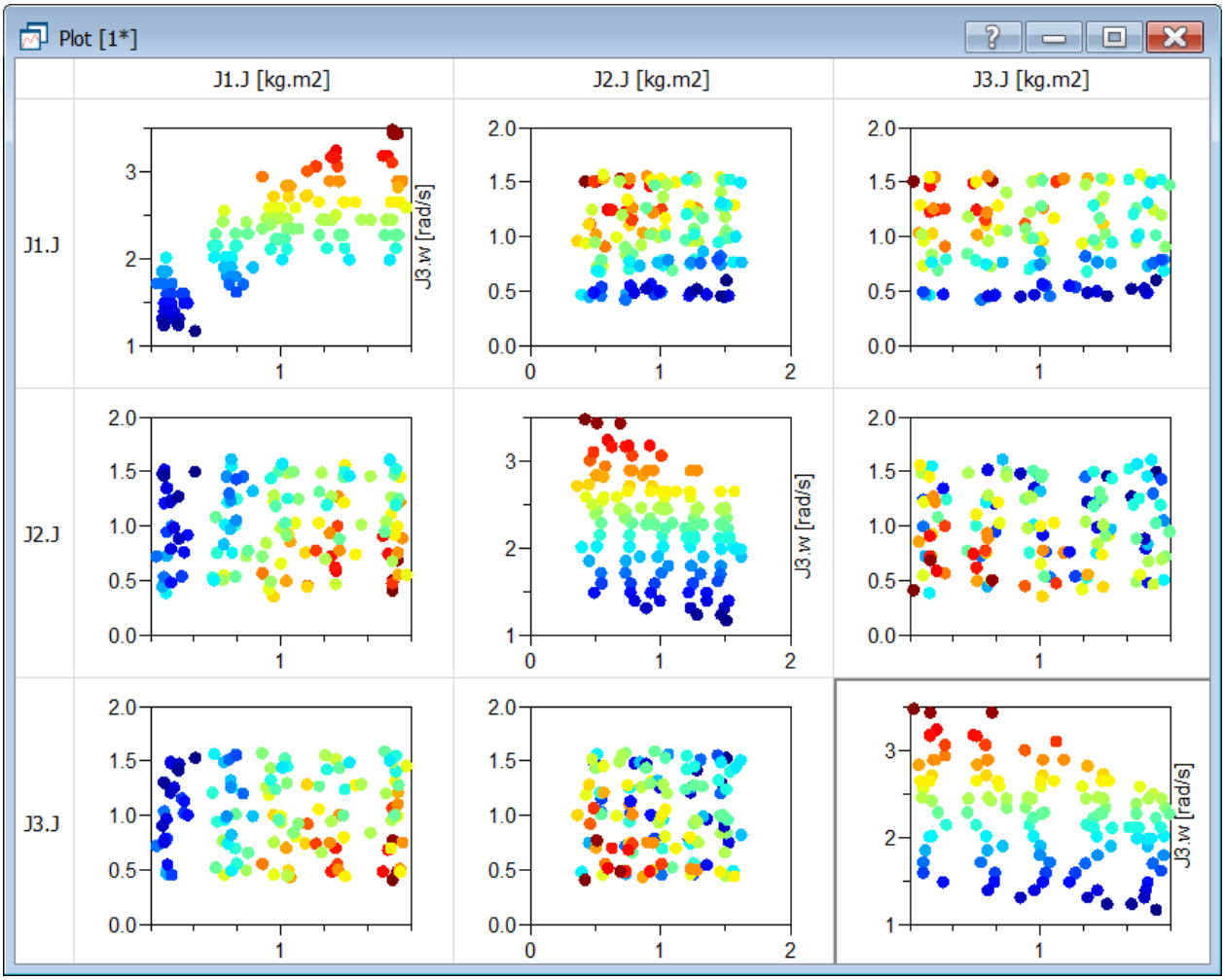
6.2.4 Sweeping more than two parameters

The following example illustrates sweeping three parameters in the Coupled Clutches model.

Configure the sweep setup like:



Performing the sweep, you will get the following scatter plot:



The problem we want to solve is to understand how variations in many parameters explain the variation in the output variable, and it´s not simply explained by changes in individual variables. Thus, we take the next step and see whether pairs of variables explain the variation in the "output" variable. Note that the dots are for sweeps a bit randomized – because they would otherwise overlap.

That explains the non-diagonal entries in the matrix of scatter plots, and that part is symmetric (the pair J1.J, J2.J behave the same as J2.J, J1.J). The output value is given as the color of the dots, and since we sweep parameters, the scattering of the points does not provide any useful information. (Compare with the matrix of colored scatter-plots on https://plotly.com/python/splom/ (scroll down a bit for color ones).)

There are two ways of using that information:

- Select one parameter and see which parameter combines with it to best explain the output.

- See which pair-wise combination overall are the most important.

Using the same logic for the diagonal entries would just be uninteresting straight lines since it is the same parameter on both axis; and normally a matrix of scatter-plots either leaves the diagonal empty or includes something else.

In this case, it is just a color-coded plot for the output variable against the parameter; there are two ways of using this information:

- It shows how much variations in that parameter explain the variation in the output variable – if the other parameters had no impact it would just be a curved line – and the other parameters would turn it into a "fatter" line.
- As reference for the color-scale of the output variables.

General references:

https://www.itl.nist.gov/div898/handbook/eda/section3/eda33qb.htm

Note: Since it is based on sweeps of parameters (not observations) we cannot see outliers or clustering; just the relationships between variables.

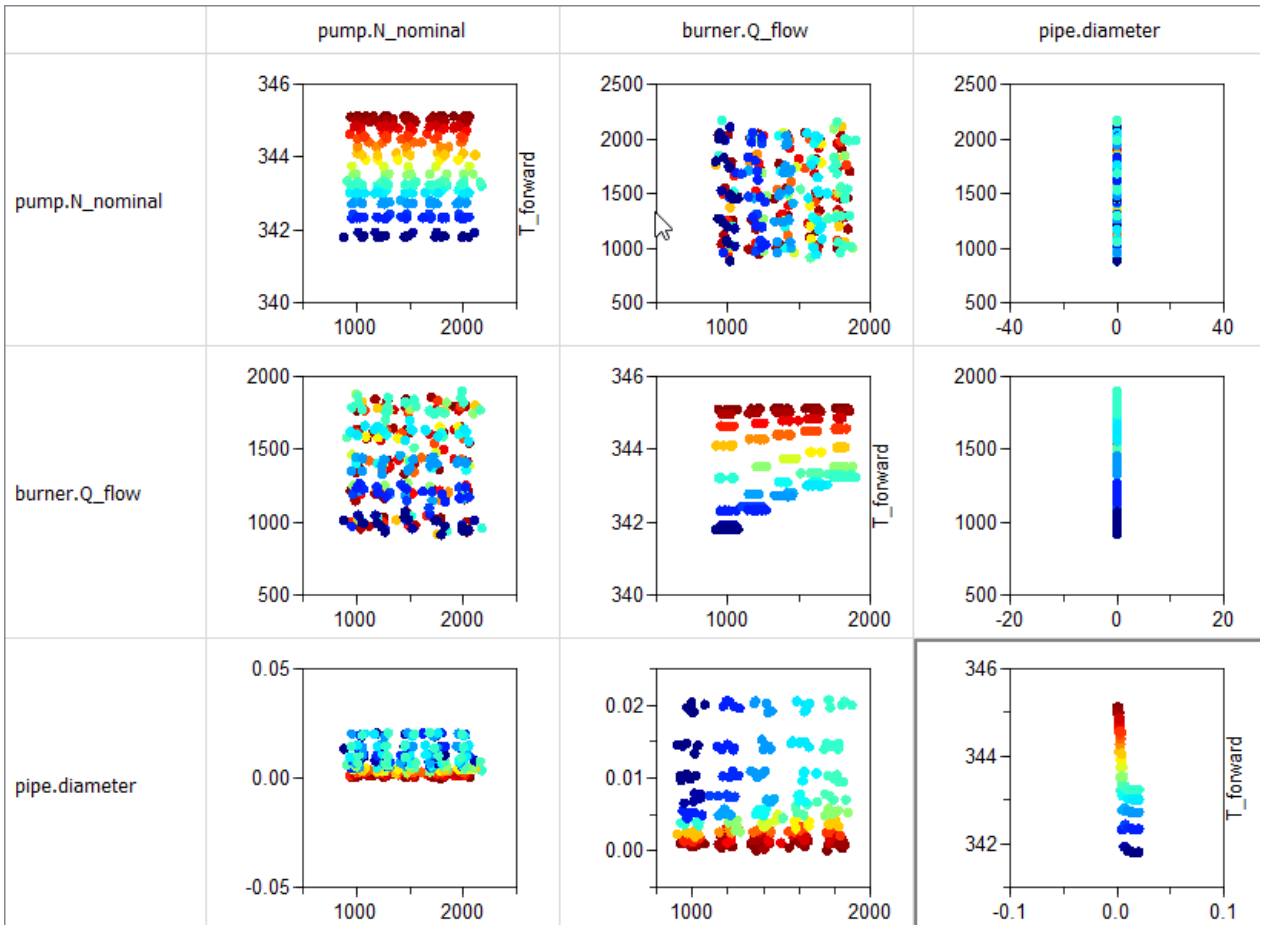
https://plot.ly/python/splom/

Another simple example could be:

```
Design.Experimentation.sweepManyParametersScatter(Design.Internal.Records.
ModelDependencySetup(Model="Modelica.Fluid.Examples.HeatingSystem",
dependencyParameters={Design.Internal.Records.DependencyParameter(name=
"pump.N_nominal", values=linspace(1000, 2000, 5)),
Design.Internal.Records.DependencyParameter(name="burner.Q_flow",
values=linspace(1000, 1800, 5)),
Design.Internal.Records.DependencyParameter(name="pipe.diameter",
values=exp(linspace(log(0.001), log(0.02), 10)))},
VariablesToPlot={Design.Internal.Records.VariableToPlot(name="T_forward")}
, integrator=Design.Internal.Records.Integrator(startTime=0,
stopTime=6000, numberOfIntervals=500, outputInterval=0, method="Dassl",
tolerance=0.0001, fixedStepSize=0)));
```

Note! The model contains a number of protected variables, before sweeping you must, in the simulation setup, the **Output** tab, the **Store additional variables** group, activate **Protected variables**. If this setting is dimmed, you first have to, in the **Debug** tab, the **Simulation analysis** group, clear **Provide variable dependencies and equation incidence for plotting**. (The simulation setup can be reached by the command **Simulation > Setup**.)

The result of this sweeping is:



Looking at the third row above, the first plot indicates that the nominal speed for the pump doesn´t matter. The second plot in the same row indicates that for a large pipe-diameter, the burner makes a difference in the temperature, whereas for a very small pipe it doesn´t matter.

Note the possibility to use Monte Carlo analysis rather than sweeping when working with many variables. Monte Carlo is more efficient in such cases. The result of a Monte Carlo analysis can also be presented as color-coded scatter plots. For Monte Carlo analysis in general, see section “Monte Carlo Analysis” starting on page 1082.

Note: You cannot change any display unit after the sweep.

6.2.5 Storing and retrieving a sweep setup in the model

It is possible to store the sweep setup as a command in the model and retrieve it when needed. To illustrate this, the following example shows how to store a sweep on one parameter as a command in a model, how to retrieve it, and finally how to change a stored sweep setup.

Note that this way of storing and retrieving is not limited to the new sweeping GUI; it is a general tool of storing and retrieving function calls that can be used for any suitable function call, for example, other functions in the Design package.

Store the sweep setup in the model

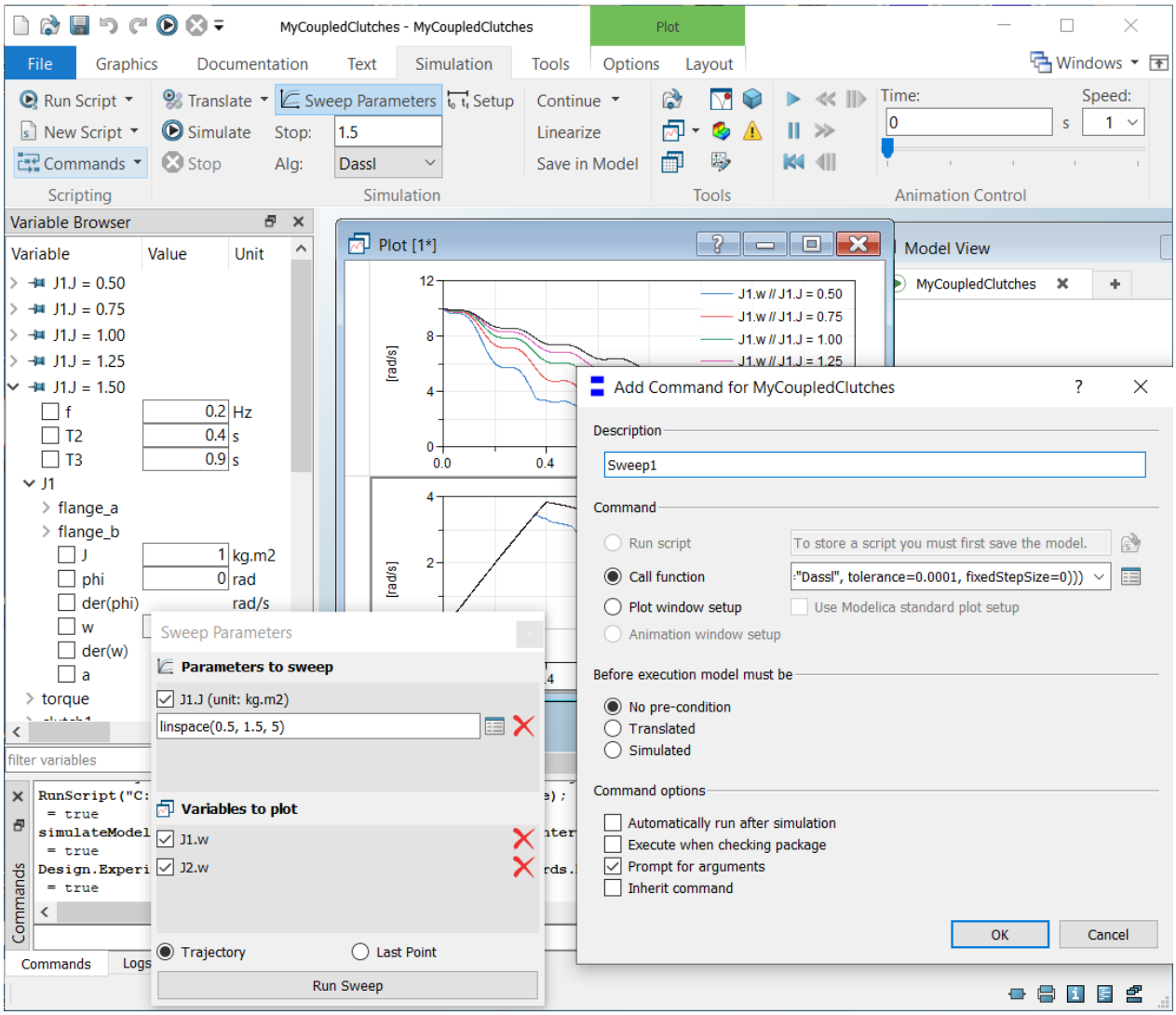
To have a suitable example, please perform the example of sweeping one parameter according in section "Sweeping one parameter" starting on page 1045. Note however, that in order to be able to associate commands to the model, you have to, as first steps:

- Open the Coupled Clutches demo by the command **File > Demos > Coupled Clutches**.
- In the Package browser, right-click CoupledClutches and select **New > Duplicate Class…** and change the name of the model to MyCoupledClutches.

The above will create an editable copy of the CoupledClutches model. Now you can proceed with the sweeping of one parameter according to the example.

To save the resulting sweep in the model MyCoupledClutches, do the following:

- Select **Commands > Add Command**. A dialog for adding the command appears:

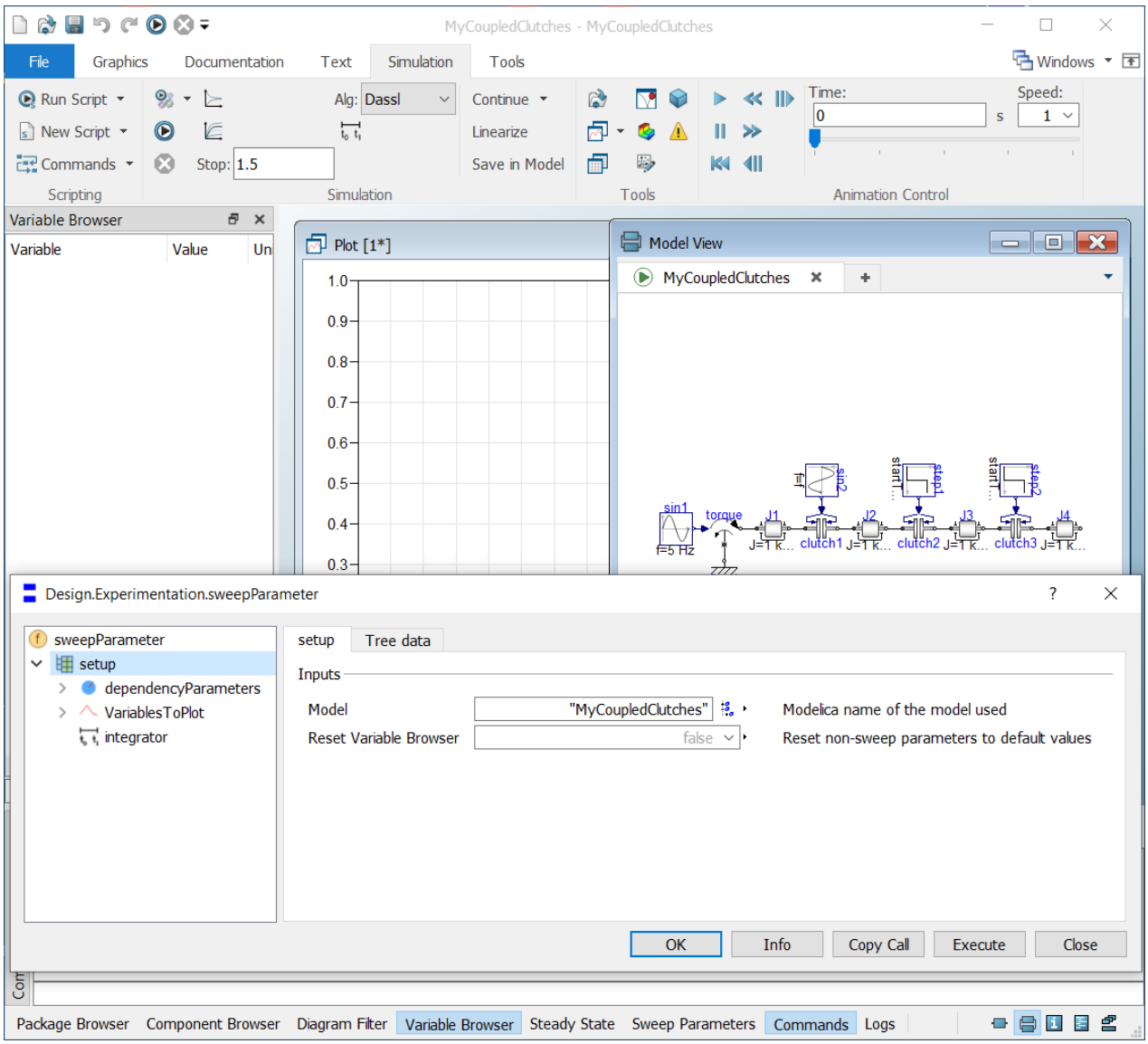


- You need to change two things in the dialog:
    - Activate **Prompt for arguments** (as done above).
    - Enter a name for the sweep setup, in this example the name **Sweep1** has been entered in the dialog above. Then click **OK** to store the command in the model.
- Save the model MyCoupledClutches, in a suitable location, by the command **File > Save > Save**.
- Close Dymola. (The reason for this is just to better illustrate the retrieval below.)

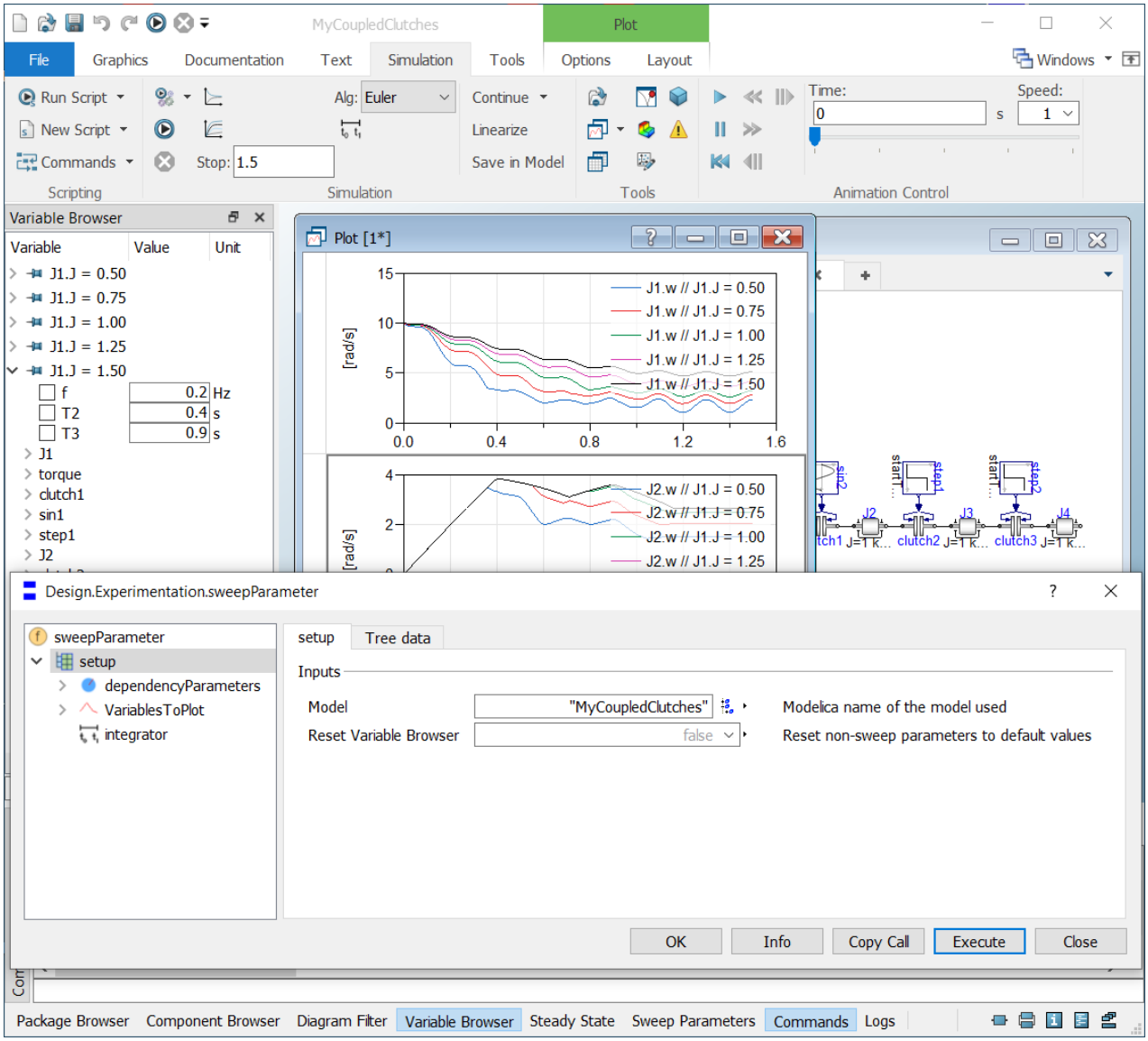
Retrieving the sweep setup

To retrieve the sweep setup stored in the example in the previous section, do the following:

- Start Dymola again and open MyCoupledClutches (a convenient way is to use the command **File > Recent Files**).
- Retrieve the sweep setup by giving the command **Simulation > Commands > Sweep1**. The result is:



- The setup is retrieved, but in the form of a function call of the function underlying the new sweep GUI. If you click **Execute**, you will sweep again with the stored sweep setup:



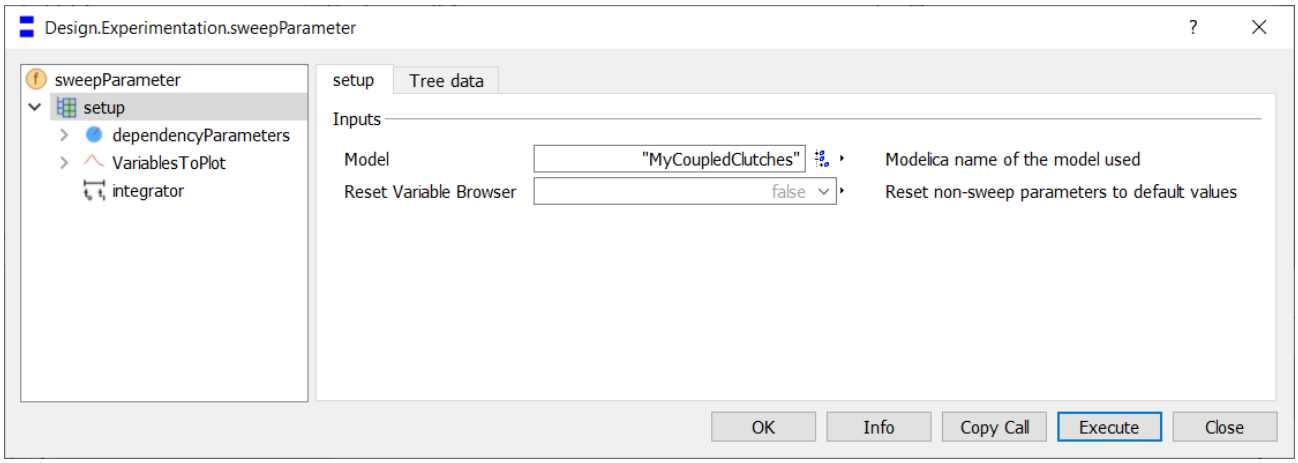
Notes:

- If the Sweep Parameters window is present, or if you open it, the dialog is not populated, since the new sweep GUI is not used in this case.
- If you click the Dymola main window, the dialog of the underlying function call is still available, although now hidden under the Dymola main window.

Changing a stored sweep setup

If you want to change the stored sweep setup **Sweep1**, you need to display the dialog of the underlying function. If you don´t have it from retrieving the sweep setup as above, do the following:

- Select **Simulation > Commands > Organize Commands**. In the dialog that appears, select **Sweep1** and then again click the **Edit** button. The dialog you used to define the command appears again; see the figure in the section “Store the sweep setup in the model” above.
- Click the **Edit** button after the function call. The following appears:

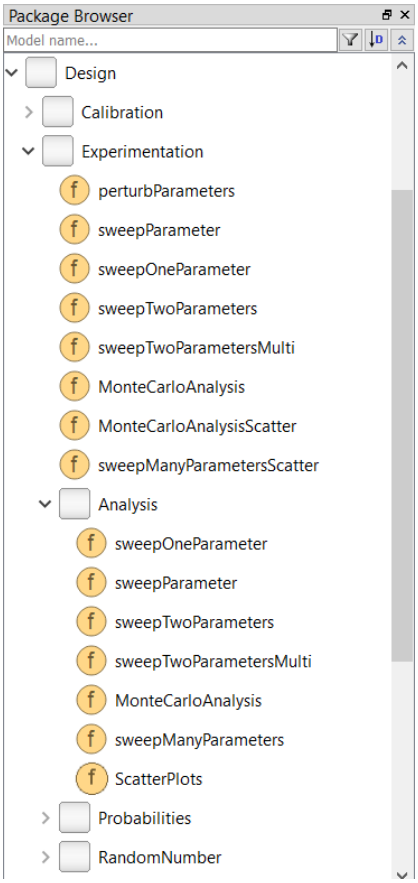


This is the same dialog that the one you got when retrieving the saved sweep setup. You can use this dialog to make changes, and then click **OK** to store them. (You need to click a couple of more **OK** is the dialogs that preceded this one to acknowledge your changes, and also to store the model again.)

Note the option **Reset Variable Browser**. Setting that option to true means that non-swept parameters that have been modified in the variable browser are reset to default values at sweeping.

6.3 Varying parameters of a model

The Experimentation package provides several ways of analyzing the behavior of a model. The main functions are the top-level functions perturbParameters, sweepParameter, sweepOneParameter, sweepTwoParameters, sweepManyParameters, MonteCarloAnalysis, MonteCarloAnalysisScatter, and sweepManyParametersScatter.



The top-level functions perturbParameters, sweepParameter and sweepTwoParameters have corresponding functions in the Calibration package and can be used for more general parameter studies. The main difference in this package compared to Calibration is that the resulting output is the response of the model. We give a short overview of these functions now.

The top-level functions sweepOneParameter, sweepTwoParametersMulti, sweepManyParametersScatter, MonteCarloAnalysis and MonteCarloAnalysisScatter complete the set, giving the possibility of plotting the response at the end of the integration interval, sweeping two parameters and plotting more than one criteria variable, sweeping more than two parameters, and finally plotting and analyzing random draws of numbers for the parameters in Monte Carlo simulations with or without scatter plots.

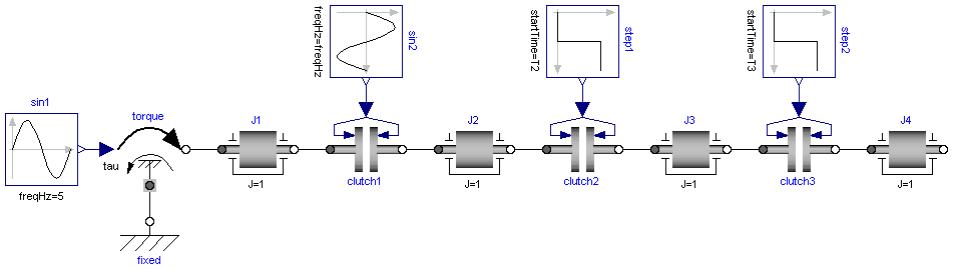
The functions in the Analysis subpackage are lower level functions obtaining the numeric result from the top-level functions. Note the function ScatterPlots that can be used to create scatter plots from external data.

The example studied for this package is the model Design.Experimentation.Examples.CoupledClutches. This example is an extension of Modelica.Mechanics.Rotational.Examples.CoupledClutches.

6.3.1 Case Study: CoupledClutches model

General

The model CoupledClutches is composed by four rotating inertias J1, J2, J3 and J4 coupled by three clutches that make them interact. The diagram looks as follows.

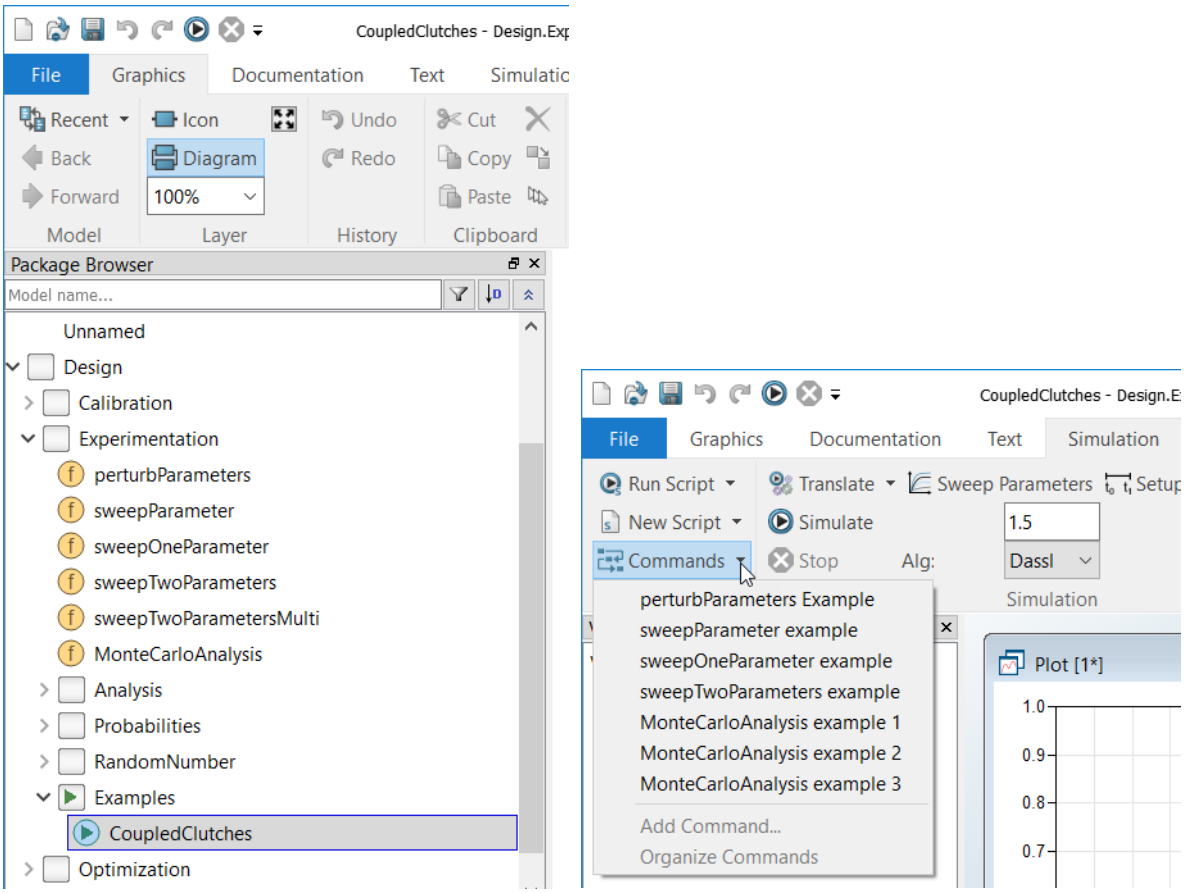


The parameters of the model to explore are the inertia values J1.J, J2.J, J3.J and J4.J. The observed variables are the rotational speeds J1.w, J2.w, J3.w and J4.w. The setups of the functions are very similar and their description will therefore be brief.

The demo CoupledClutches can be reached either in Modelica Standard Library, as Modelica.Mechanics.Rotational.Examples.CoupledClutches or in the Design library, as Design.Experimentation.Examples.CoupledClutches.

The Coupled Clutches demo with predefined commands

Opening the Coupled Clutches demo in the Design library, as Design.Experimentation.Examples.CoupledClutches will also give access to a number of commands that corresponds of some of the cases below. (The figure to the left shows the demo in the package browser available in, for example, the **Graphics** tab; the figure to the right shows the predefined commands available in the **Simulation** tab.)



Selecting any of these commands will display the relevant function with variables etc already selected. The only thing to do then is to click the button **Execute** to see the result. Please note that all cases are not handled by the commands, and not some minor adapting of e.g. curve legends after executing the command. More curves than needed might also be shown.

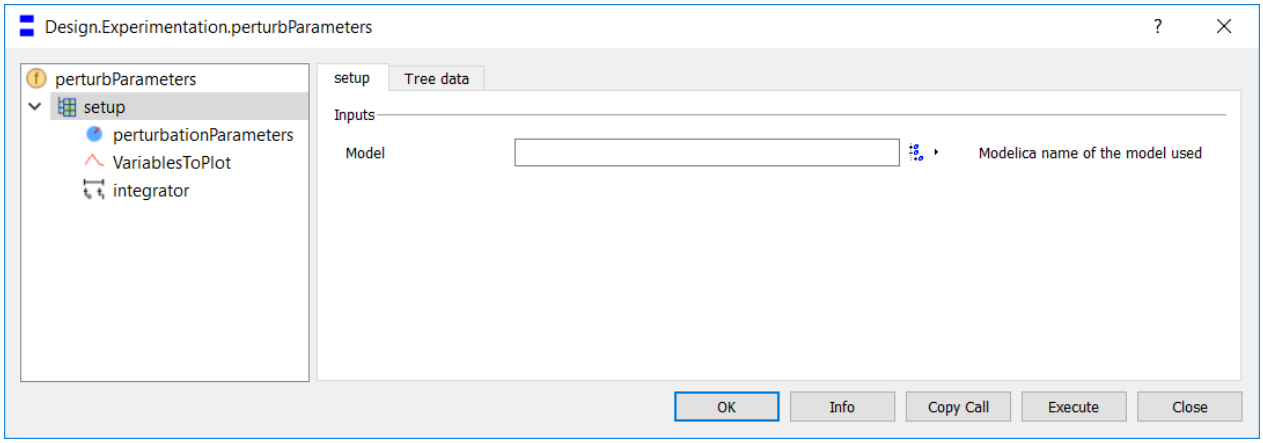
It is a good idea to open the CoupledClutches example from the Design package before continuing.

6.3.2 Response to parameter perturbations - perturbParameters

Let us check the behavior of the model if we perturb the nominal values of the parameters.

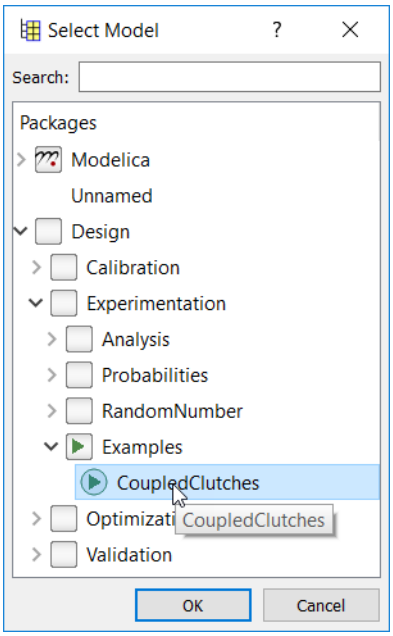
(Shortcut: Use the command **perturbParmeters Example** as described in “The Coupled Clutches demo with predefined commands” on page 1066.)

The function perturbParameters must be visible in the package browser. If not, expand Design and then Experimentation by clicking on the > in front of them. Now you can right-click on perturbParameters and select **Call Function ...**. The following menu pops

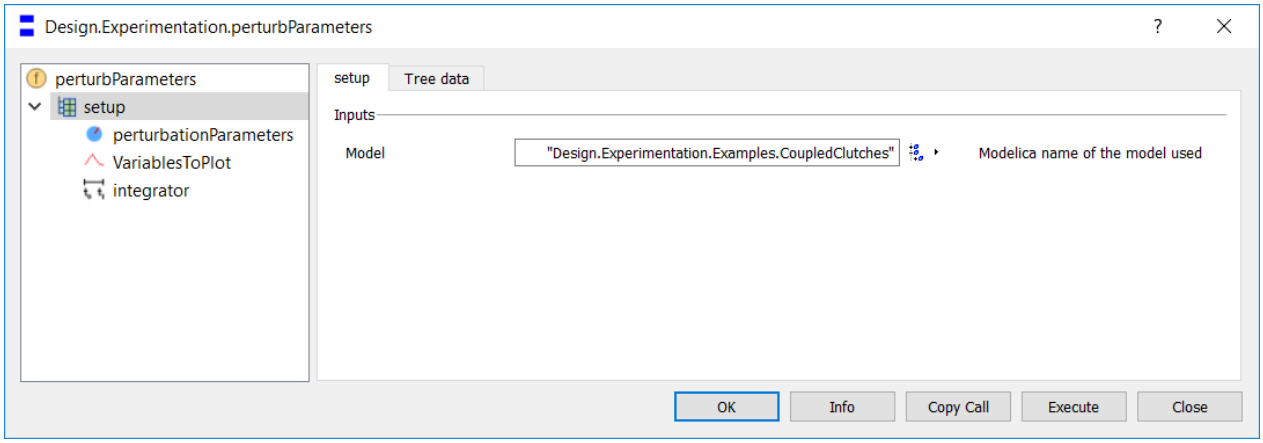




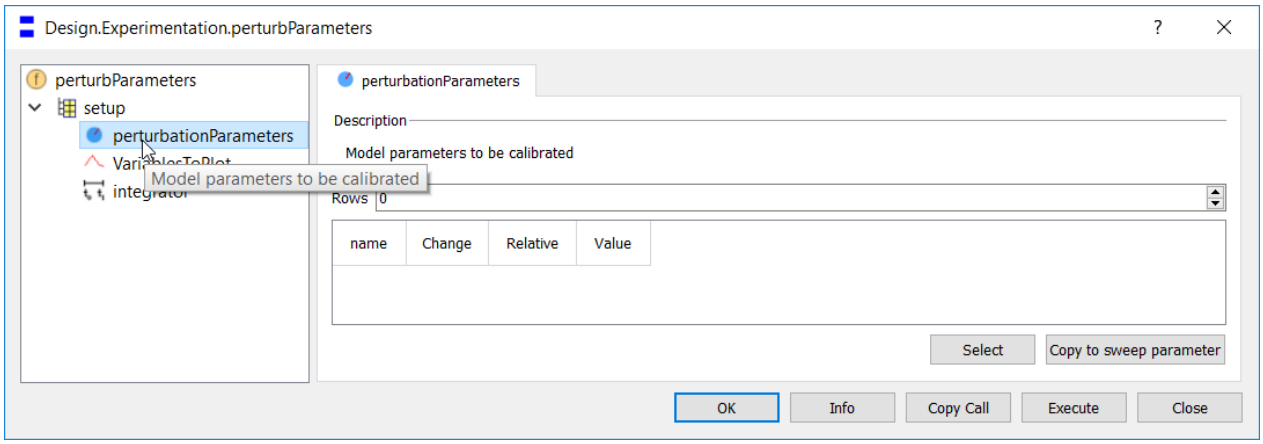
Now, to specify the model to use, click on **Edit** icon to the right of the input field. A package browser pops up. Use it to select the model.



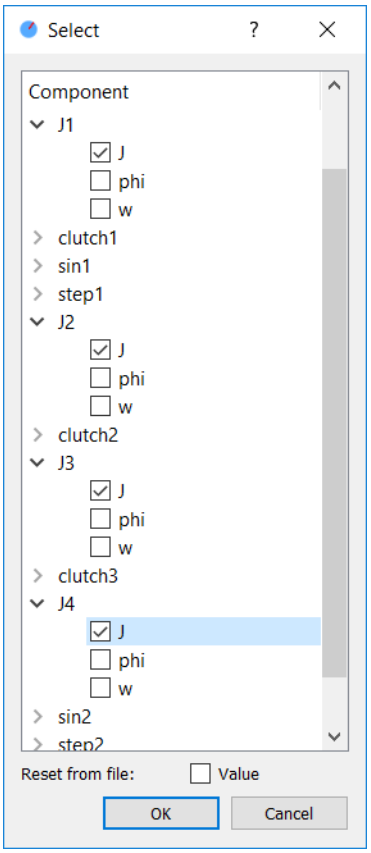
Click **OK**. The model is now translated in order to gather information needed to build browsers and selectors to support the remaining setting up. If Dymola already has a translated model, then this model appears as the default model.



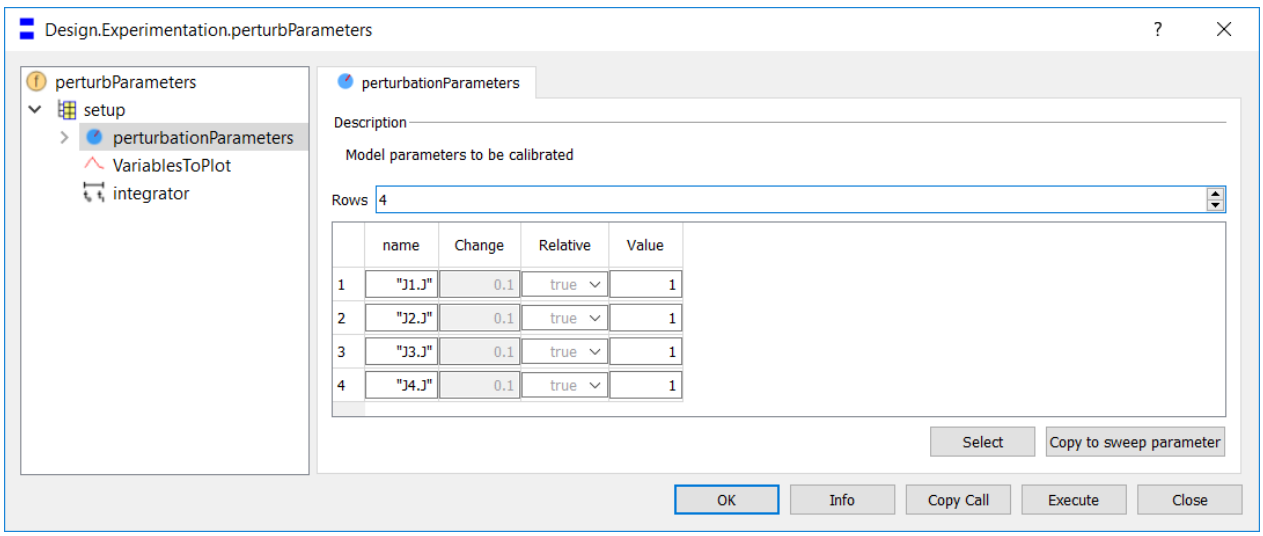
The next task is to select the parameters to perturb and the variables to observe and plot. Click on **perturbationParameters**. The following menu pops up.



Click on the **Select** button. The following browser pops and the parameters J1.J, J2.J, J3.J and J4.J can be selected as perturbation parameters. Their nominal value is 1 for all of them. The perturbation is by default 10 percent.

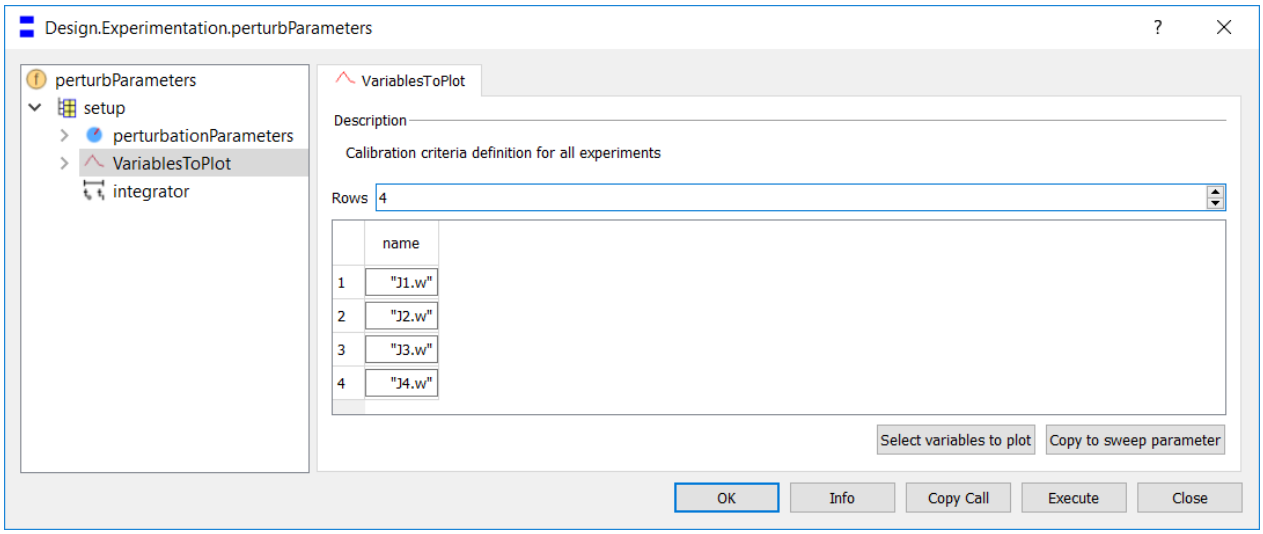

Select
Component
J1
J
phi
w
clutch1
sin1
step1
J2
J
phi
w
clutch2
J3
J
phi
w
clutch3
J4
J
phi
w
sin2
step2
Reset from file:
Value
OK
Cancel

Click **OK**.

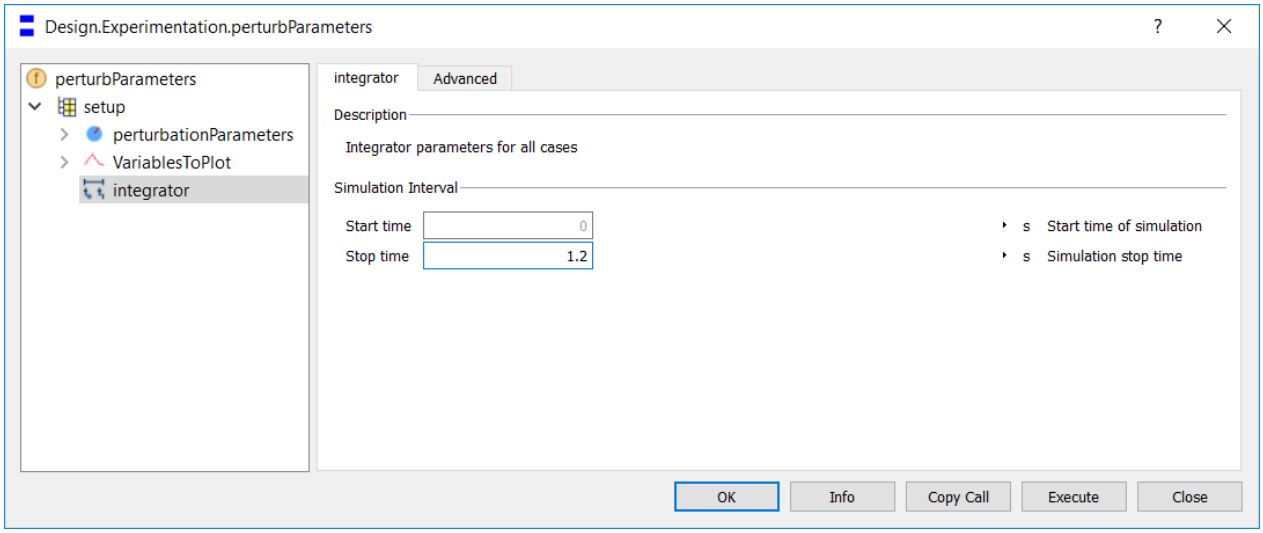

Design.Experimentation.perturbParameters
perturbParameters
setup
perturbationParameters
VariablesToPlot
integrator
perturbationParameters
Description
Model parameters to be calibrated
Rows 4
name
Change
Relative
Value
1 "J1.J" 0.1 true 1
2 "J2.J" 0.1 true 1
3 "J3.J" 0.1 true 1
4 "J4.J" 0.1 true 1
Select
Copy to sweep parameter
OK
Info
Copy Call
Execute
Close

We can select a percent change of absolute change if we like. In the setup presented, the parameters are perturbed 10 percent from their nominal value.

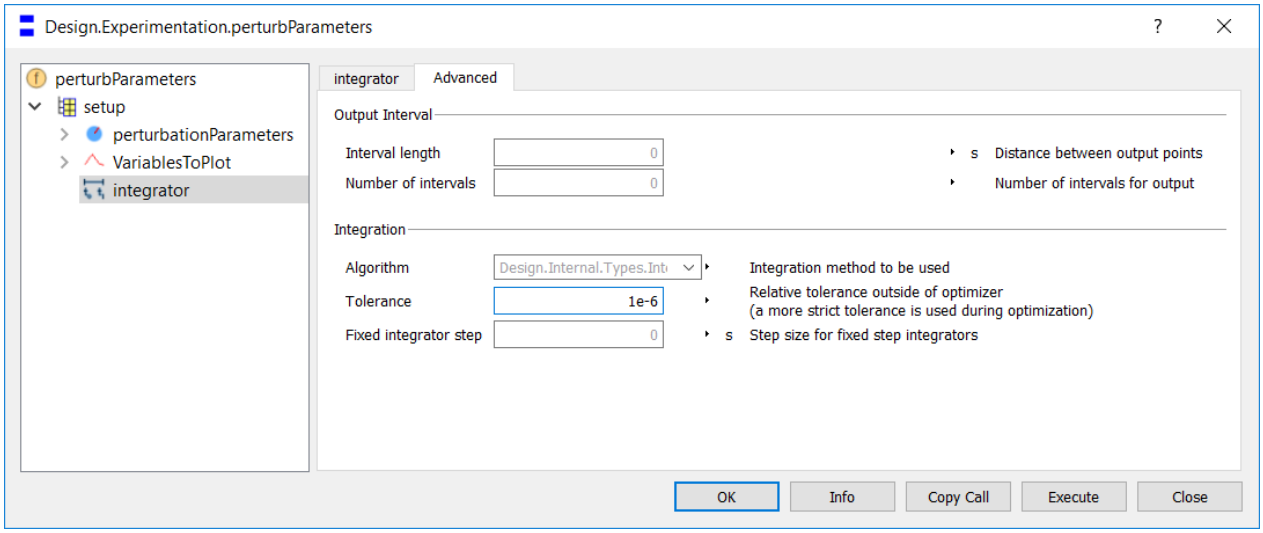
Now, let us select the variables to plot. Click on **VariablesToPlot** and then clicking on **Select variables to plot** button we get a variable browser where the selection of J1.w, J2.w, J3.w and J4.w is possible. The resulting menu looks as following.



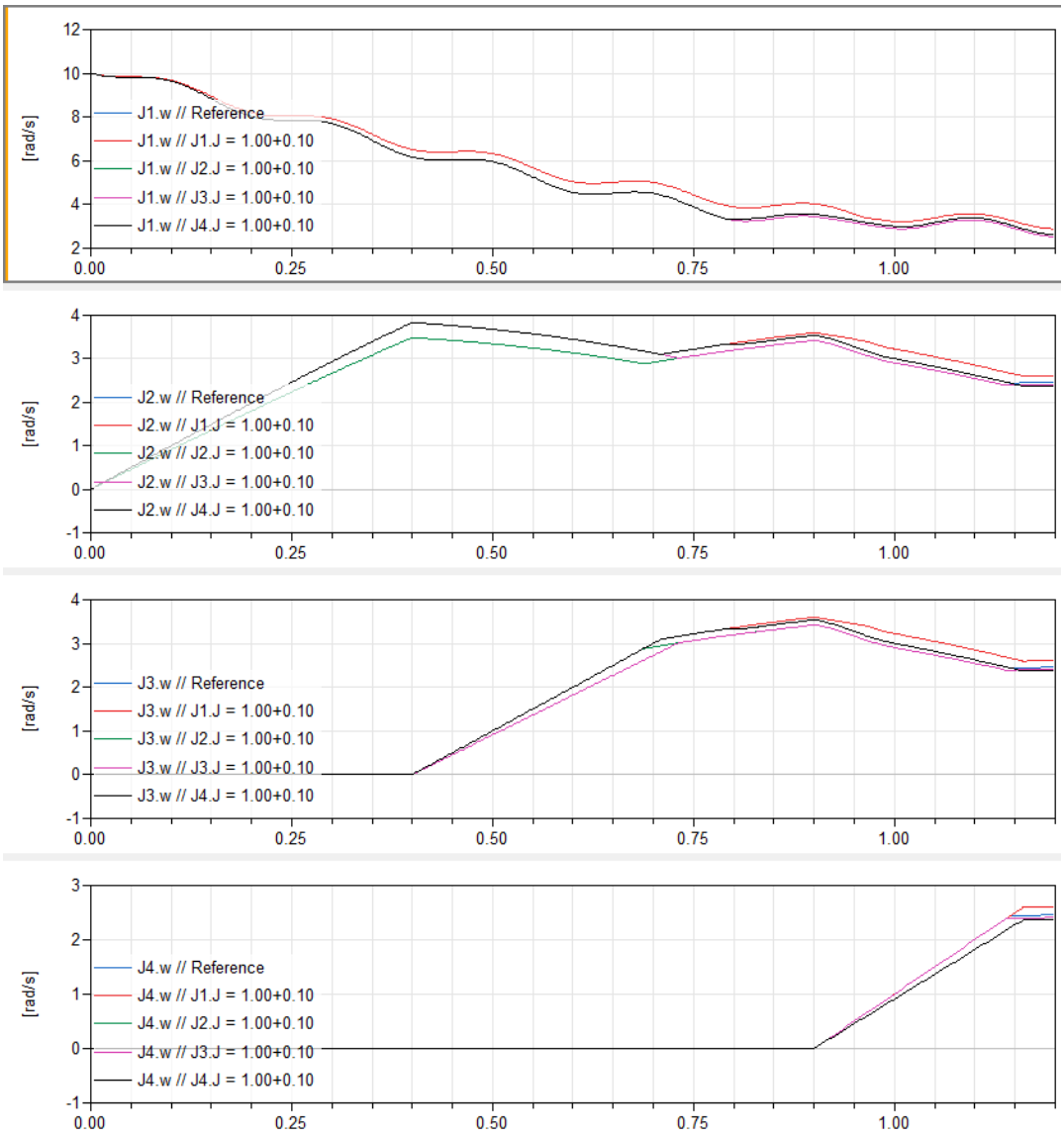
Finally, the setup for the integrator is to be done. Click on **integrator** in the left pane and set the stop time to 1.2.



Click on the **Advanced** tab and select the default tolerance for the integrator lowered to 1e-6.



Now we can run the command. Click on **Execute**. After the simulations, and moving the curves and legends to the appropriate place (some curves are on top of each other), we get the following sequence of images.



The plots show the variation of every variable when varying the parameters J1.J, J2.J, J3.J and J4.J 10 percent, one at a time. We observe, for instance, in the first plot that only the variation of J1.J affects the response on J1.w.

6.3.3 Sweep one parameter – two variants

The phenomenon described before can be observed in another fashion. We can sweep one parameter and observe the result along the whole interval form 0 to 1.2, or just at the final

time of 1.2 seconds. These variants are implemented in two functions; sweepParameter and sweepOneParameter.

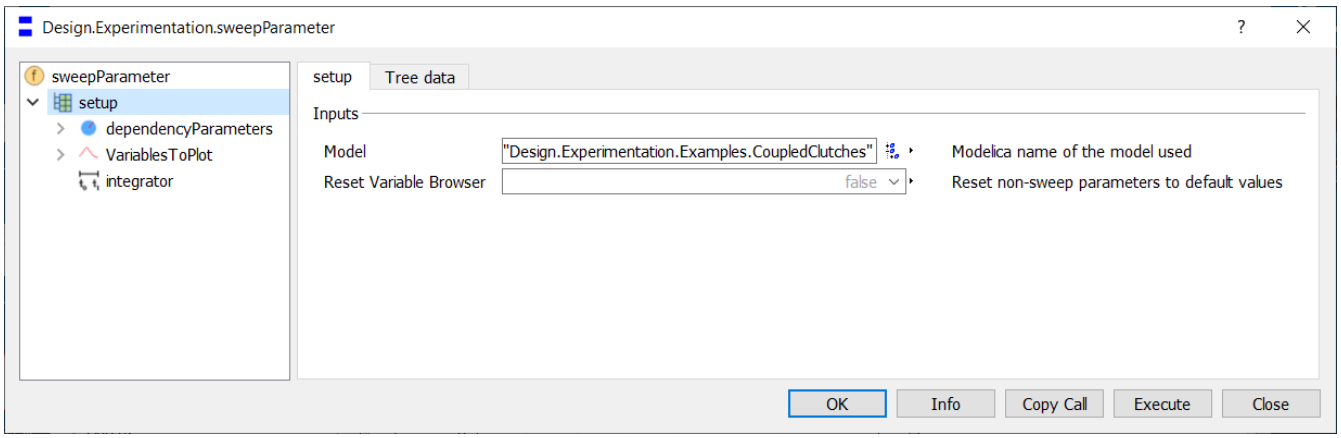
Note! You can also use a new GUI to sweep parameters, see section “Sweeping parameters using new GUI” starting at page 1044.

sweepParameter

The setup of this function is very similar to perturbParameters.

(Shortcut: Use the command **sweepParameter example** as described in “The Coupled Clutches demo with predefined commands” in page 1066.)

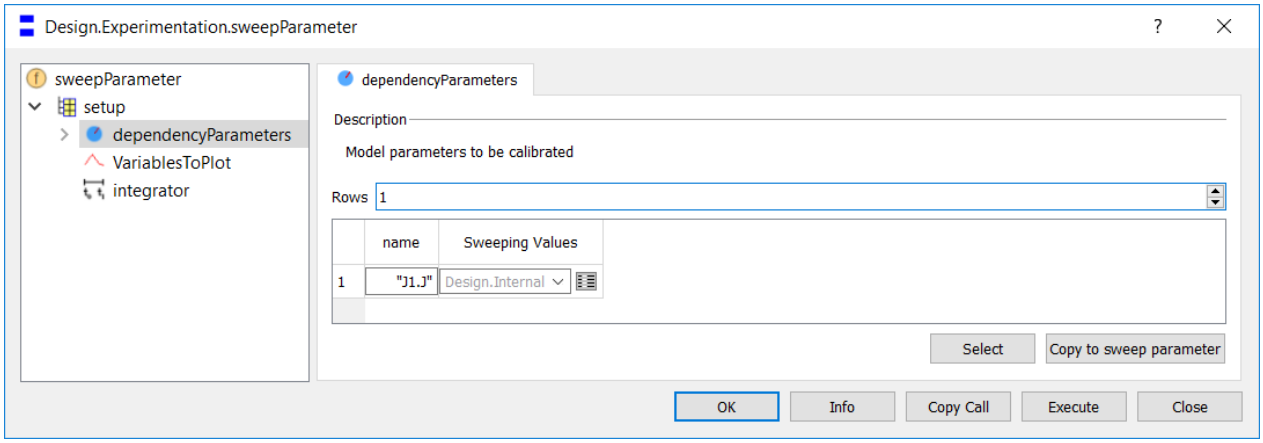
If the previous example has been executed, display the package browser by, for example, activating the **Graphics** tab, and, in the Experimentation package, right the function **sweepParameter** and select **Call Function...**. The model is already filled in (if not it has to be selected as in previous example).



We have to select the dependency parameter and the variable to plot. The way is the same as in the previous example.

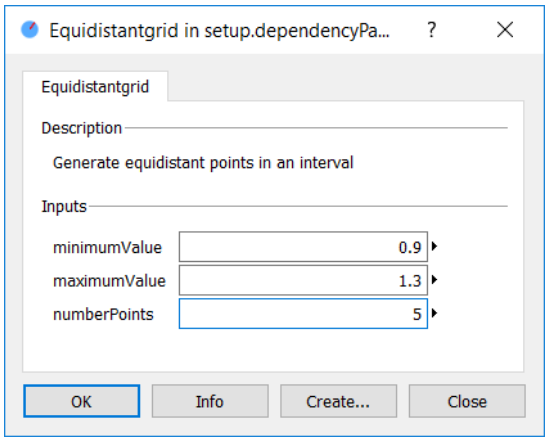
(**Note** the option **Reset Variable Browser**. If you set that option to true, non-swept parameters that have been modified in the variable browser are set to default values when performing the sweep; this is a specific option for some sweep functions.)

The first thing to do is to specify dependencyParameters (click on **dependencyParameters** in the left of the menu). The **Select** button can be used to select J1.J. The result will be:

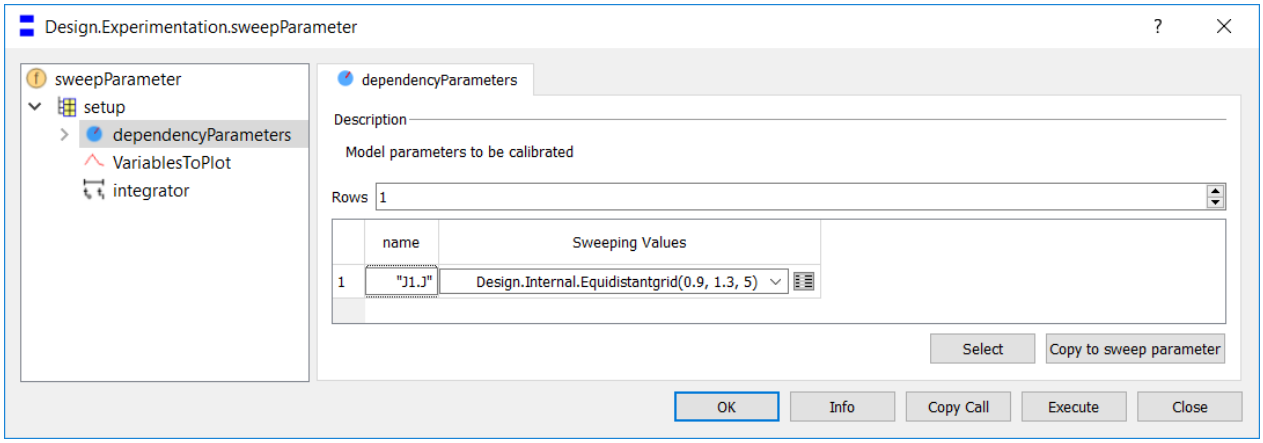




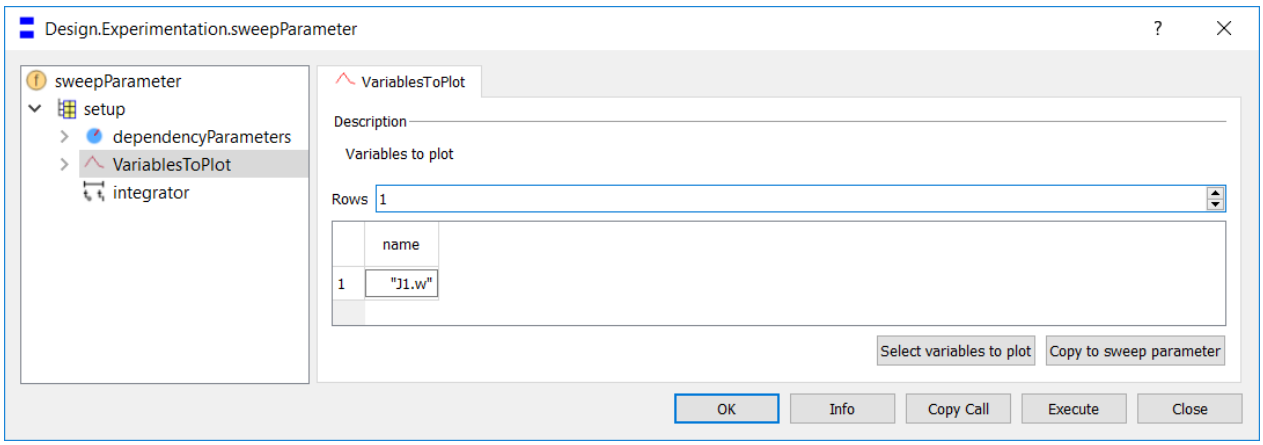
The **Edit** icon to the right of the Sweeping Values column can be used to select five equidistant values between 0.9 and 1.3 for J1.J.



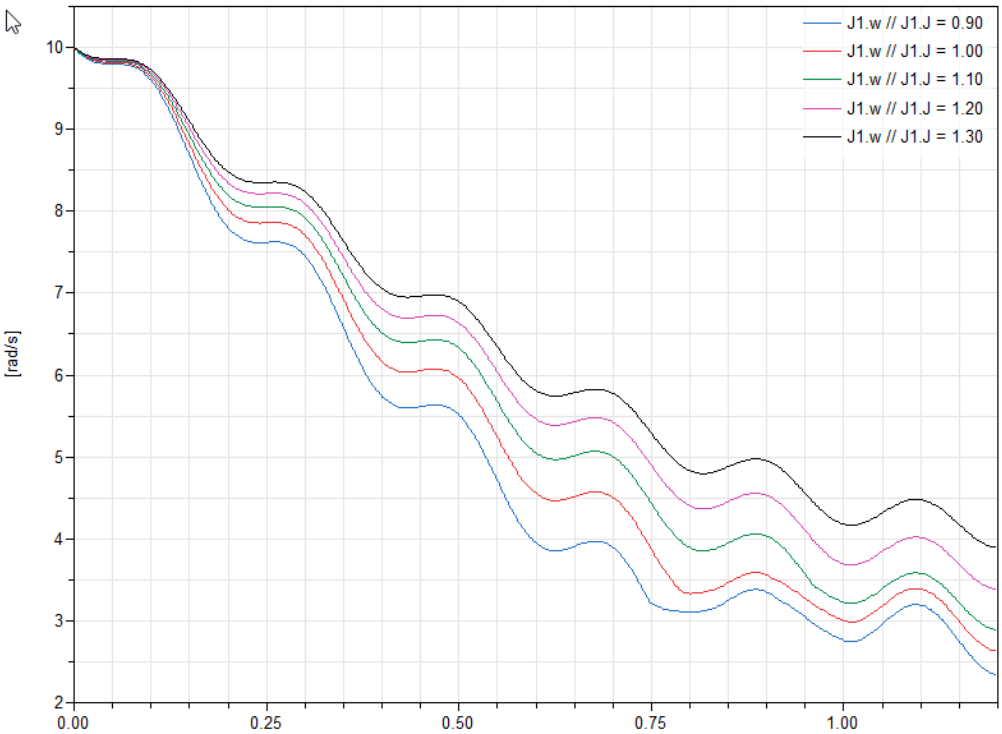
Click **OK**. The result will be



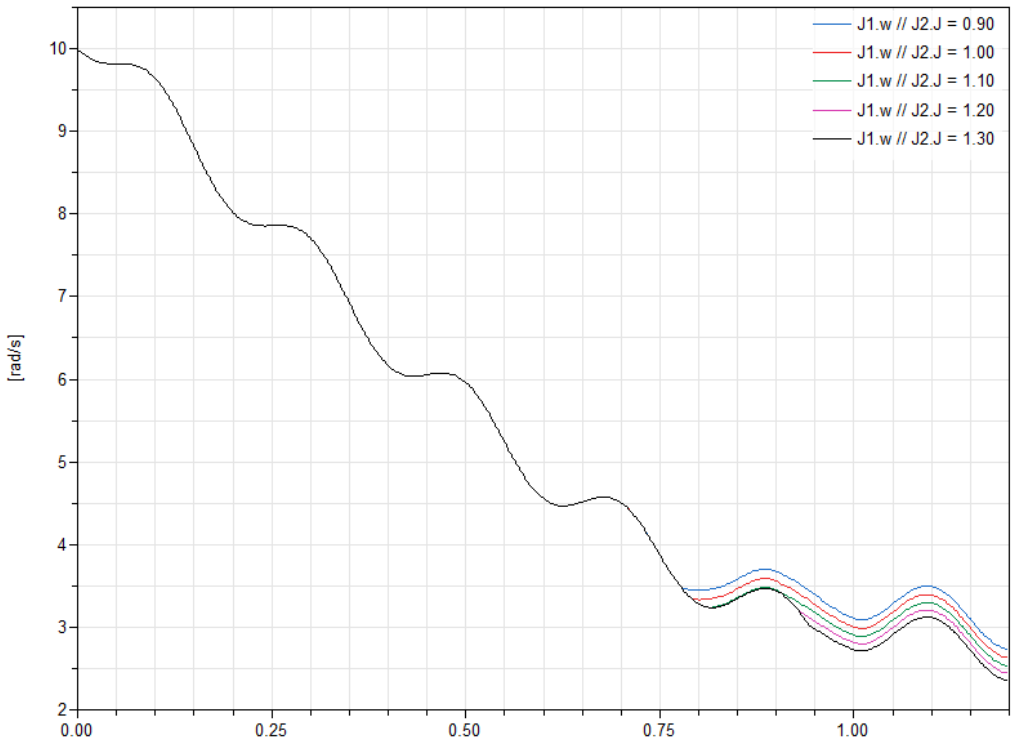
Use **VariablesToPlot** to select the variable to plot (like in previous example) in this case the variable should be J1.w



Don't forget to set the Stop Time to 1.2 in the **integrator** setup and the tolerance to 1e-6 (like in the previous example)! Press **Execute** and the result follows.



Let us observe now J1.w and vary J2.J. Exchange in the setup J1.J with J2.J, in **dependencyParameters** setup. Don't forget that the Sweeping Values has to be set again. (You can also just edit the name of the variable; then you don´t need to redo the sweeping values.) Press **Execute** again. (This example is not included in the demo commands in the beginning of this chapter.)



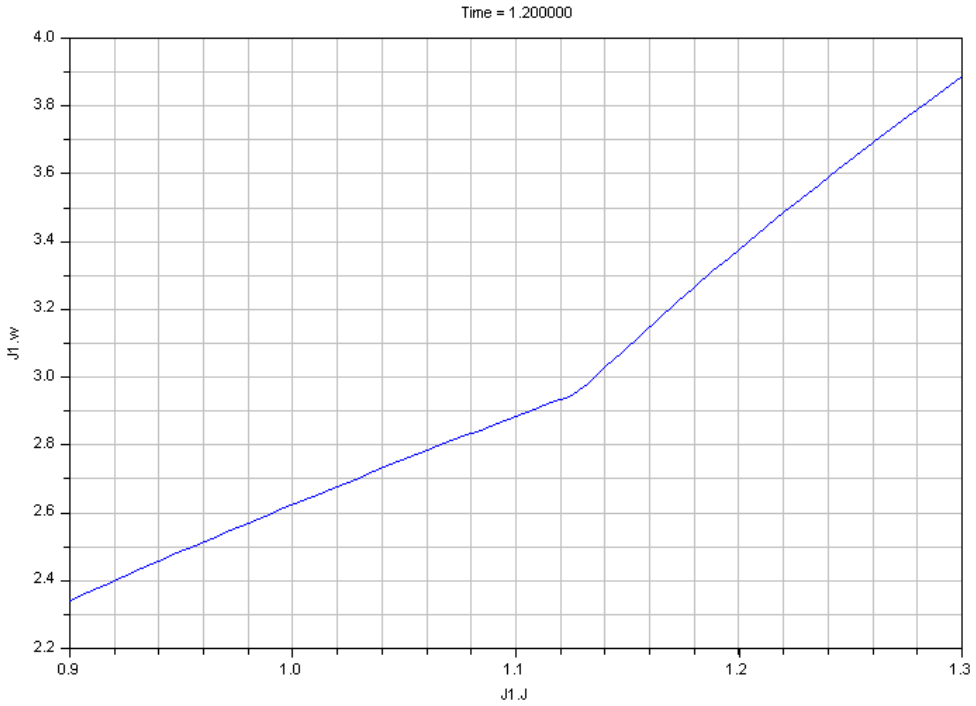
The response J1.w is less sensitive at the beginning of the interval to variations of J2.w. At the end, when all inertias are coupled, the variation is larger.

sweepOneParameter

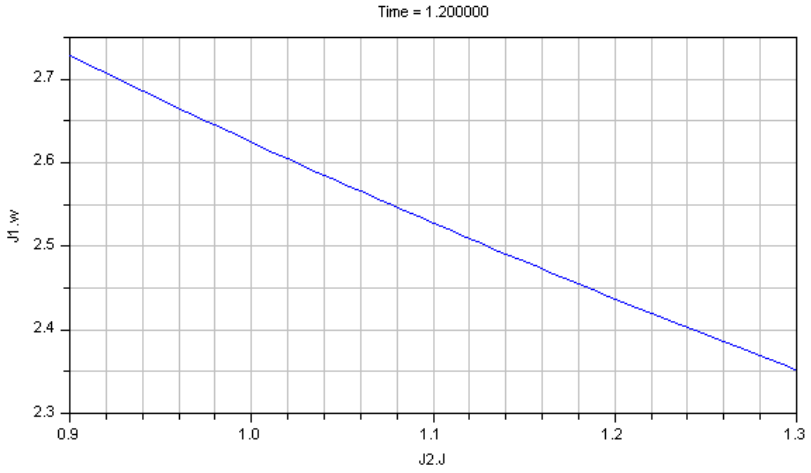
If our interest is just the response at end point of the interval, we use sweepOneParameter. This setup is the same as for sweepParameter.

(Shortcut: Use the command **sweepOneParameter example** as described in “The Coupled Clutches demo with predefined commands” on page 1066.)

Just choose J1.J as dependency variable in the same way, take 51 values between 0.9 and 1.3 and use J1.w as variable to plot. The following curve is obtained when the command is executed. Once more, don’t forget to set the Stop Time to 1.2 in the integrator setup and the tolerance to 1e-6.



This curve relates at t=1.2 the parameter J1.J and the response J1.w. The same situation can be depicted for J2.J as parameter and J1.w as response. (This case is not covered by any command in the beginning of this chapter.)

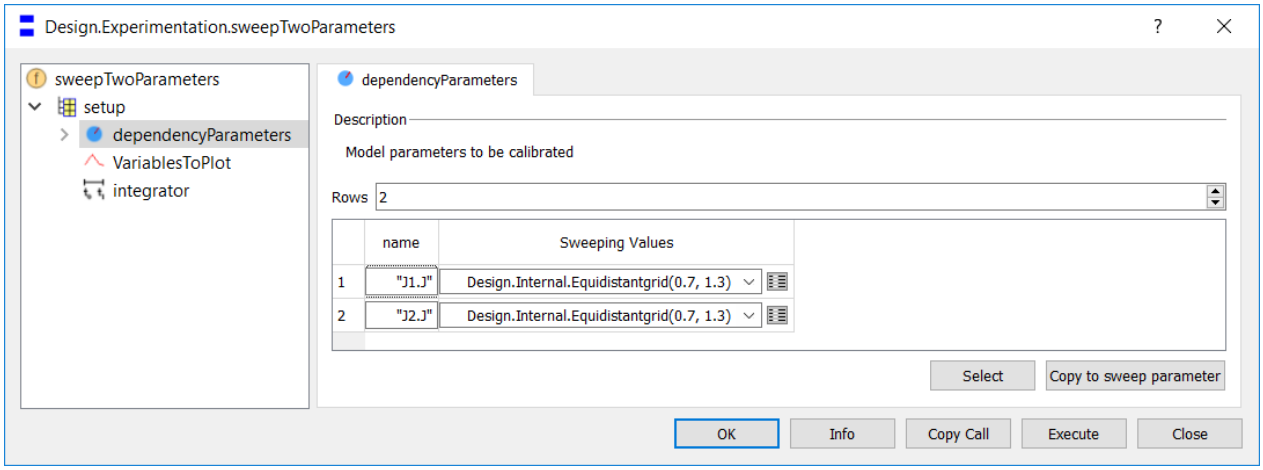


6.3.4 Sweep two parameters - sweepTwoParameters

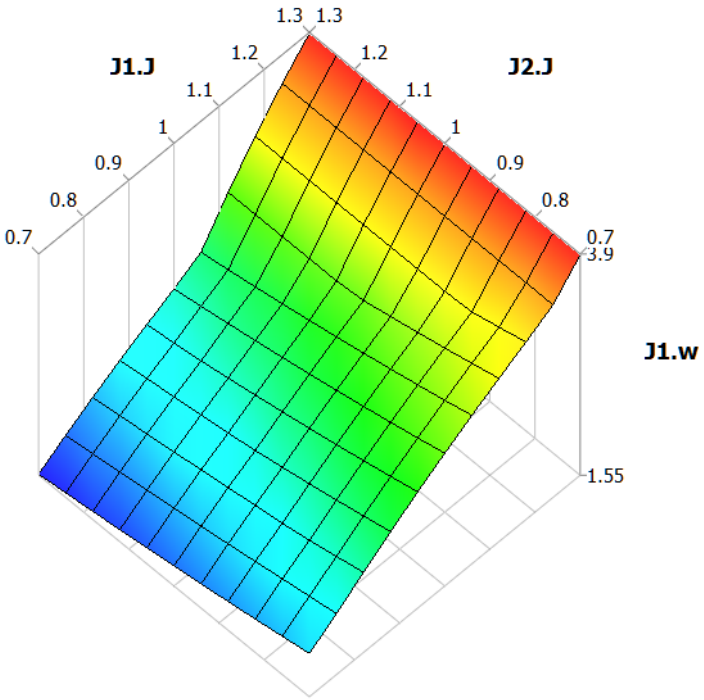
To study the dependence of one response with respect to two parameters at the end of the integration interval, the function sweepTwoParameters is to be used. The setup is almost identical to sweepParameter and sweepOneParameter. The only difference is that two dependency variables are to be selected instead.

(Shortcut: Use the command **sweepTwoParmeter example** as described in “The Coupled Clutches demo with predefined commands” on page 1066.)

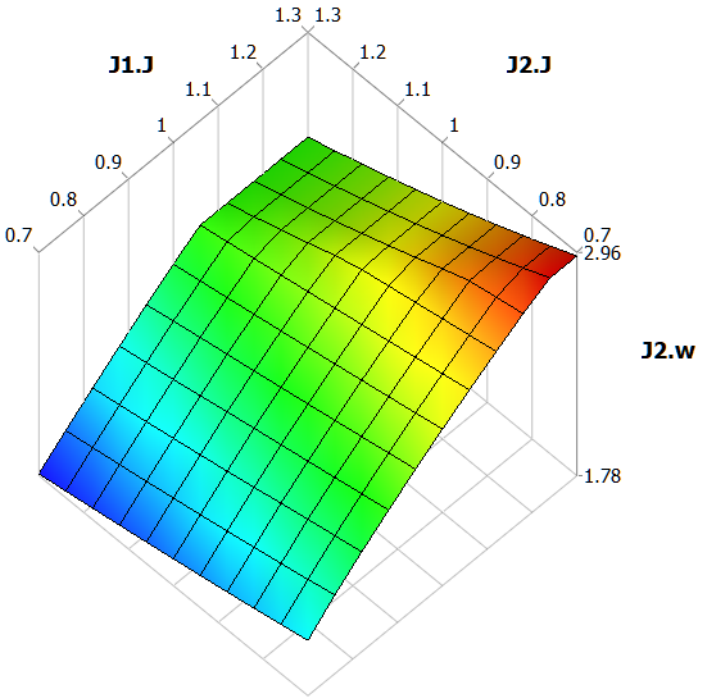
Note! You can also use a new GUI to sweep parameters, see section “Sweeping parameters using new GUI” starting at page 1044.



We observe now J1.w against J1.J and J2.J. The values chosen for J1.J and J2.J are eleven values between 0.7 and 1.3 for both variables. (Keep the number of points to the default 11.) Even for this case, the Stop time is 1.2 and the tolerance is 1e-6 in the integrator tab. The result of execution is:



Observing J2.w gives the following result. (Please note that this case is not covered by the demo command in the beginning in this chapter.)



Note that the function **sweepTwoParametersMulti** can be used if you want to plot more than one criteria variable.

6.3.5 Sweeping many parameters - sweepManyParameters

The function **sweepManyParametersScatter** makes it possible sweeping several (more than two) parameters. The results are quite similar to the **MonteCarloAnalysisScatter** function except for the randomization in Monte Carlo (see next section). For more about sweeping many parameters, see “Sweeping more than two parameters” starting on page 1055.

6.3.6 Monte Carlo Analysis

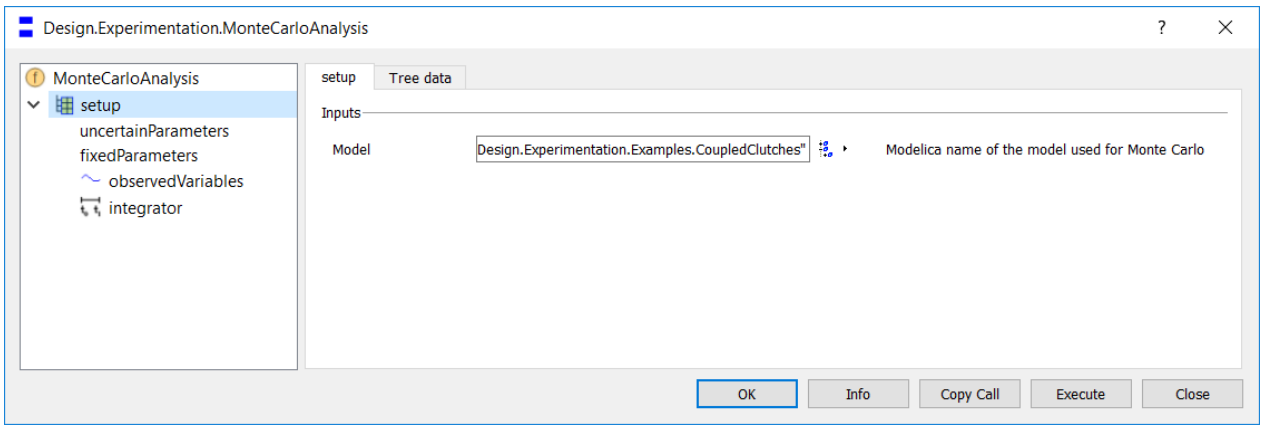
Monte Carlo Analysis without scatter plots

Monte Carlo Analysis is widely used to explore the behavior of a model when the input parameters are multidimensional. We will set up now the command MonteCarloAnalysis to observe the model response when varying J1.J, J2.J, J3.J and J4.J at the same time.

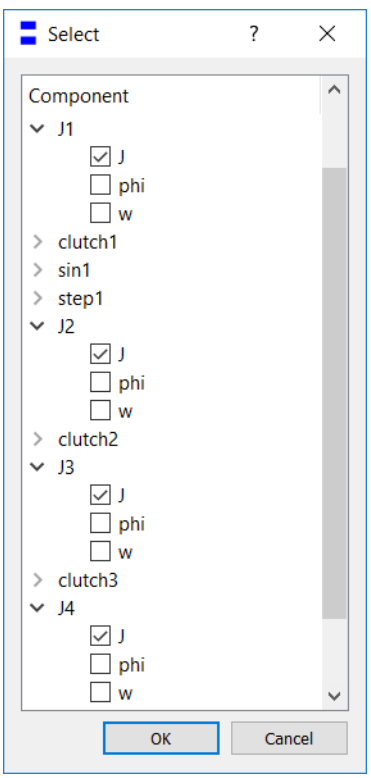
Note the possibility to present the result as a matrix of color-coded scatter plots, see the next section.

(Shortcut: Use the command **MonteCarloAnalysis example 1**, as described in “The Coupled Clutches demo with predefined commands” on page 1066.)

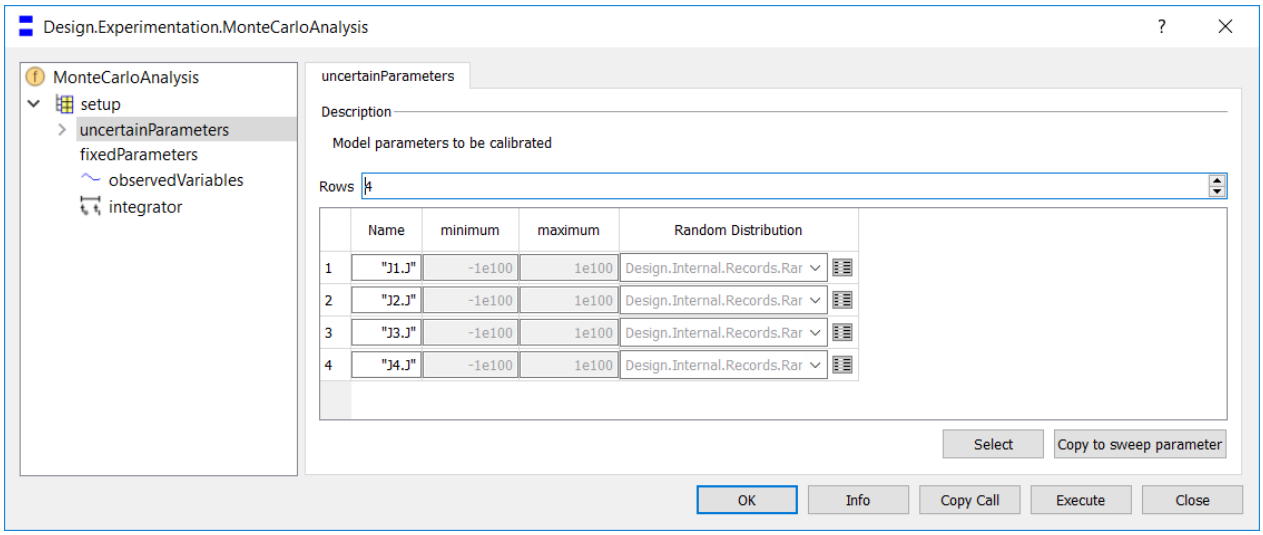
Right-click on the function **MonteCarloAnalysis** in the Experimentation package and select **Call Function...**. A menu pops up. Since we so far started by defining the setup, please click on **setup** in the browser to the left. Select the model (if not preselected) like in previous examples. The result will be:



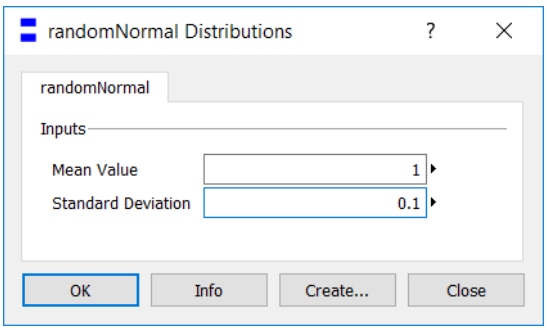
The task now, as before, is to select the uncertain parameters. Click on **uncertainParameters** and click on the **Select** button. Select the browser J1.J to J4.J.



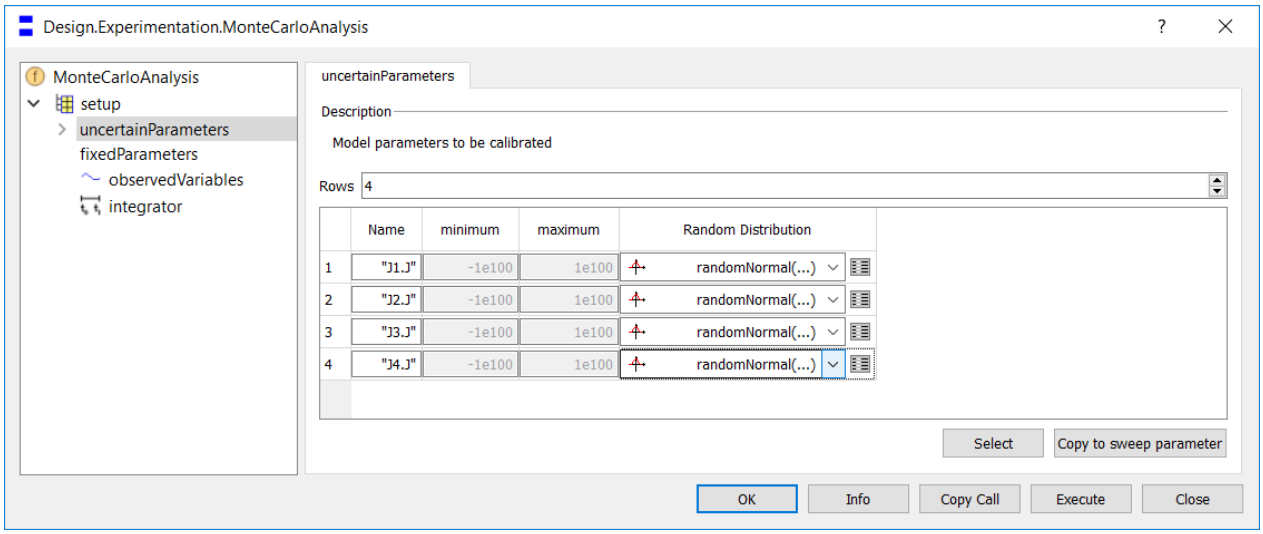
Click **OK**. The next step is to select a random distribution for every inertia.



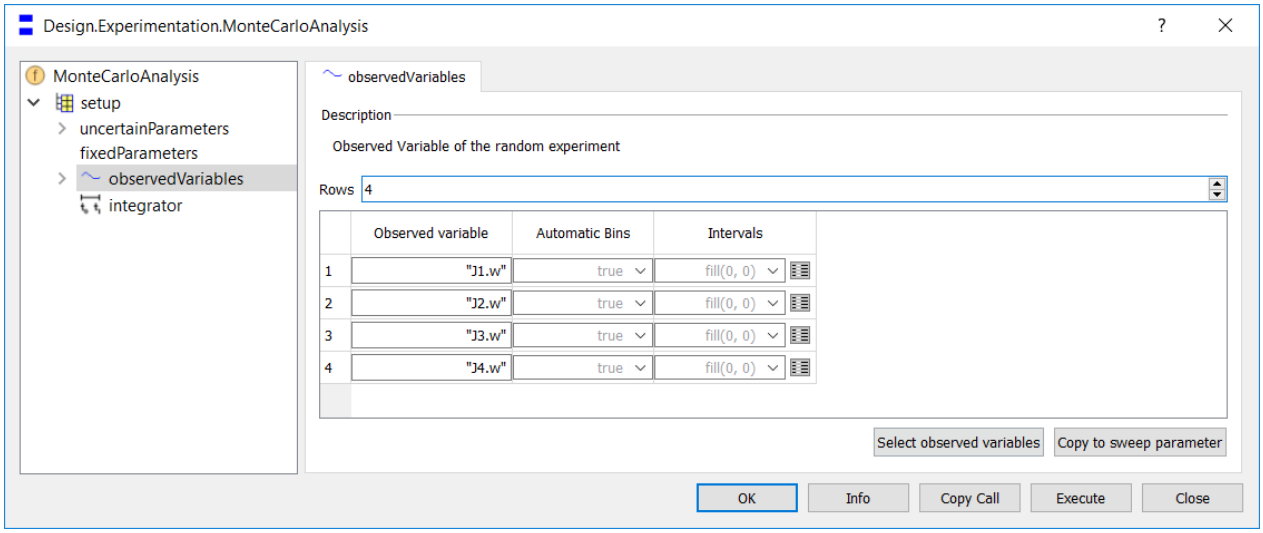
Click on the arrow of the combo box and select **randomNormal** for J1.J. Another menu pops up asking for values for **Mean Value** and **Standard Deviation**. Those values characterize the normal distribution to be used. Set mean to 1 and standard deviation to 0.1.



Click **OK**. Repeat the same process for J2.J to J4.J.



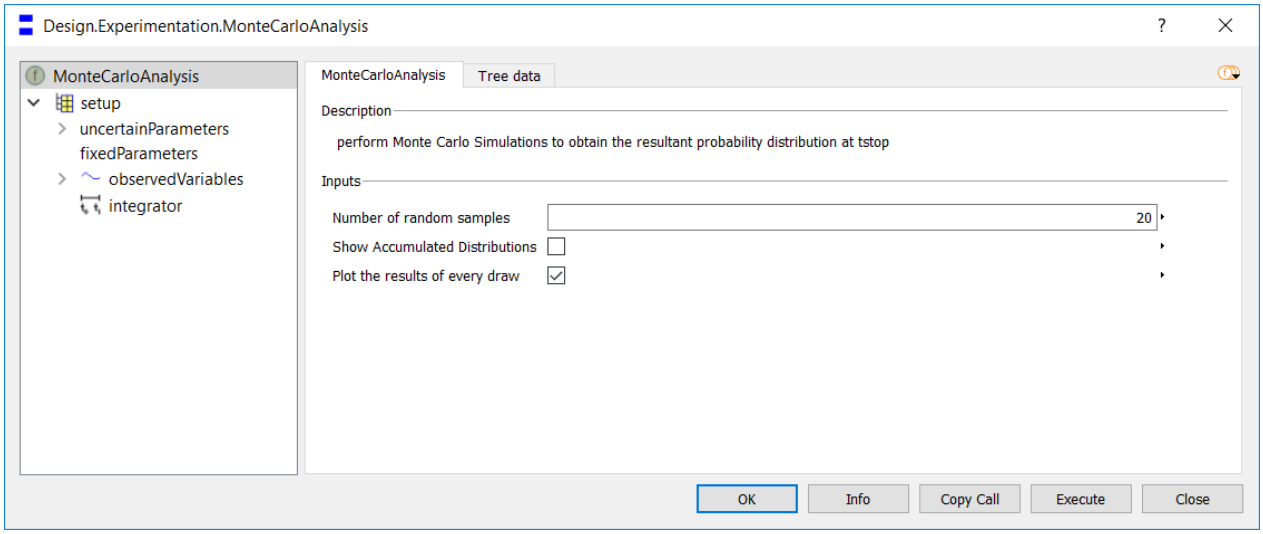
The setup for **fixedParameters** is used if we want to specify other simulation situations than the nominal values written in the model. For instance, if the initial angle J1.phi is specified and different from zero, we should add it there. In our case, we don't have such fixed parameters so we just go directly to observed variables. Click on **observedVariables** and press the button **Select observed variables**. Mark in the browser J1.w, J2.w, J3.w and J4.w.



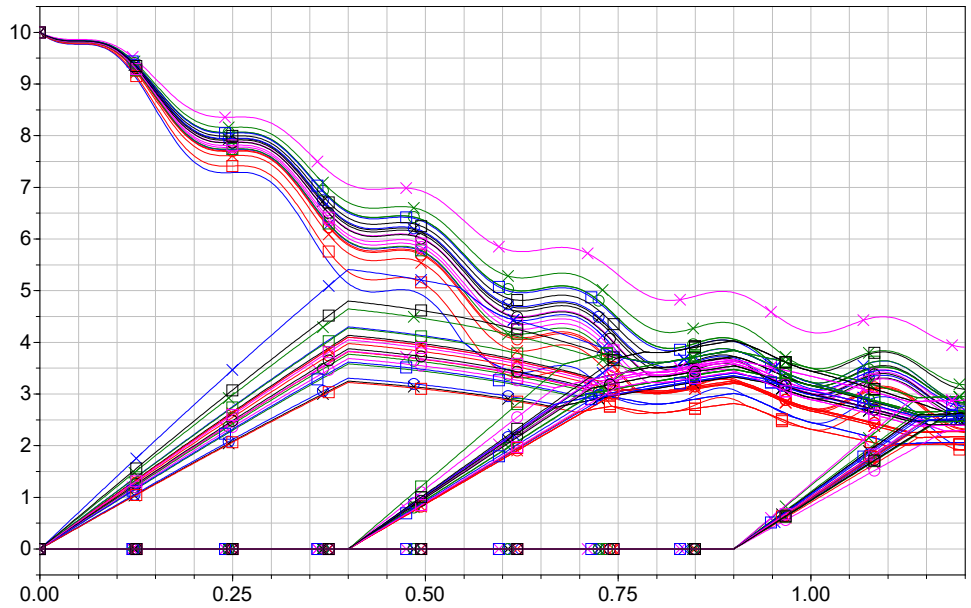
The flag **Automatic Bins** set to true allows the algorithm to choose automatically an appropriate set of bins, according to the maximum and minimum values observed in the result. It takes also into account the total number of samples to set the appropriate resolution.

Set the integrator stop time once more to 1.2 (click on the **integrator** in the browser in the left pane of the window to display this setting).

To set up the type of desired result, click on **MonteCarloAnalysis** in the browser in the left pane of the window. We set the number of draws in the field **Number of random samples**. As we want to plot the result of every draw, only twenty draws are needed. Check also **Plot the results of every draw** to obtain the plot of the responses and the density of probability.

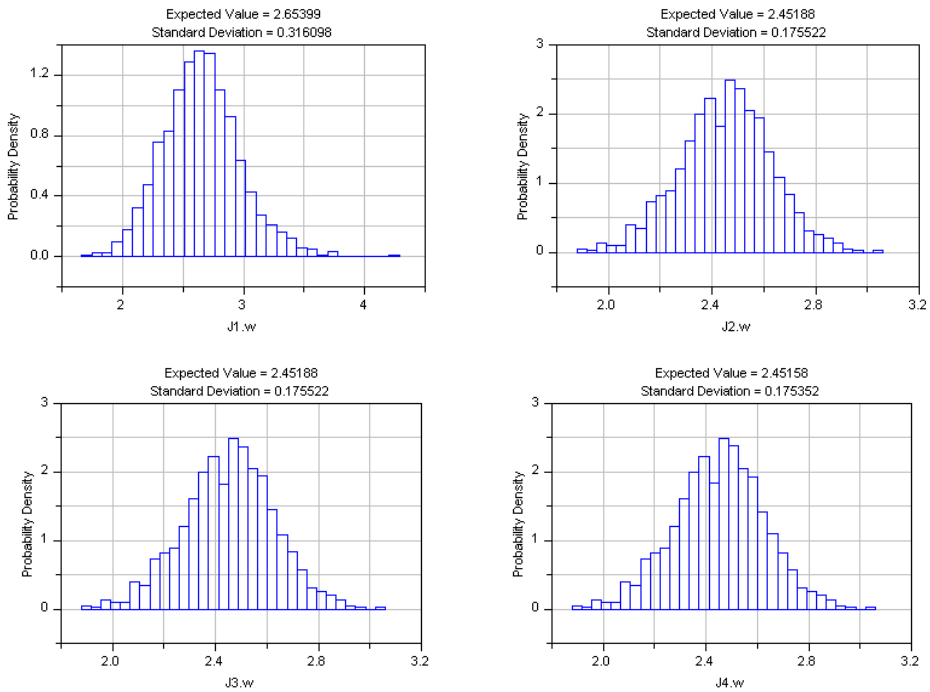


Click on **Execute**. The result will be a number of plot windows. The first (that is, you have to minimize the ones on top to see it) plot will look similar to the following:

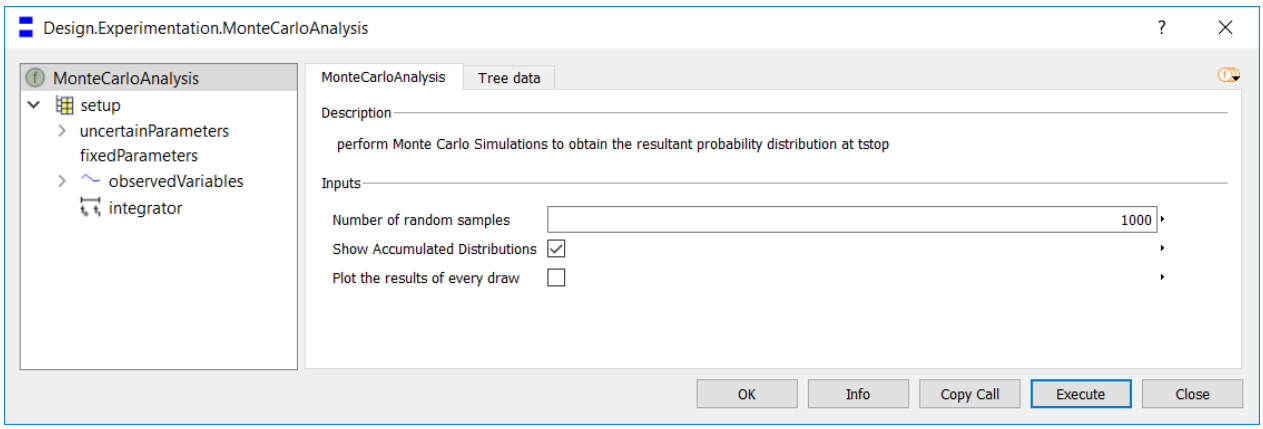


In this graph we observe the variation of slope and behavior produced by random sampling of the values of J1.J, J2.J J3.J and J4.J in time.

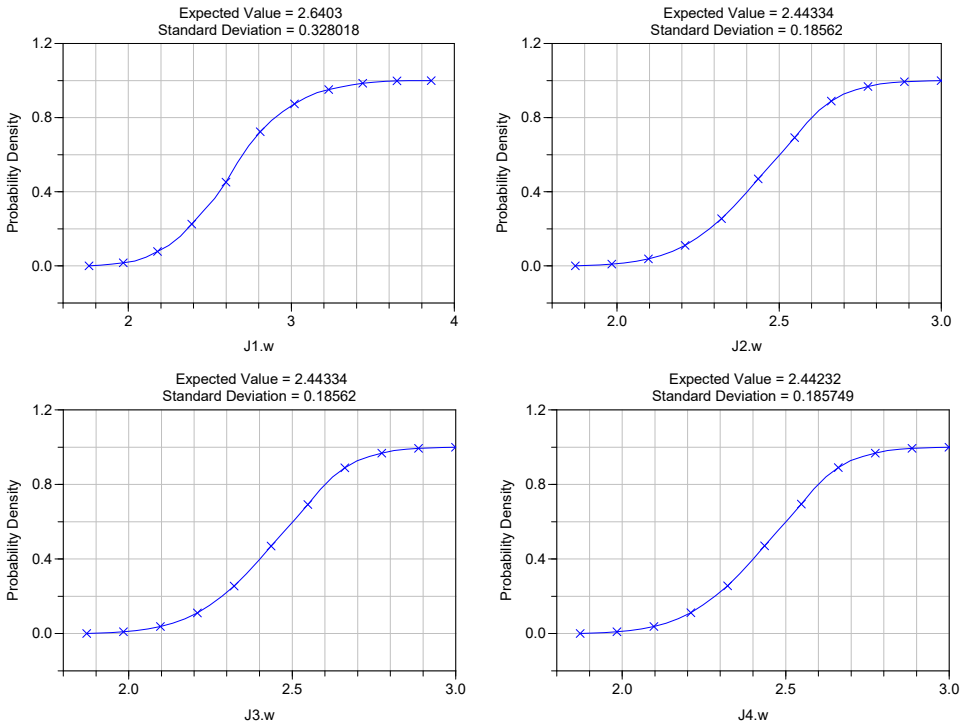
If the plots of the density of probability or accumulated probability are important, we change the setup to plot those with more samples. To plot the densities, we take 1000 samples and uncheck the flag **Plot results of every draw**. Press **Execute** to obtain the plots (this corresponds to using the command **MonteCarloAnalysis example 2** as described in the beginning of this chapter).



To plot the accumulated distributions, check the flag **Show Accumulated Distributions**.



Click on **Execute**. (This corresponds to using the command **MonteCarloAnalysis example 3** as described in the beginning of this chapter.) The result plots follow.

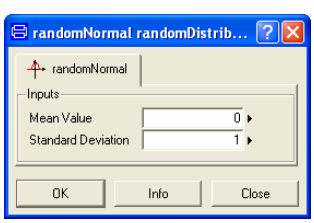
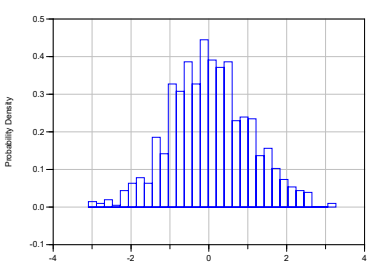
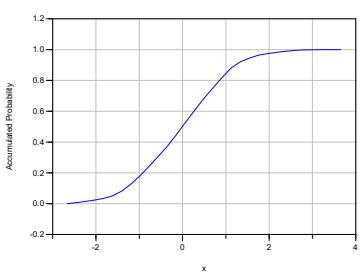


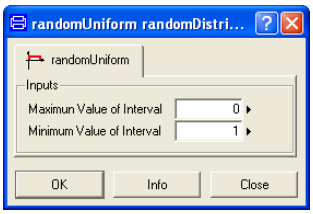
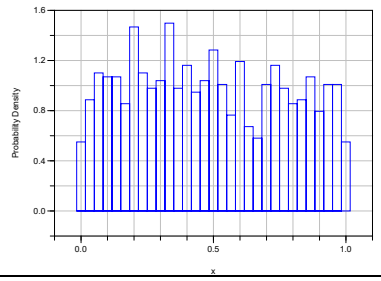
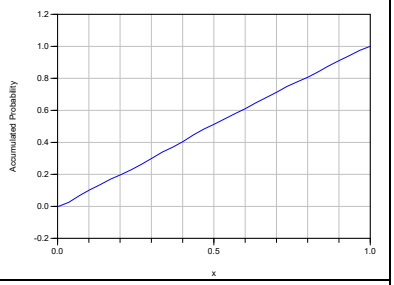
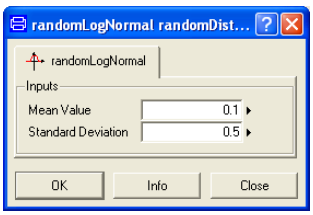
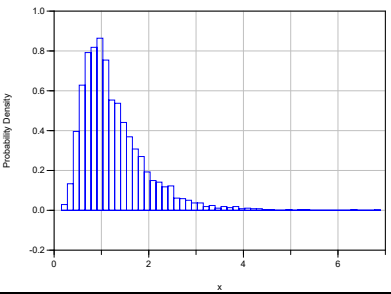
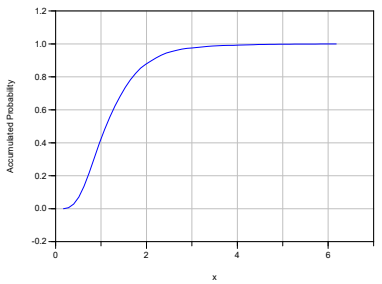
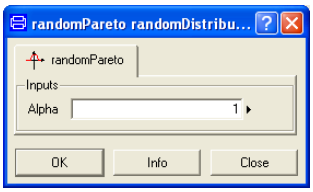
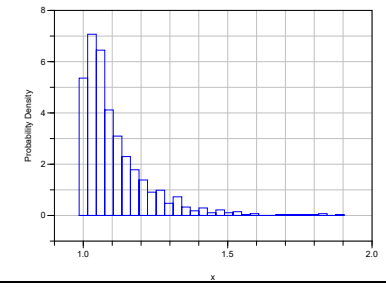
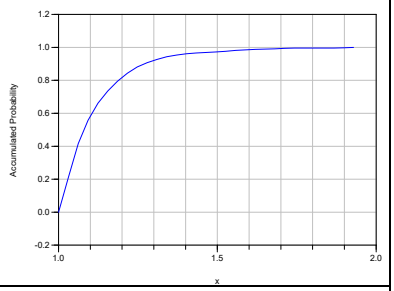
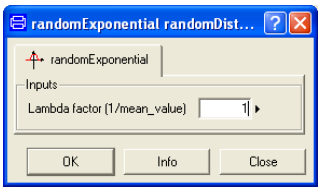
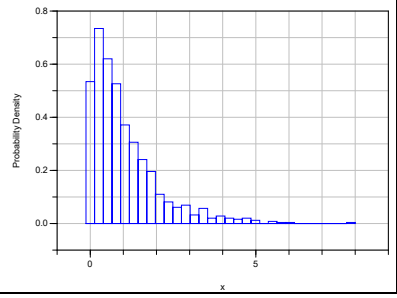
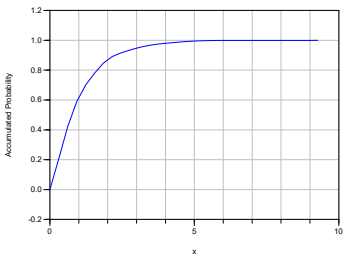
Monte Carlo Analysis with scatter plots

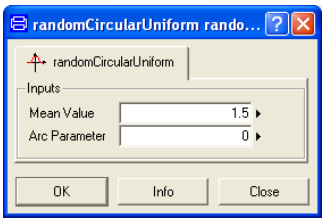
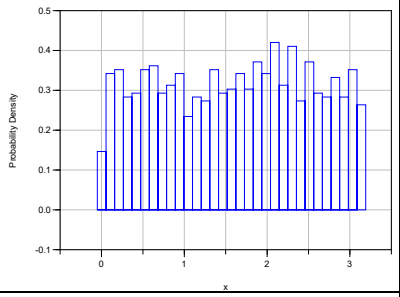
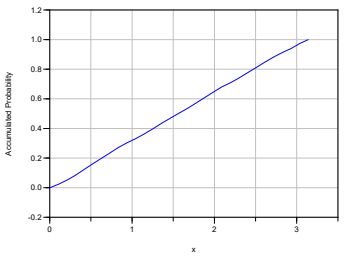
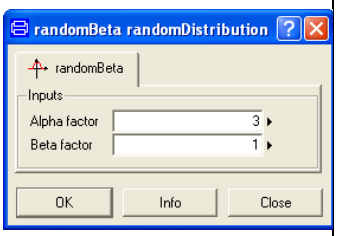
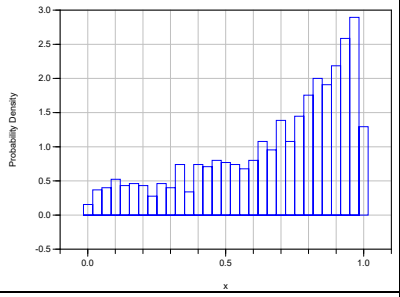
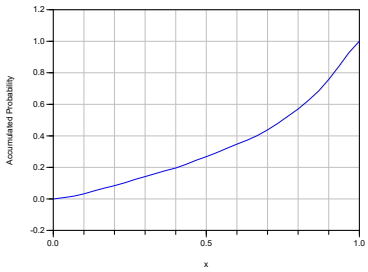
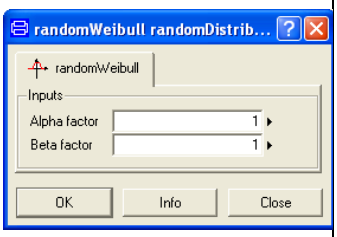
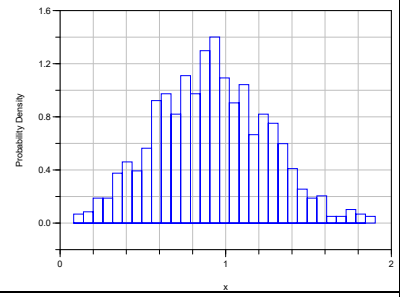
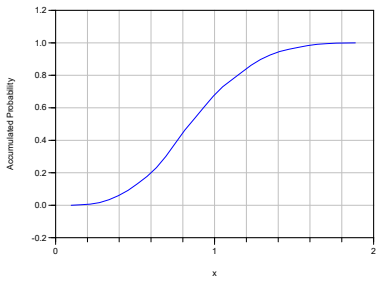
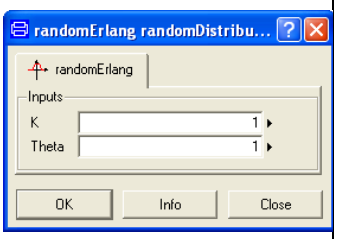
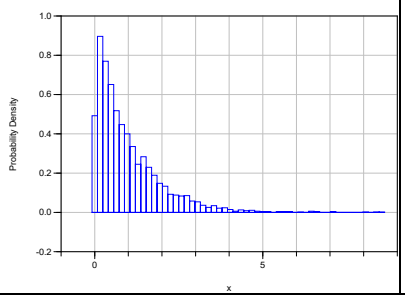
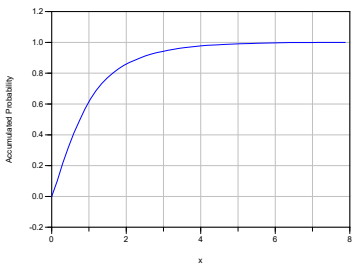
You can use the function **MonteCarloAnalysisScatter** to perform a Monte Carlo analysis and present the result as a matrix of color-coded scatter plots. Apart from using the scatter plots, the function is very similar the above function.

Random Distributions available and their parameters

The following table reviews briefly the random distributions in Experimentation package that can be used together with MonteCarloAnalysis.

Distribution	Parameters	Probability density	Accumulated probability
Normal			

Uniform			
Logarithmic Normal			
Pareto			
Exponential			

Circular Uniform			
Beta			
Weibull			
Erlang			

7 MODEL CALIBRATION

7 Model Calibration

7.1 Introduction

Dymola includes features to perform integrated computer experiments with Modelica models. This document describes the features to calibrate and to assess models. The functions described in this document are parts of the Design.Calibration package. There is no licensing for Model Calibration.

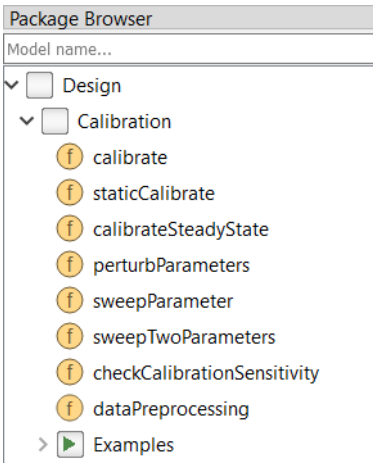
Consider a Modelica model describing a physical system. Such a model includes typically many parameters, which have to be set. Some parameter values can be found from design sheets. Some parameters such as physical dimensions may be easy to measure on the system. Direct measurements of the weights of the parts are more difficult since it requires the system to be dismounted. Moreover, it is for example not simple to measure the inertia of a part. Friction and loss parameters are good examples of parameters that often are unknown.

Model calibration (parameter estimation) is the process where measured data from a real device is used to tune parameters such that the simulation results are in good agreement with the measured data. The parameters that we tune are often referred to as tuners. Dymola varies the tuners and simulates when it searches for satisfactory solutions. Mathematically, the tuning procedure is an optimization procedure to minimize the error between the simulation results and the measurements.

When tuning parameters from measurements, a basic question is “Which parameters can be estimated from the measurements available?” Changing a parameter to be estimated must of course influence the output. However, this is not enough. Two or several parameters may influence the result in a similar way such that it is not possible to estimate them individually. Dymola includes function to analyze and to plot parameter sensitivities. When a set of parameters have been tuned, it is recommendable to validate the model and the tuned parameters against other measured data to check that there is a good agreement between the simulation result and the new measurements. For a specific series of measured data it is possible to get good fits by increasing the model complexity and the number of tuned parameters. However, this does not guarantee that the result is that good for other operating conditions.

To load the package Design.Calibration, select **File > Libraries** and select **Design**. When the library is opened, expand Design and then Calibration by clicking on the > in front of them. The result in the package browser will be:

The functions of Design.Calibration.



The function Design.Calibration.calibrate is the main function for calibration and validation of models[17]. There is also a set of functions for analyzing parameter sensitivities and dependencies of calibration tasks. For parameter studies in general see the chapter “Model Experimentation” in this manual. The function Design.Calibration.calibrate supports easy setup of calibration to tune static characteristics.

The content of this chapter is the following:

In section 7.2 starting on page 1095 the basics of setting up and executing a basic calibration task is described with a number of examples based on a simple car model describing translational motion available in the Design library.

Section 7.3 starting on page 1118 describes how to store a setup for later reuse.

[17] The optimization method used by the function is a least-square fit with regularization to ensure that it does not get stuck due to redundant tuners.

Section 7.4 starting on page 1120 describes how to reuse a setup for a similar operation.

Section 7.5 starting on page 1121 describes a number of functions to analyze parameter sensitivities and dependencies. The functions sweepParameter, sweepTwoParameters, perturbParameters and checkCalibrationSensitivity are described in this section.

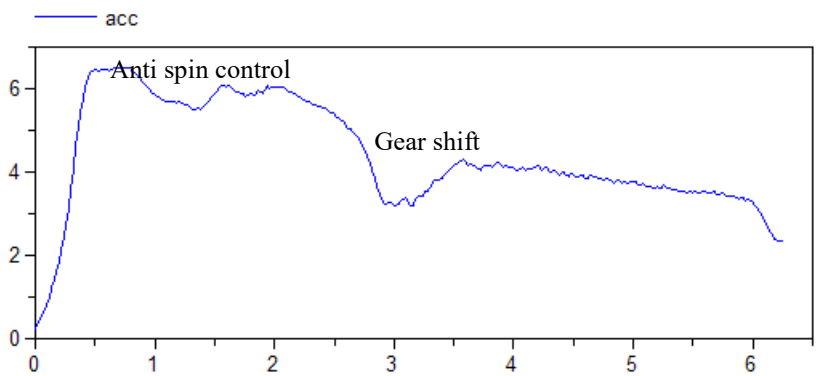
Section 7.6 starting on page 1135 describes data preprocessing, the process of adjust the data eliminating noise, zones where the model is not valid and erroneous or not representative measurements. The function used is dataPreprocessing.

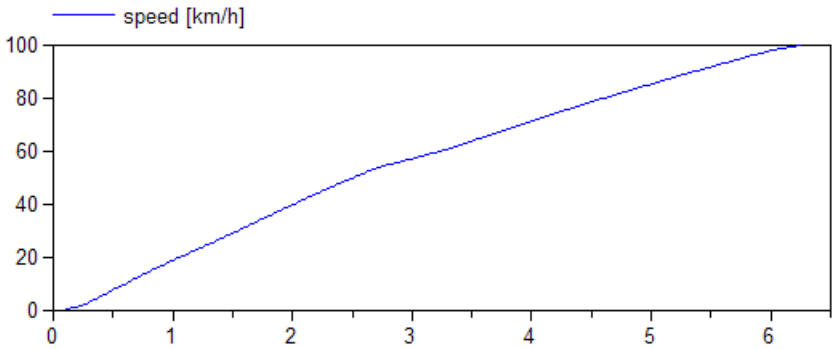
Section 7.7 starting on page 1146 describes static calibration. Two cases are described. One case is the calibration of completely static models (to tune static characteristics of components such as pipes, valves, throttles etc.). Such models are always in steady-state. The function to use is staticCalibrate. The other case is steady-state calibration to tune steady-state cases for which either dynamics is ignored or each case is simulated until steady-state is obtained. The function used is calibrateSteadyState.

7.2 The basics of setting up and executing a calibration task

We have acceleration and speed measurements from a BMW 645i at full throttle as shown in the plot below. For further information, we refer to Auto Mobil, Issue 1, 2005.

Acceleration and speed measurements from a BMW 645i at full throttle.





Anti spin control and gear shifting make the acceleration curve complex. Here we will focus on the time interval 3.8-6 seconds when the second gear is engaged.

We need to describe how the generated torque makes the car move. Thus we need to make a simple power train model including gearbox and rotating elements which make the wheels rotate.

7.2.1 Vehicle data

By searching on the web we can find the following data for the car

Data for a BMW 645i.

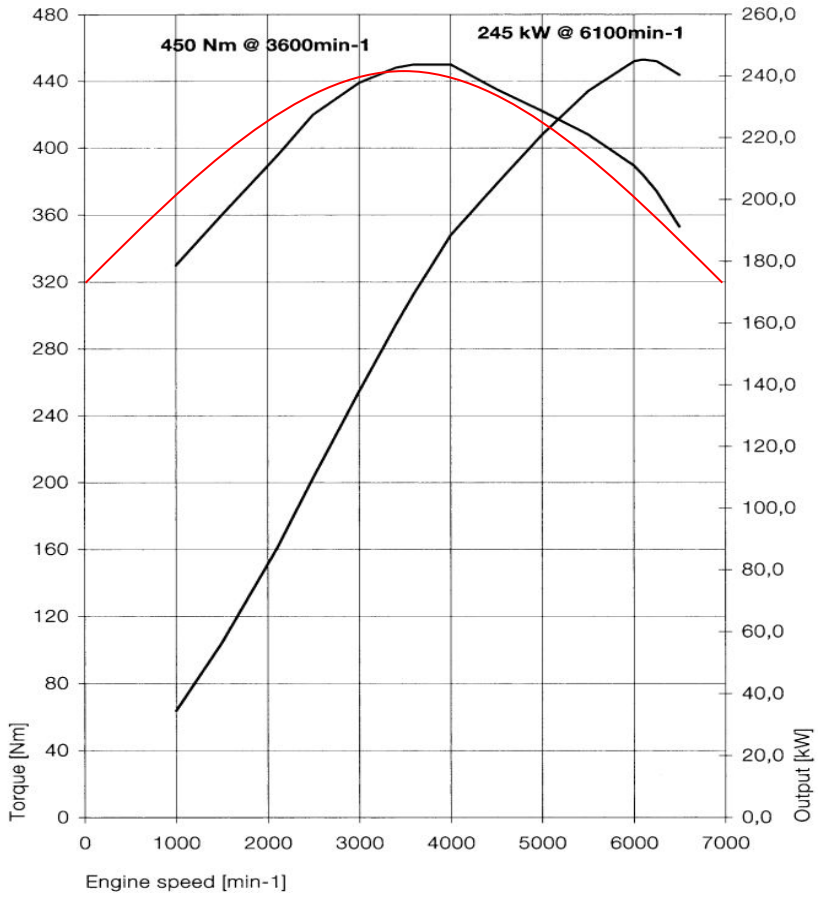
Engine torque at 3600 rpm [Nm]	450
Engine inertia [kgm^2]	0.4
Gearbox and cardan inertia [kgm^2]	0.01
Wheel inertia [kgm^2]	4* 1
Wheel radius [m]	0.34
Car mass [kg]	1690
Automatic gear ratios I-VI	{4.17, 2.34, 1.52, 1.14, 0.87, 0.69}
Gear ratio of final gear	3.46

The wheel radius is calculated for 245/45 R18 W saying that the radius is 18”/2 + 0.45*245 = 0.338 m.

Engine characteristics at full throttle for a BMW 545i were found at

`http://www.e60.net/information/options/engines/my2004_545i`

Engine characteristics at full throttle for a BMW 545i.



BMW 545i and BMW 645i have the same 4.4-liter V8 engine. The black lines in the plot above show the torque and power characteristics.

As a first approximation we fit a quadratic characteristic:

```
tau = tau_0 +(tau_max-tau_0)*(1-((w-w_max)/w_max)^2);
```

The parameter w_max is 3600*2π/60 rad/s and tau_max is 450 Nm. Choosing tau_0 to 320 gives the red curve in the plot above.

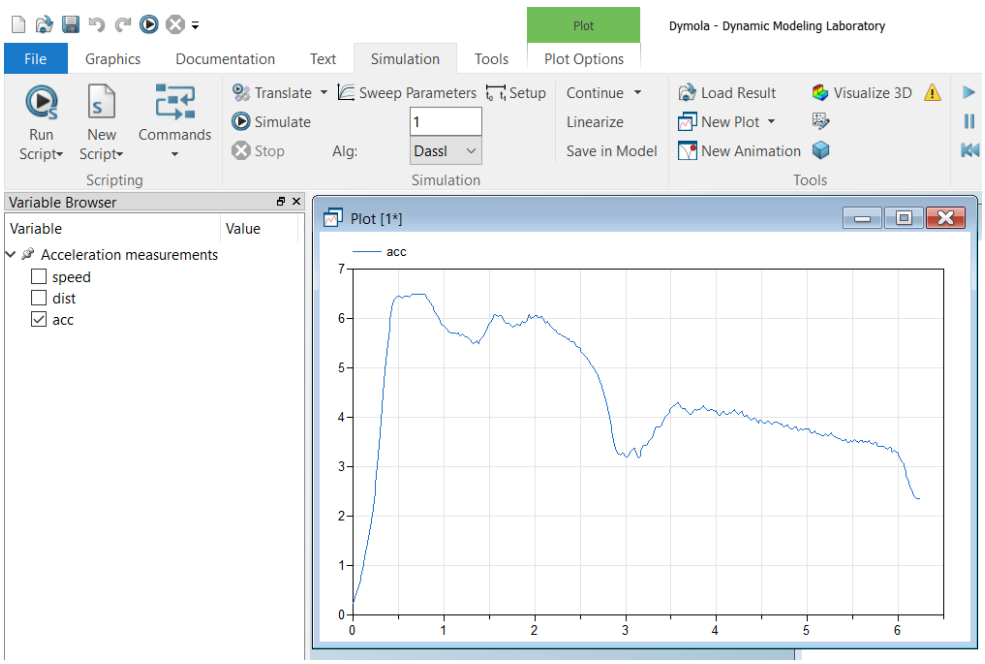
The velocity and acceleration measurements are stored simply as a csv file in `Program Files\Dymola 2023\Modelica\Library\Design 1.1.3\Acceleration measurements.csv`

The measurements stored as a csv file.

	A	B	C	D
1	time	speed	dist	acc
2	0	0	0	0.22
3	0.02	0.2	0	0.33
191	3.78	68.1	37.84	4.14
192	3.8	68.4	38.22	4.16
193	3.82	68.7	38.6	4.13

The first row of the file includes the column headings and then the data follow. Dymola supports plotting of such a csv file. Select **Simulation > Load Result** and a file browser pops. Use it to select the csv file (you have to select the file type "Comma separated values"). The file and its variables appear in the variable browser and can be plotted in the usual way.

Plotting the measured data in Dymola.



7.2.2 Vehicle model

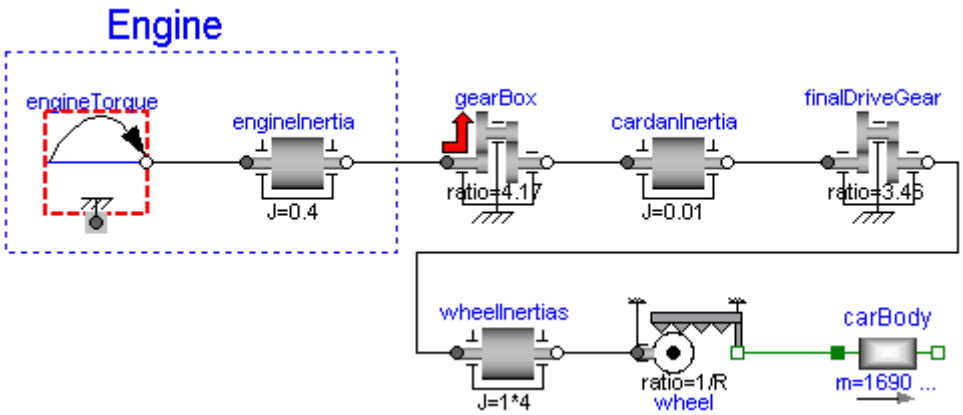
The model we are going to build is available as:

```
Design.Calibration.Examples.SimpleCar
```

Useful modeling components are found in

```
Modelica.Mechanics.Rotational
Modelica.Mechanics.Translational
```

The vehicle model.



To the left there is the engine driving the gearbox, which is connected to the cardan system giving a final drive to the four wheels. The rotational motion of the wheels results in a translational motion of the car. Let R be the wheel radius then 1/R gives the ratio between the driving rotational motion and the resulting translational motion where R is the wheel radius. The model defines

```
parameter Real R=0.34;
```

and binds the parameter wheel.ratio = 1/R. Setting of parameters are indicated by the diagram. Additionally the mass of the car, carBody.m is set to 1690+70+50 kg to include the weight of the driver and measurement equipment.

The quadratic torque characteristics at full throttle is modeled by extending from

```
Modelica.Mechanics.Rotational.Interfaces.PartialSpeedDependentT
orque
```

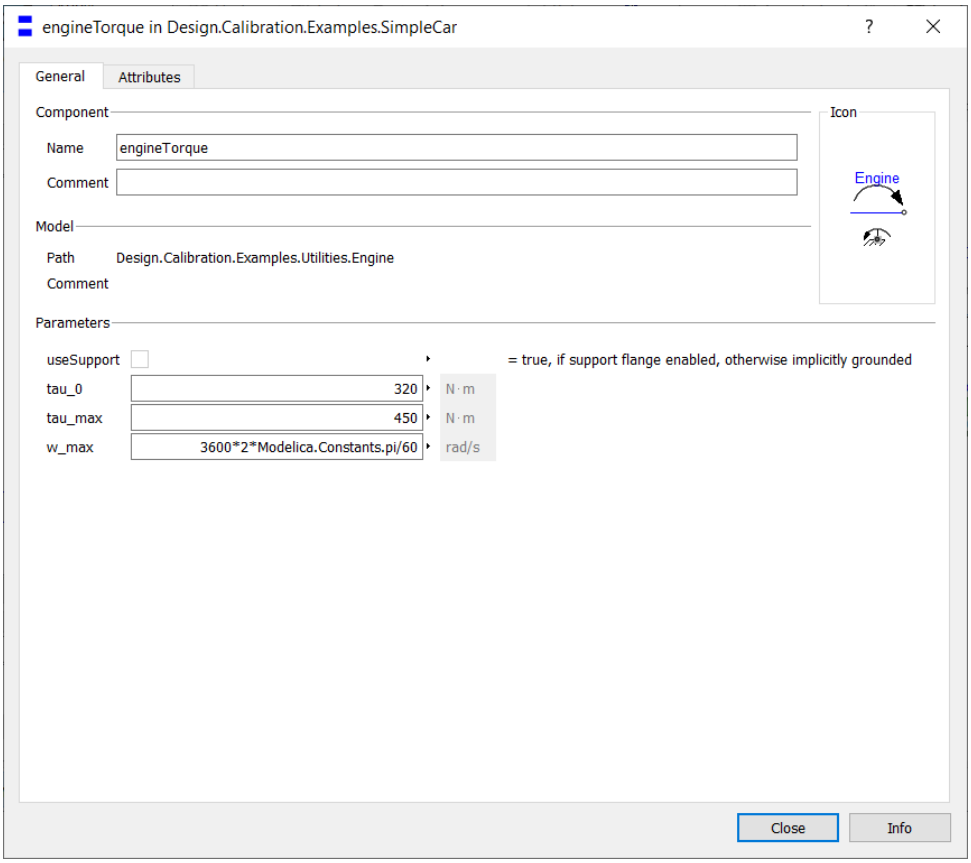
and adding the quadratic torque characteristics and the definitions of its parameters

```
model Engine
  extends
Modelica.Mechanics.Rotational.Interfaces.PartialSpeedDependentT
orque;
  parameter Modelica.SIunits.Torque tau_0;
  parameter Modelica.SIunits.Torque tau_max;
  parameter Modelica.SIunits.AngularVelocity w_max;
equation
  tau = - (tau_0 + (tau_max-tau_0)*(1-((w-w_max)/w_max)^2));
end Engine;
```

Please, note minus sign for the torque to specify that the torque is a driving torque and not a reaction torque.

The parameters of the component engineTorque are then set as shown by its parameter dialog (double-click on the component or right-click on it and select **Parameters**):

The parameter settings for engineTorque.



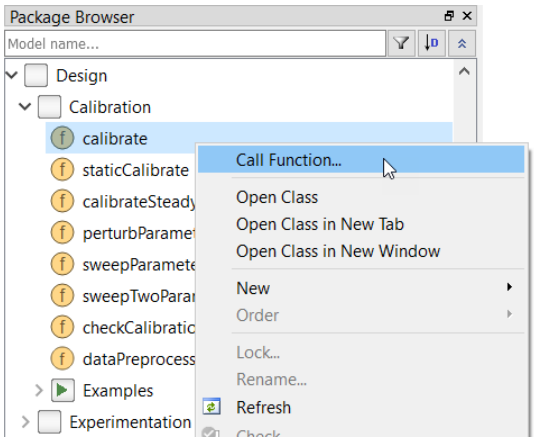
7.2.3 Validation of the nominal model

Let us first check how the model with nominal parameters compares with measured data. Validation is set up very similar to calibration. A basic difference is of course that no tunable parameters need to be specified for the validation. The functions described in this document are parts of the Design package. If you have not loaded the Design package by now, please see the section “Introduction” above on how to do it.

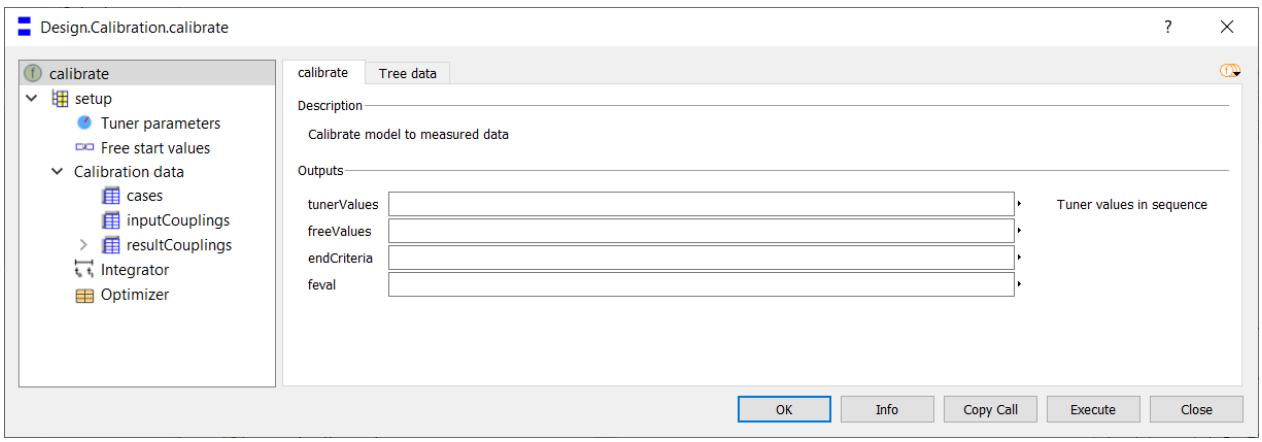
The following section corresponds to using the command **Simulation > Commands > Validation of original model**.

To set up the calibration, select Design.Calibration.calibrate in the package browser. Right-click and select the command **Call function…**

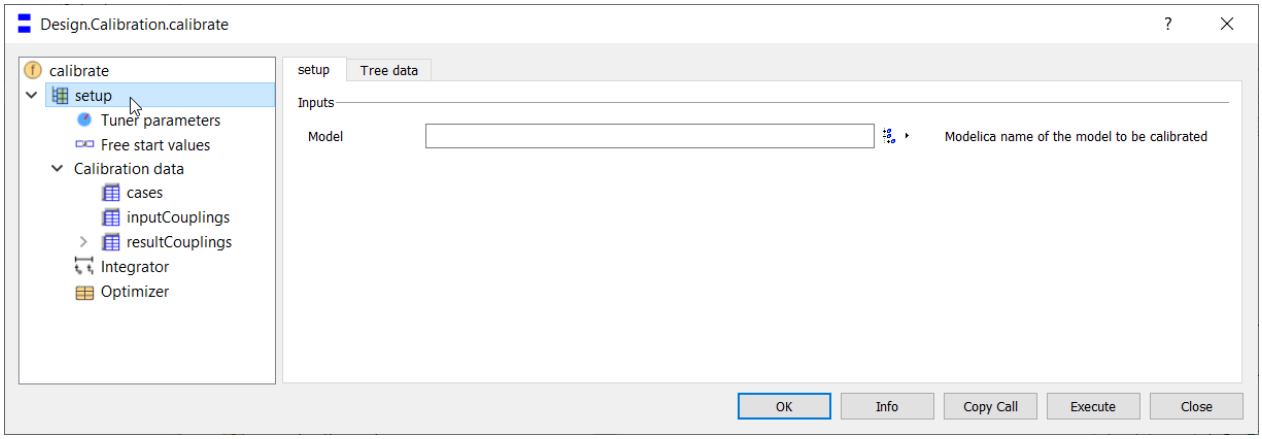
Selecting calibrate and right-clicking.



The following menu pops up:

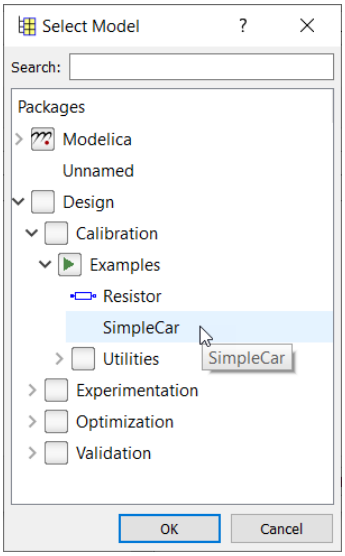


However, we want to start specify the model to be calibrated, so select **setup** in the pane to the left in the dialog; you will get:

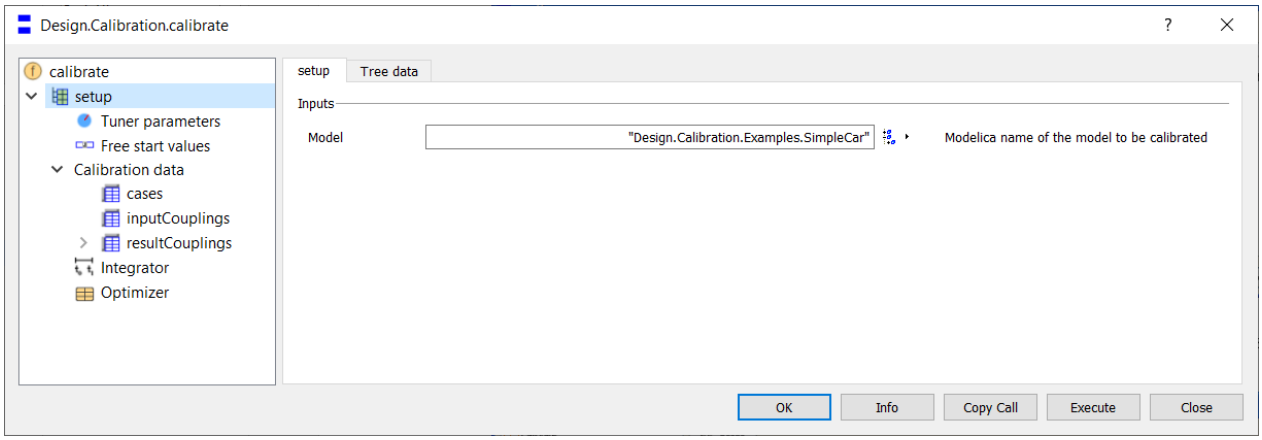


Selecting model to calibrate.

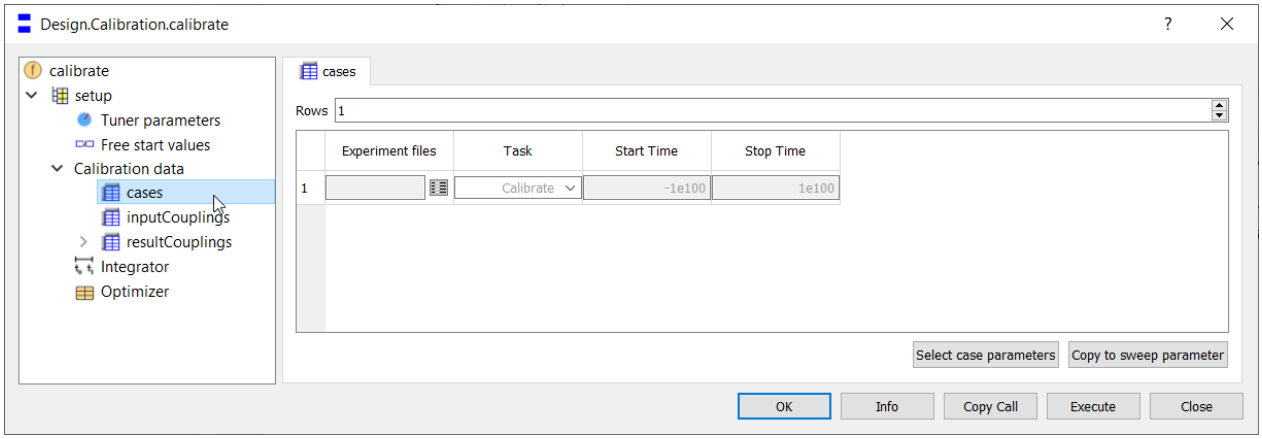
To specify the model to be calibrated, click on the Browser icon to the right of the input field. A package browser pops up. Use it to select the model.



Click **OK**. The model is now translated in order to gather information needed to build browsers and selectors to support the remaining setting up of the calibration task. If Dymola already has a translated model, then this model appears as the default model.

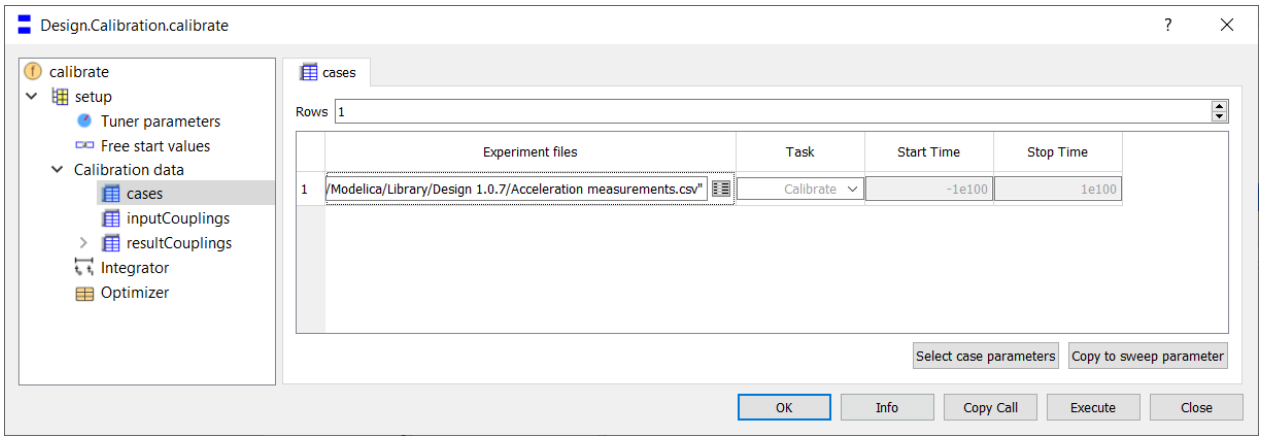


The next task is to specify the measurements and how they are stored. Consider the tree browser to the left. Select **cases** under **Calibration data**.





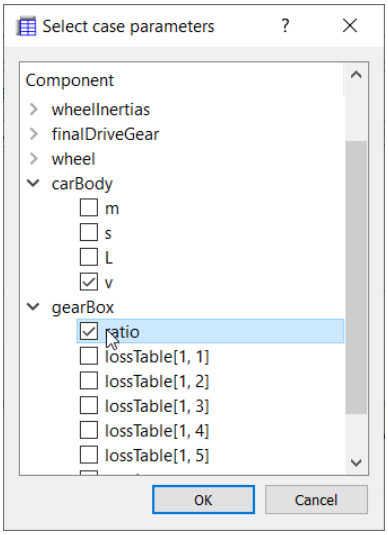
To introduce an experiment file, click on the **Edit** icon of the first element in the **Experiment files** column. A file browser pops up. Use it to select the file *Program Files*\Dymola 2023\Modelica\Library\Design 1.1.3\Acceleration measurements.csv.



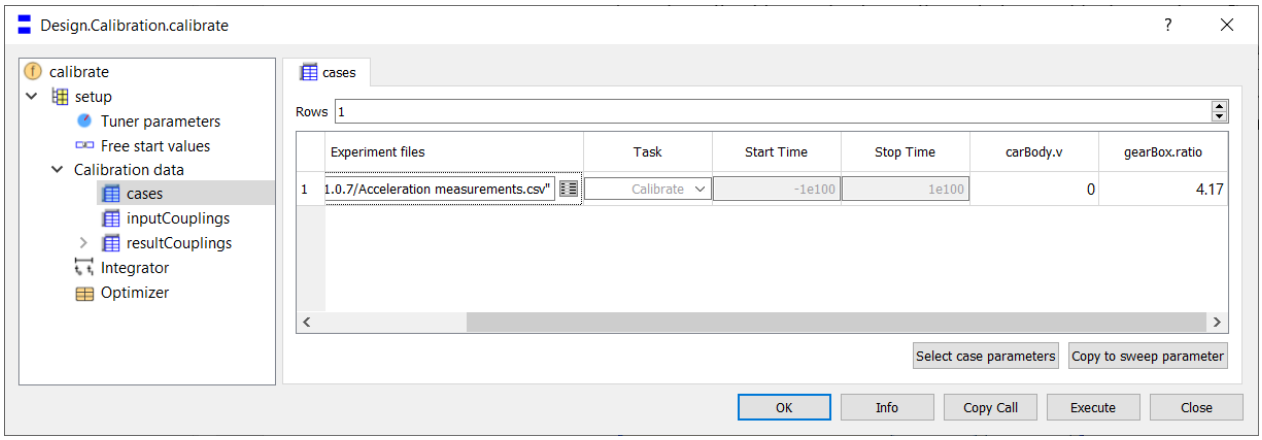
If we had had more measurement files we had increased the number of rows and selected more measurement files. In this case the measurements are stored in a csv file as described above. Dymola supports some common ways of storing measurements, see further below. An advanced user can replace the calibration data input with a routine accessing data in different formats, without having to change the underlying calibration routines.

The different cases may need individual parameter settings or individual initial values for some of the states. Recall that we are to use the measurements from the time interval 3.8-6 seconds when second gear is engaged. Thus we need to use the gear ratio of the second gear and an initial velocity. To enter this information, click on **Select case parameters**. Use the browser to select carBody.v and gearBox.ratio.

Specifying case dependent parameters.



Click **OK**. The default values appear in the new columns.

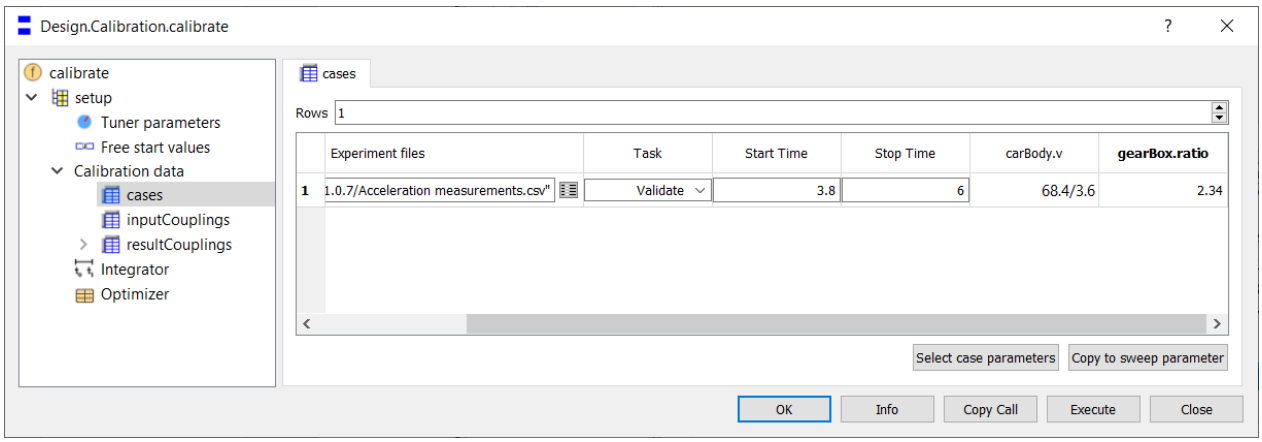


From the measurement file we can find that the velocity at time 3.8s is 68.4 km/hour = **68.4/3.6** m/s.

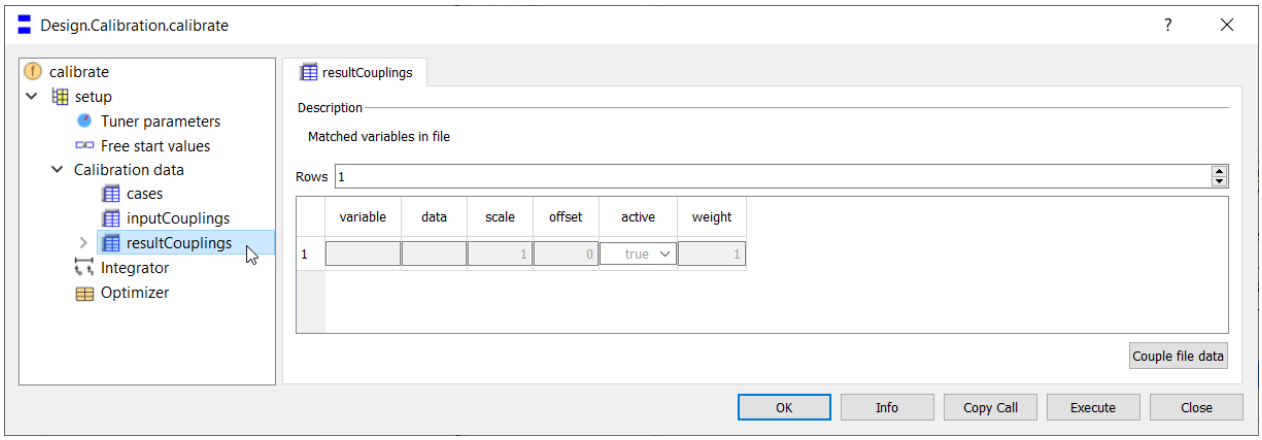
A part of the csv file with the measurements.

	A	B	C	D
1	time	speed	dist	acc
191	3.78	68.1	37.84	4.14
192	3.8	68.4	38.22	4.16
193	3.82	68.7	38.6	4.13

Enter this value for carBody.v and set gearBox.ratio to have the value of the second gear, namely **2.34**. Enter also start time (**3.8**) and stop time (**6**) and set **Task** to **Validate**.

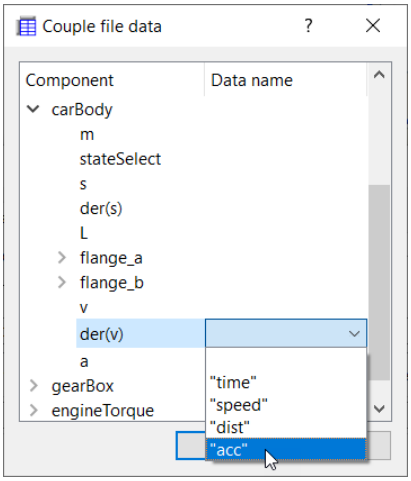


The files may include input signals to drive the model, parameter values to be used and measured data that the model shall reproduce. In this case the file includes measured speed, distance and acceleration for each 20 ms in the time interval 0-6.24 seconds. The acceleration measurements will be used for the calibration criterion. To specify that click on **resultCouplings** in the browser to the left

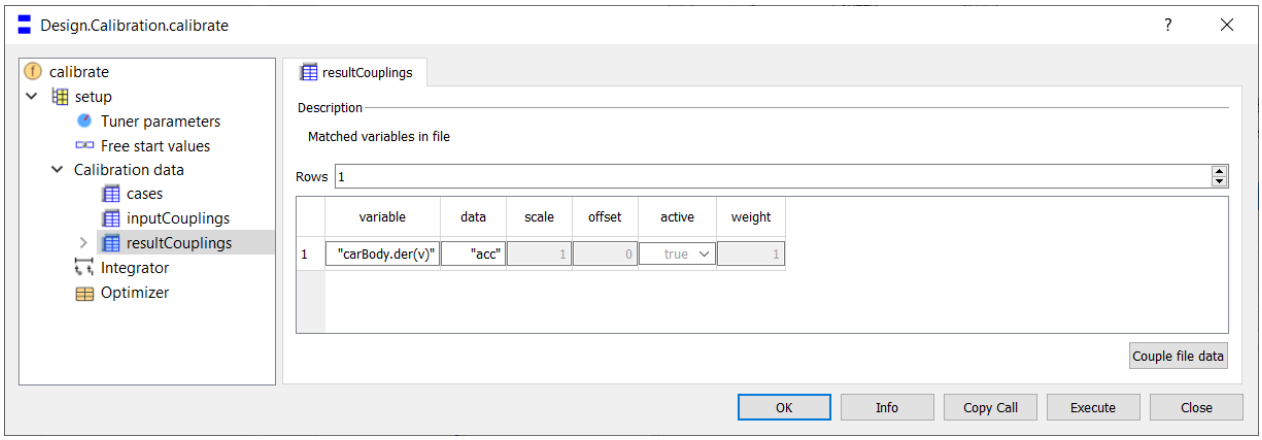


Click on **Couple file data**.

Couple the accelerator variable carBody.der(v) with measured acceleration "acc".



Use the browser to select the car acceleration, **carBody.der(v)**, and then right-click click to the right to see the names of the data series in the input files. Select **"acc"**. We could also have chosen carBody.a, because carBody.a = carBody.der(v). Click **OK**.

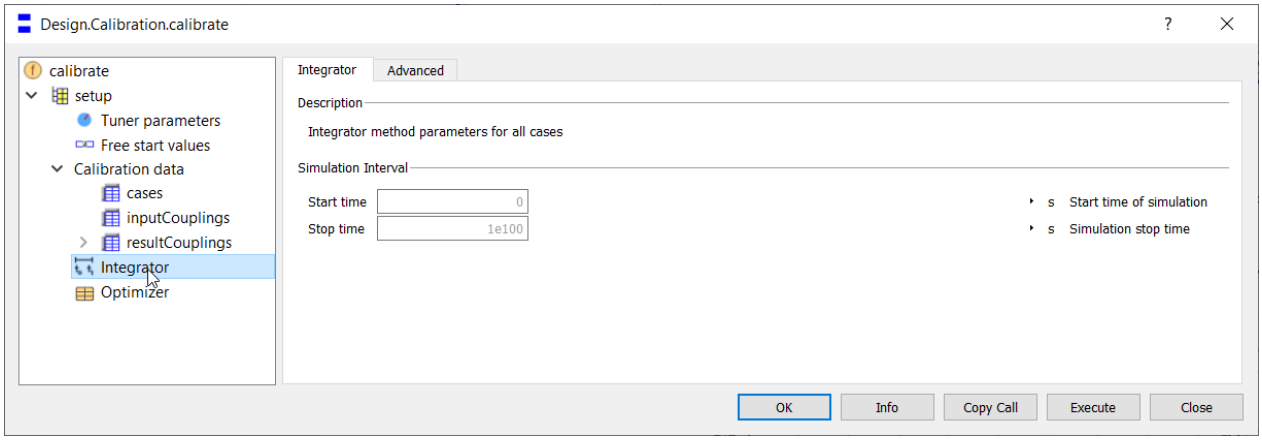


We have now specified that the difference between carBody.der(v) and the data column “acc” shall be used as the criterion for calibration. If the measured data are given in some unit different than that used in the model, the **Scale** column allows scaling of the measurements: variable = data * scale + offset

In case the deviations of several variables shall be used to specify the criterion, the **Weight** column allows the user to give them different weights.

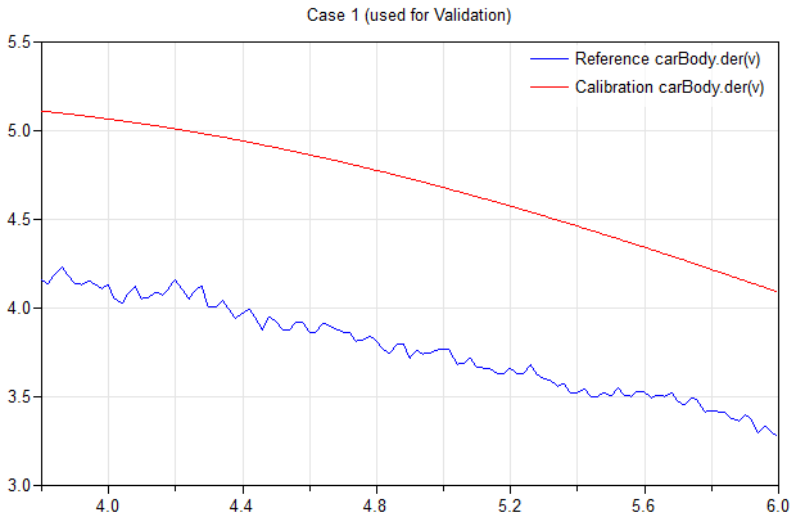
The model SimpleCar has no inputs. In case the model has inputs, click on **inputCouplings** and couple them to the file data in a similar way as done for the outputs.

The **Integrator** element allows specification of a global simulation interval.



To perform the validation, click **Execute**.

Comparing measured acceleration with simulated acceleration before calibration.



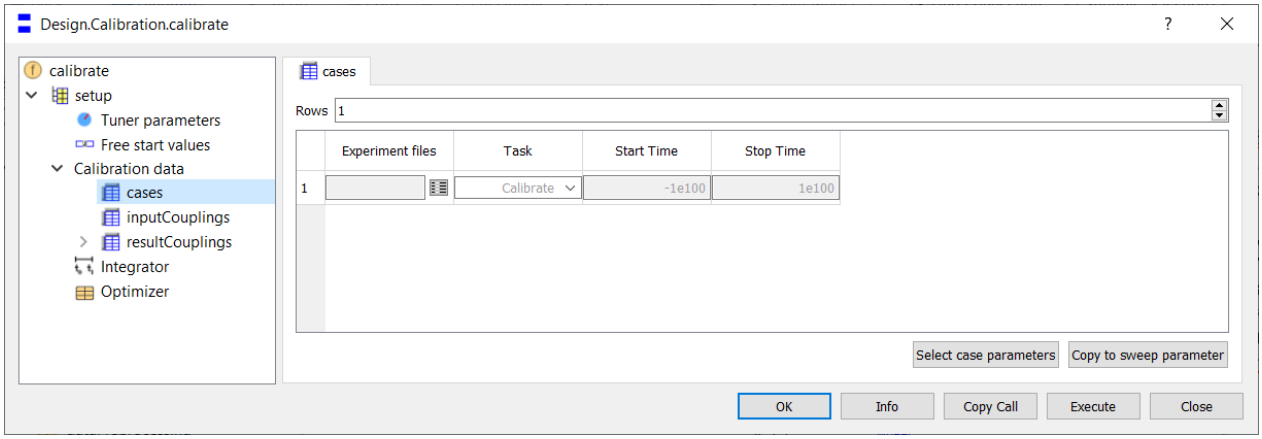
The result is plotted above. The curves have similar shapes, but there is an offset. The model gives a higher acceleration than measured. This may make you think of losses not being modeled. Soon we will discuss calibration – please do not shut down any window, the next section "Measurement file formats" describes how working with a Matlab file differs from working with a csv file. If you want to continue with calibration etc. directly, please jump the next section.

(If you by mistake have shut down the window, please see the tip to get back in section "Saving the setup for reuse" on page 1118.)

7.2.4 Measurement file formats

In the example above the measurements are stored in a csv file as described. Dymola supports some common ways of storing measurements and allows users to specify their own storage formats. The measurement files must have the same format.

In case the measurement data are stored in (Matlab 4) mat files, we need to specify the name of the matrix containing the measurement data to be used and the data are referred by column number. The acceleration measurements are also available as a .mat file. Let us use this file instead.

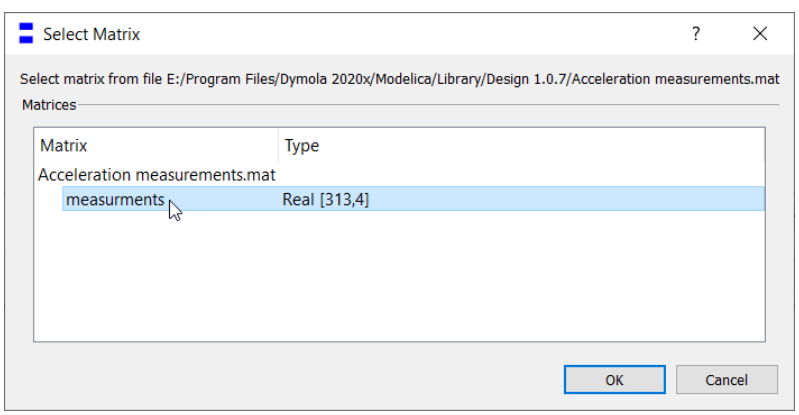


As previously, click on **cases**. Click on the **Edit** icon of the first element in the **Experiment files** column. A file browser pops up. Use it to select the file

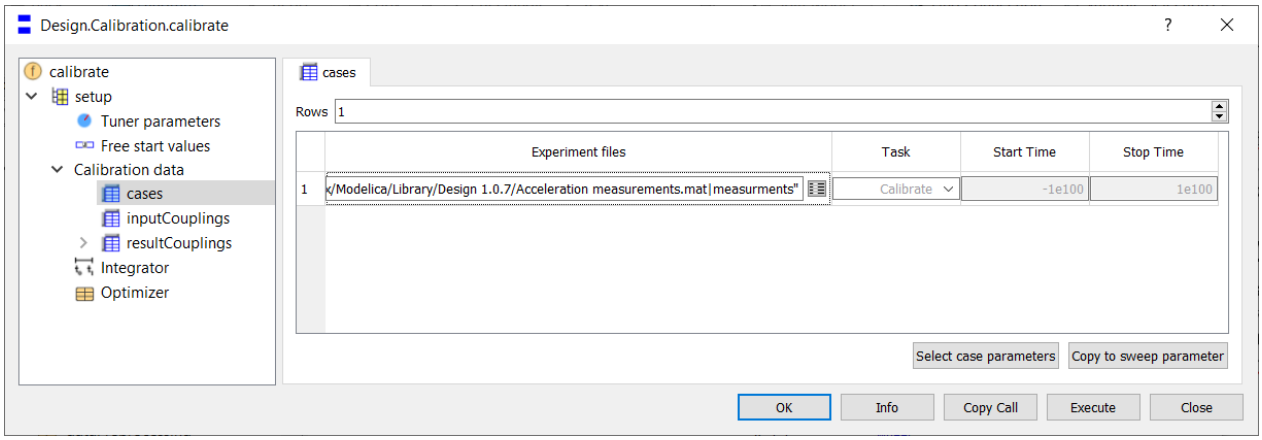
```
…\Program Files\Dymola 2023\Modelica\Library\Design
1.1.3\Acceleration measurements.mat
```

Dymola then pops a menu to select the appropriate matrix. After selection (in this case no alternative is possible) it will look:

Selecting matrix.

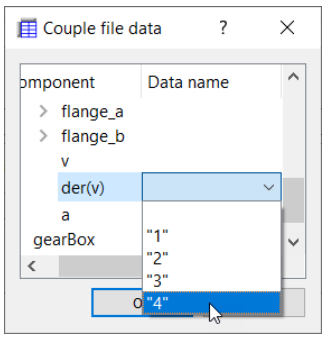


Click **OK**.

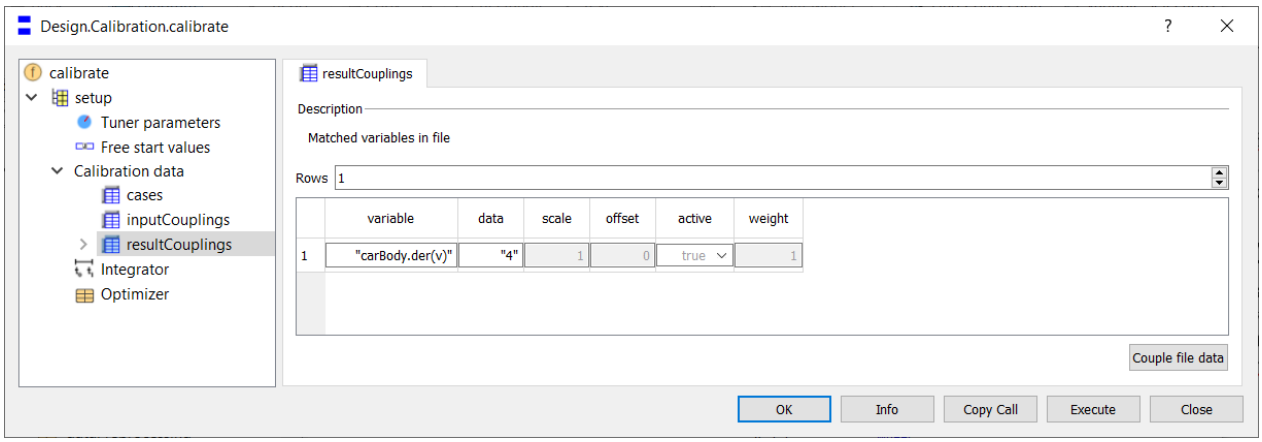


Proceed as previously to select case parameters, setting their values and start and stop time. The specification of result couplings is slightly different, because the data is referenced by column number. The acceleration measurements is column 4.

Couple model variable and data.



The result of the coupling now becomes



The data field has **"4"** instead of **"acc"**.

The simulation results of Dymola are stored as .mat files, which includes information on the name of the variables. If such trajectory files are used as measurement files, then the information on variable names are used. The user will not be prompted for matrix name. When coupling inputs or results, the browser will display variable names.

7.2.5 Calibration

The task of a calibration is to tune some parameters to obtain a better agreement between measured behavior and behavior predicted by the model. Thus, we need to address the question, which parameters to tune. When deciding which parameters to tune, it is good to consider the question: Which parameter values are most uncertain? In the model above, friction and losses in the gearbox elements have been neglected. Frictions and other losses are good examples where calibration is useful. There are for instance losses in both gearBox and finalDriveGear, however, having only measurements of the translational motion of the car, it is not possible to decide the individual losses of these two elements. Thus, it is necessary to aggregate all losses to one element and gearBox is selected, since it has provisions to model efficiency. The efficiency is given by gearBox.lossTable[1,2], see the documentation of Modelica.Mechanics.Rotational.Components.LossyGear.

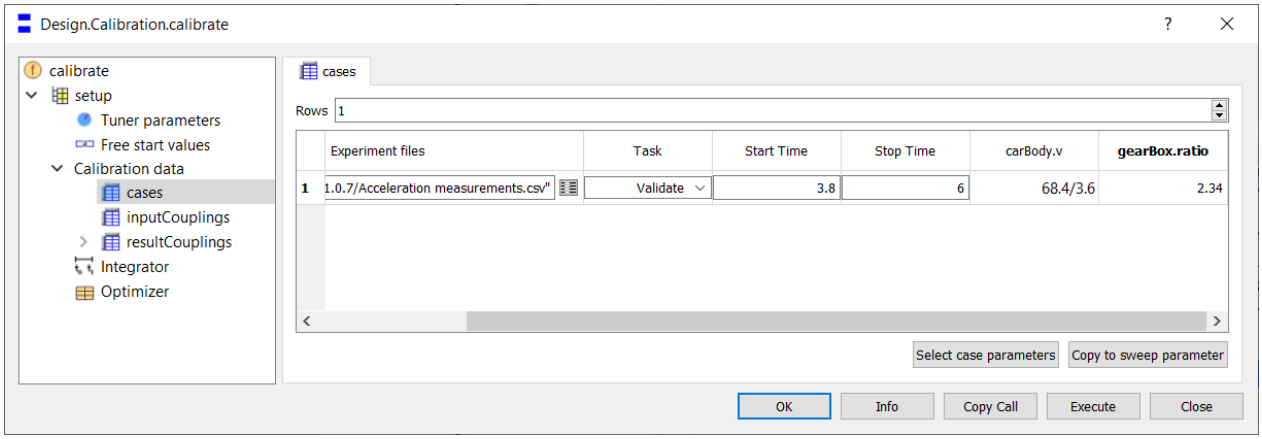
The parameter tau_0 was manually selected to 320, so it is a good candidate for tuning.

Dymola supports an interactive explorative approach to this problem. Dymola has powerful functions to perform parameter sweeps and to analyze parameter sensitivities and possible couplings between parameters with respect to the result variables to eliminate irrelevant parameters and to diagnose over- parameterization. However, let us come back to these later and first try tuning the two parameters.

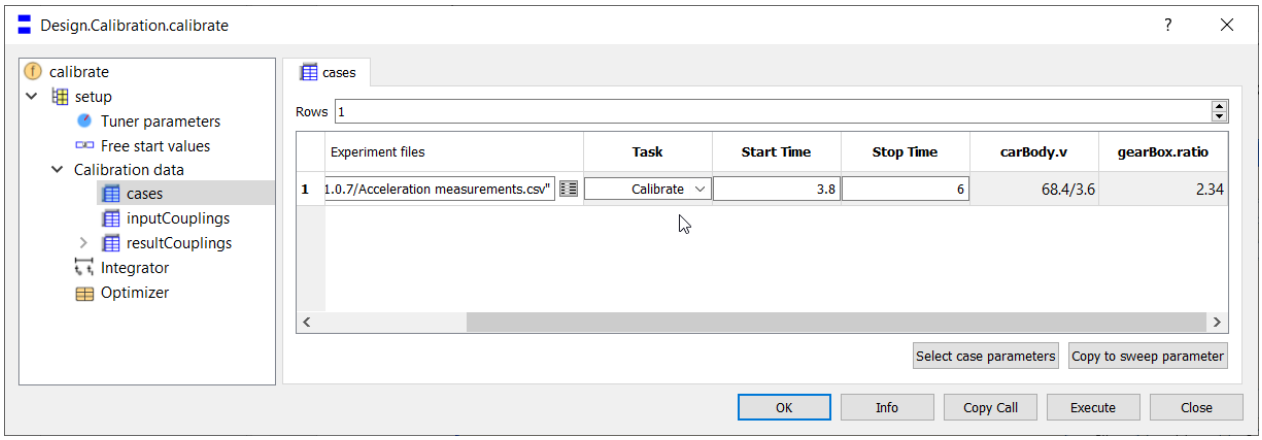
We continue with the initial example where a .csv file was used as measurement file. If you happen to have shut down the window below, you can use the command **Simulation > Commands > Validation of original model** to get back the needed setup

The final result of this section ("Calibration") and the section "Validation using measurements from first gear" below can also be obtained using the command **Simulation > Commands > Calibration with validation**.

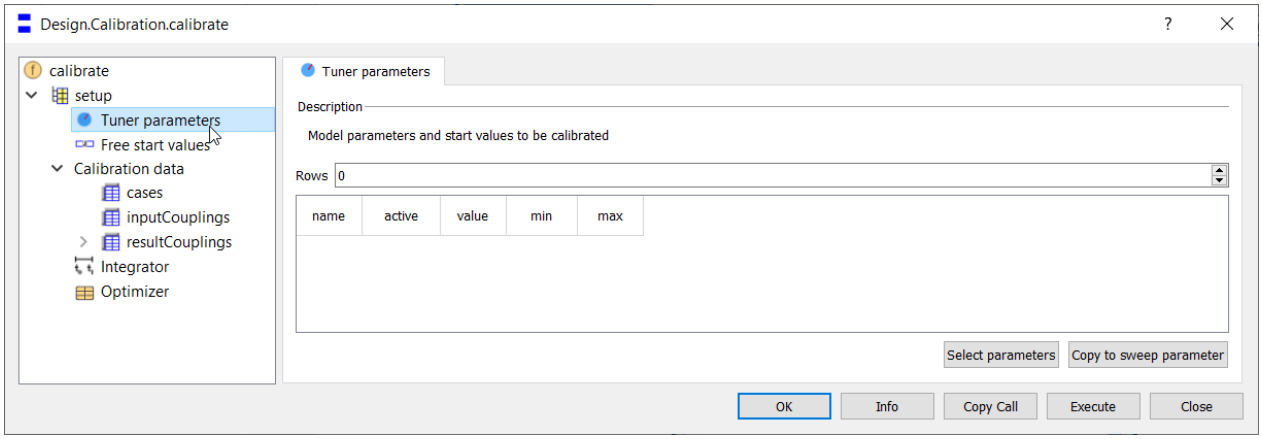
Clicking on **cases** in the browser of that window should give:



First we have to set the task to Calibrate. Click on **cases** (if not done already) in the tree browser to the left and then set the column **Task** to **Calibrate**.



Select **Tuner parameters** in the browser.



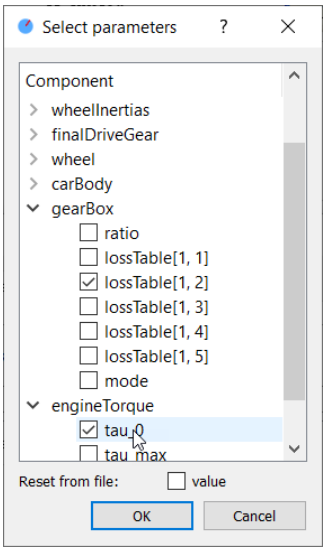
Click **Select parameters**. A menu pops. Select

gearBox.lossTable[1, 2]

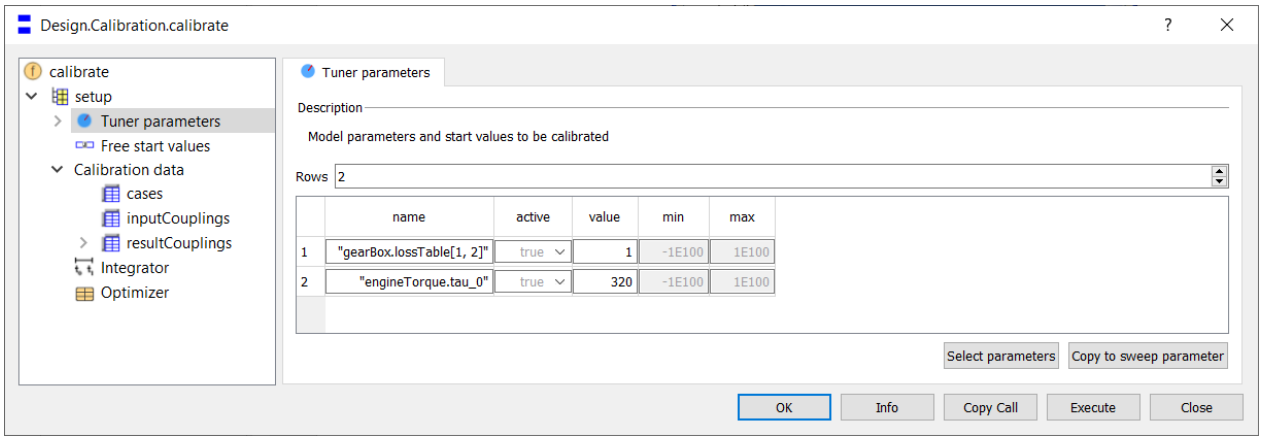
and

engineTorque.tau_0

Selecting parameters to tune.



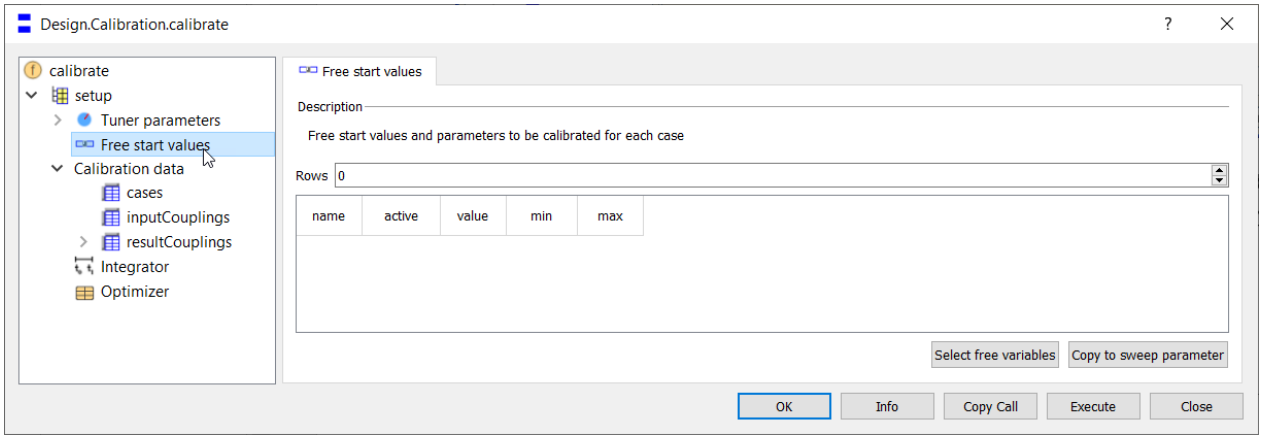
Click **OK**. The result will be



Please do not close any window, continue to the next section.

7.2.6 Free start values

The start value of a state may be unknown. By including the state as a tuner, the start value is estimated automatically. However, in case we have several measurement series, it may be necessary to tune or estimate these initial values individually for each calibration case. Dymola supports individual tuning of parameters and start values of states and they are specified as freeStartValues. Clicking on **Free start values** in the browser of the window will give:



Such parameters or states are selected by clicking **Select free variables** button which pops a variable selector as when selecting case parameters or tuners. The variable selector includes parameters and states. These values are also tuned for cases having Task = Validate.

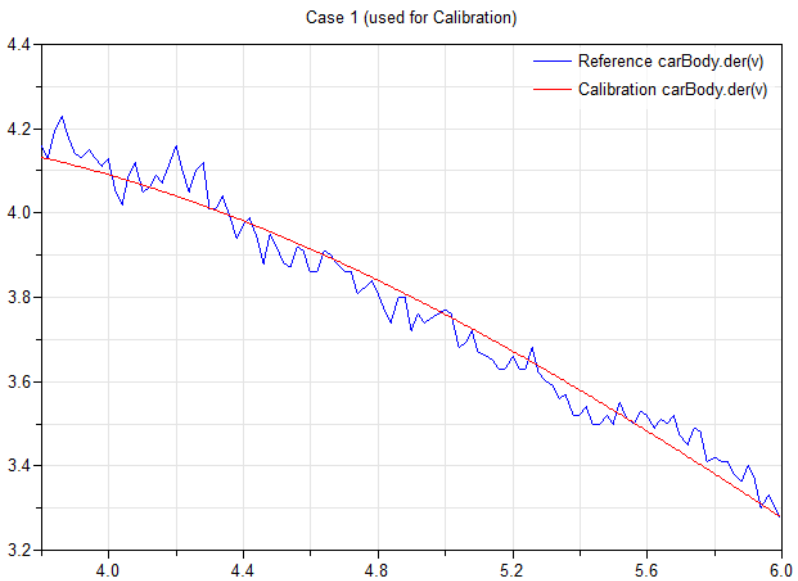
7.2.7 Tune the parameters

It is time to do the first calibration. Click **Execute**. During the calibration, results are plotted. After 25 fast iterations, we obtain the result.

The tuning result.

gearBox.lossTable[1, 2]	0.794
engineTorque.tau_0.	260.7
criterion	0.218

Comparing measured and simulated acceleration after tuning.



A passenger car has normally an efficiency of 0.90 at high gears in normal operation. The measurements are made at full throttle to give maximum acceleration. It means for example that the tires are slipping say 4%, which of course is increasing the losses.

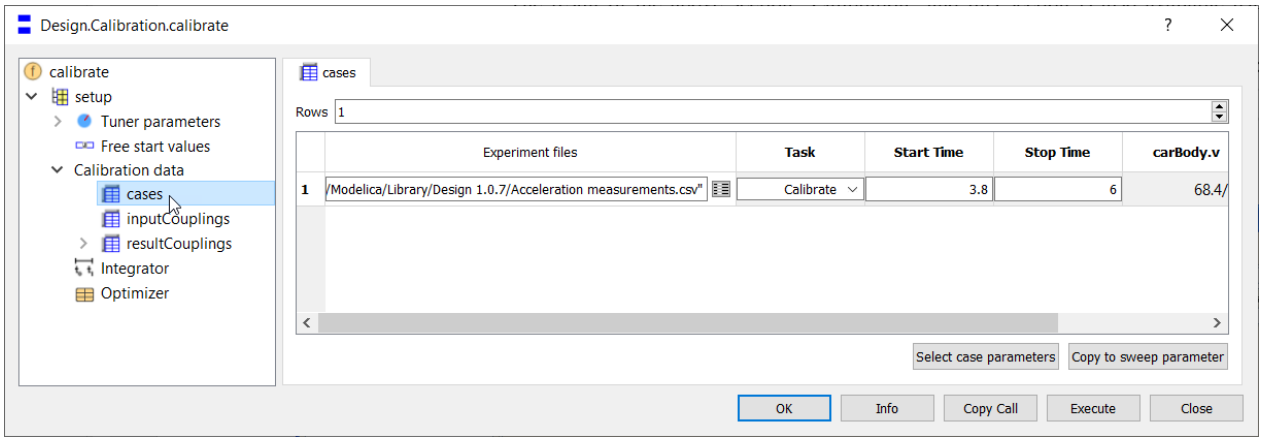
It is very easy to add new tuners. Just select **Tuner parameters**, click **Select parameters** and select new parameters. By changing active from true to false or vise versa it is easy to experiment with different set of tuners. Having a parameter as an inactive turner is a good way to set a parameter to have a value different from the value given by the model.

7.2.8 Validation using measurements from first gear

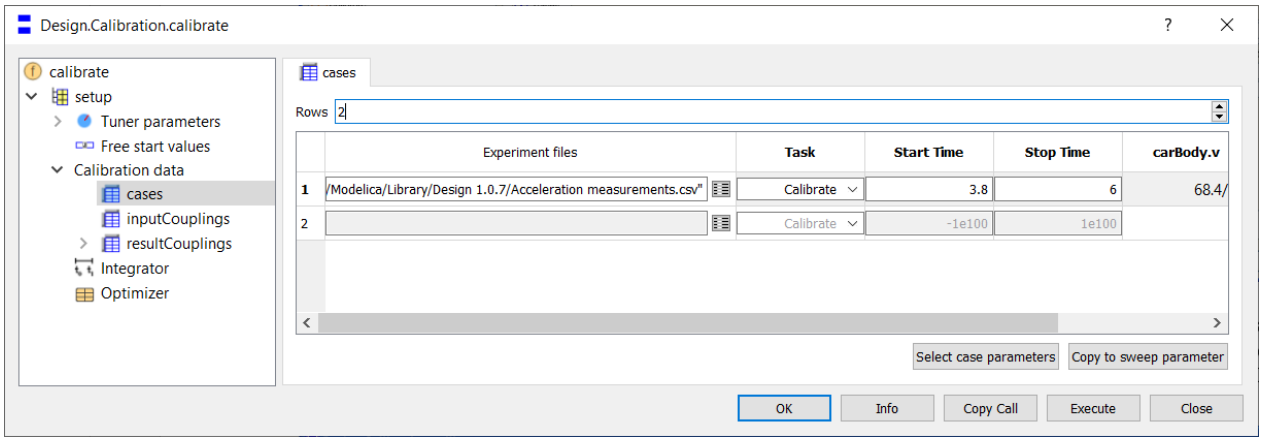
The result of the above section “Calibration” and this section is also available using the command **Simulation > Commands > Calibration with validation**.

It is recommended to validate against other measuremnts. Unfortunately, we do not have another measurement series in this case, but for validation we can use the data from the time interval where first gear is used.

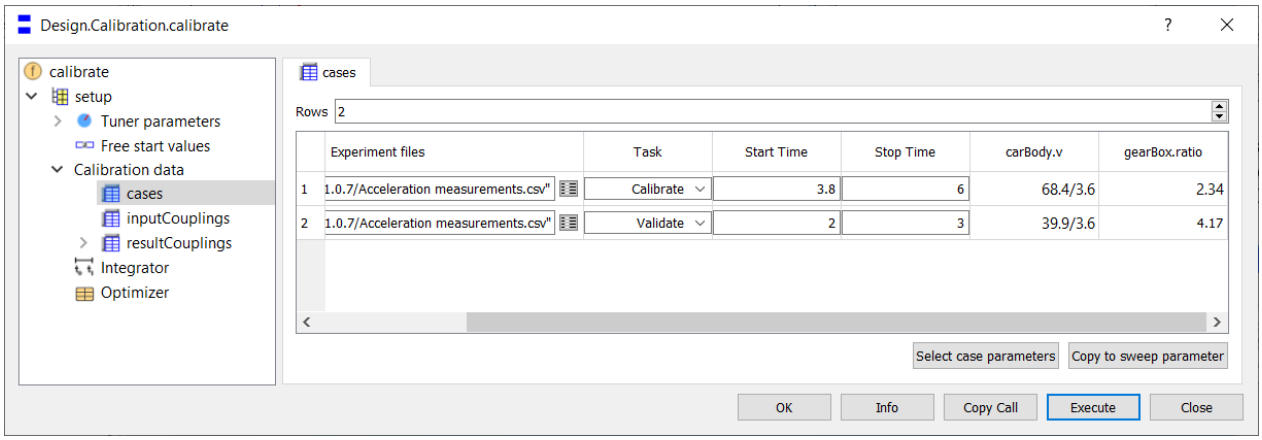
We do this by specifying another case. Click on **cases**.



Put the cursor in the input field for **Rows** and press the arrow up key on the keyboard once to increase the value by one. You may also use the arrow up of **Rows** to increase Rows to 2.

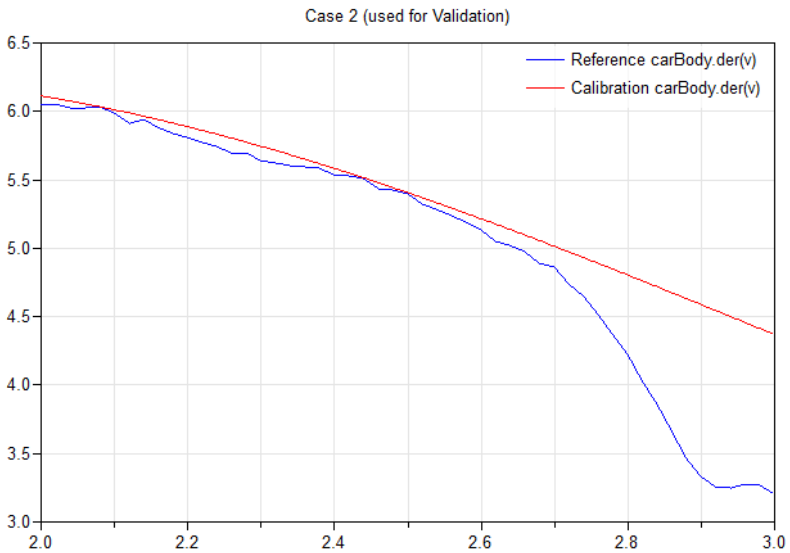


As previosly, use the **Edit** button of Experiment files to select experiment file. You can also copy and paste the file name (using **Ctrl + C** and **Ctrl + V**). Enter the values for start time (2.0), stop time (3.0), carBody.v (39.9/3.6) and gearBox.ratio (4.17) as illustrated below. Do not forget to set Task to Validate.



Click **Execute**. The calibration starts and gives the same results as previously, but also the plot below for the validation case (the criterion is 13.88).

Comparing measurement and simulated acceleration to validate mode.



The agreement for the interval 2.0-2.7 s is very good. If we rerun the validation having set the stop time of the second case to 2.7, the criterion is 0.44. As indicated above the tires are slipping when the car is run to accelarate as fast as possible. If the wheels slip too much, the anti spin control system gets active and the result is reduced acceleration after 2.7 seconds as shown by the measured data.

As illustrated, Dymola supports a flexible and incremental way of working. We need not define this total setup in one step. First we made the model and validated the nominal model against the measured data, then selected turners and calibrated. Finally we validated the calibrated model. Dymola also provides support for sentivity analysis (see below).

7.2.9 The setup as Modelica code

The calibration setup is represented in Modelica in the following way. It is a function call, where nested record constructors build the needed input arguments.

```
Design.Calibration.calibrate(Design.Internal.Records.ModelCalibrationSetup(
  Model="Design.Calibration.Examples.SimpleCar",
  tunerParameters={
    Design.Internal.Records.TunerParameter(name= "gearBox.lossTable[1, 2]", Value=1),
    Design.Internal.Records.TunerParameter(name="engineTorque.tau_0", Value=320)},
  calibrationData=Design.Calibration.Internal.Dynamic_common(
    Design.Internal.Records.DynamicCommonCalibrationCases(
      experimentNames={
"E:/Program Files/Dymola 2023/Modelica/Library/Design 1.1.3/Acceleration
measurements.csv", "E:/Program Files/Dymola 2023/Modelica/Library/Design
1.1.3/Acceleration measurements.csv}",
      task={1,2},
      startTime={3.8,2},
      stopTime={6.0,3},
      parameterNames={"carBody.v","gearBox.i"},
      parameterValues=[68.4/3.6,2.34; 39.9/3.6,4.17]),
  resultCouplings={Design.Internal.Records.DynamicCalibrationResultCoupling(
    variable="carBody.der(v)", data="acc")})))
```

7.3 Saving the setup for reuse

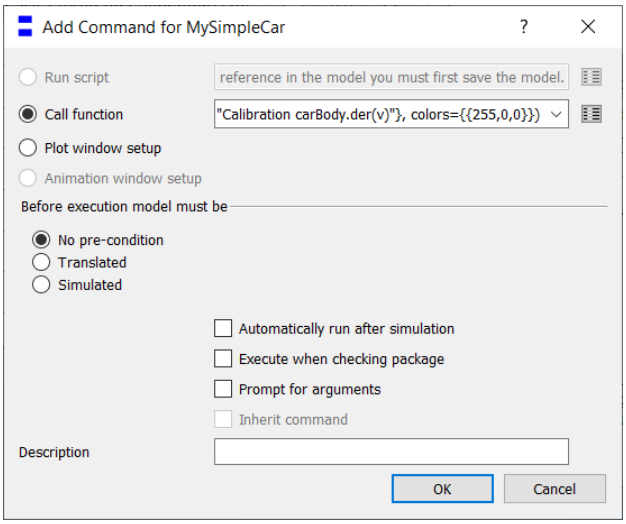
After an execution of a command we can save it in the model for later reuse. However, this cannot be done with the SimpleCar model we have used so far since it is read-only. To investigate saving the setup, we want to work with an editable model.

Do not close the calibration dialog, we will use that one later.

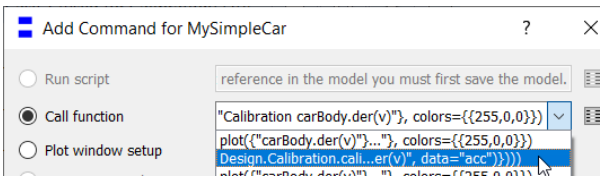
Create an editable copy of SimpleCar by by right-clicking SimpleCar in the package browser and select **New > Duplicate Class…**. Change the name to MySimpleCar and click **OK**. You have now created an editable copy to work with.

In the calibration dialog, click **setup** and use the browse button to select the model MySimpleCar, then click **Execute**. You will now get the same result as before, but this time you work with the MySimpleCar model.

Now you can save the setup in this model. To do that, select **Simulation > Commands > Add Command…**. A menu appears:

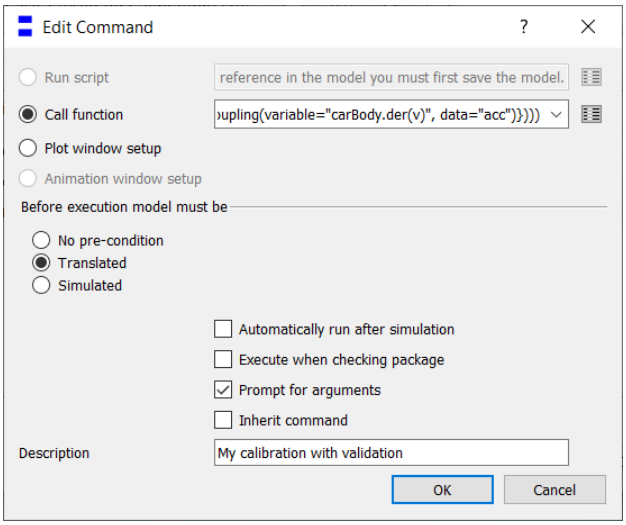


You want to save the calibration/validation function call, but the one shown is the last one given, which is the plot one. Click the arrow after the call, and select the second last one:



Tick **Prompt for arguments** and enter a description, which will be used in the commands menu. Since the model needs to be translated in order to get the select browsers we tick that model shall be **Translated**. This is not critical, only a matter of convenience. If we do not tick **Translated**, then when a browser needs to be popped, Dymola will give a prompt pointing out that the model needs to be translated. If we just select the command and then click **Execute** there will be no prompt, but function is executed as expected. The model is translated when needed. The **Edit** button next to the function call allows you browse or edit the function call once more.

Saving the calibration setup.

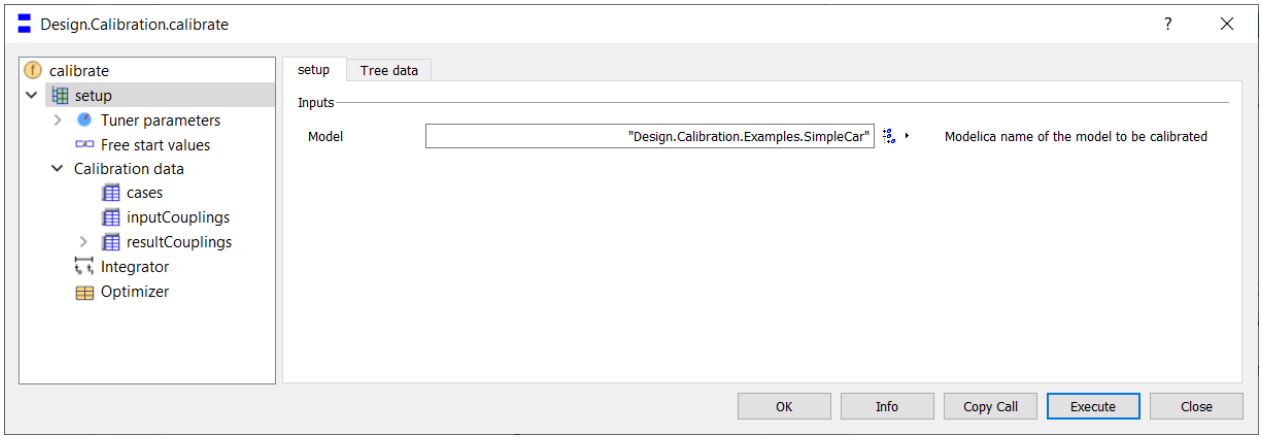


Click **OK**. You have now created you own command in the model; you can see it using the command **Simulation > Commands**. The command will be saved in the model when you save the model. (More about such commands can be read in chapter "Simulating a model", section "Editor command reference when working with a simulation", sub-section "Main window: Simulation tab".)

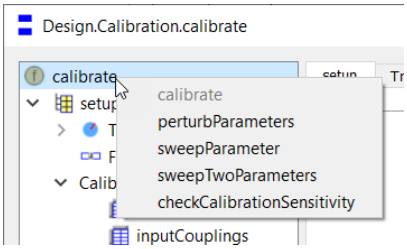
A function call menu as for calibrate has an **Execute** button. Clicking this button start an execution of the function and the menu stays popped. If we click **Close**, the menu is closed without any execution. If we click **OK**, the function is executed and the menu is closed. You click **OK** by mistake when you meant **Execute**, you can fix the situation. Click in the command input line. Press the arrow up once to scroll back in the commands given. Click right mouse button and select **Edit Function Call** and the function call menu pops. This can be done for any function call in the command log.

7.4 Reusing a setup for a similar operation

A setup can be reused for a similar operation. Assume that we just have made a calibration. The menu is then (if clicking on **setup** in the browser – we take also the opportunity go go back to the selection of the read-only model if you may have tried creating an editable model as above).



Select **calibrate** (at the top in the tree browser) and right-click to get the context menu.



The menu offers a selection of analysis and plotting functions that can exploit the calibration setup. We will describe these functions further below.

7.5 Analysing parameter sensitivities and dependencies

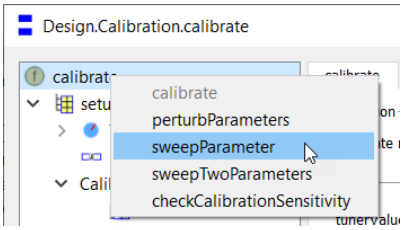
Dymola provides a set of functions to analyze parameter sensitivities and dependencies. Below the functions perturbParameters, sweepParameter, sweepTwoParameters and checkCalibrationSensitivity will be described.

7.5.1 Sweep one parameter – sweepParameter

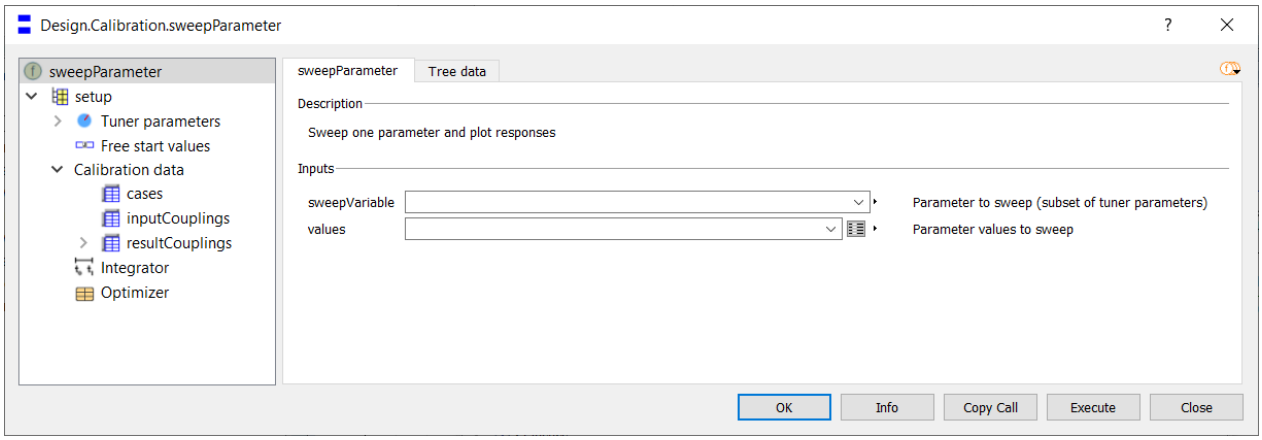
The function sweepParameter sweeps a tuner and plots the responses.

As an example open the model (if not already open) `Design.Calibration.Examples.SimpleCar` by using the command **File > Libraries > Design** and then using the package browser to open `Examples.SimpleCar`

Now select the command **Simulation > Commands > Calibration with validation**. When the calibrate window has opened, select **calibrate** (at the top in the tree browser *in that window*), right-click and select **sweepParameter**.

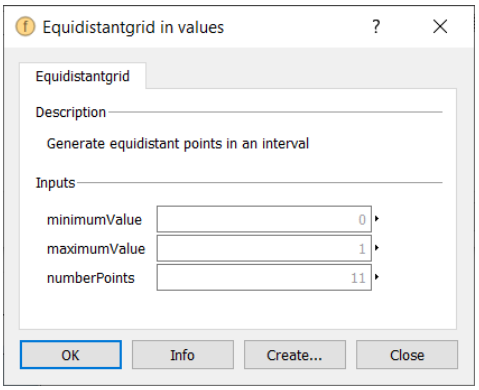


The sweepParameter menu changes since additional parameters needs to be provided.



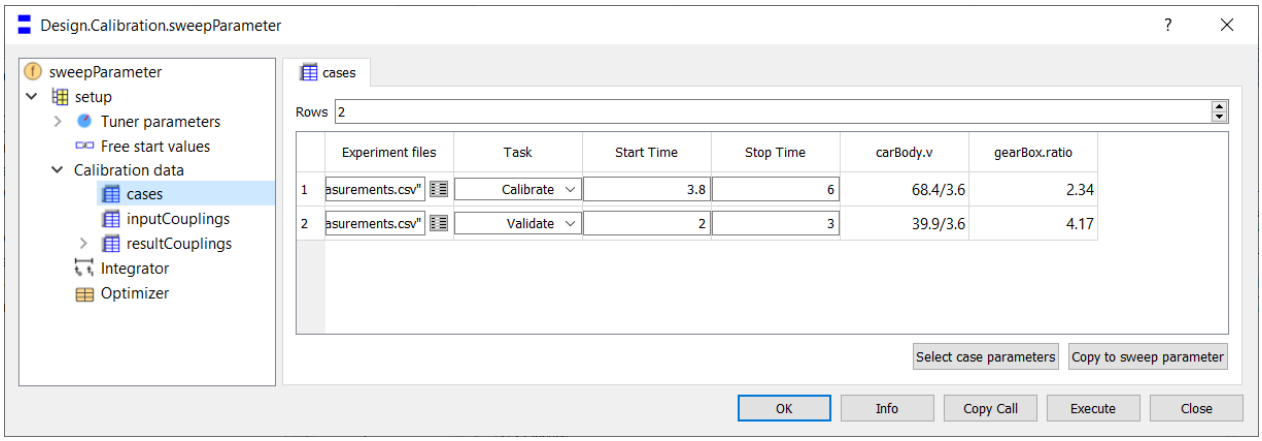
Select gearBox.lossTable[1,2] in the combobox of **sweepVariable** and select **Equidistant grid** for **values**. When doing the last selection, a new menu pops up:

Selecting grid for parameter sweep.

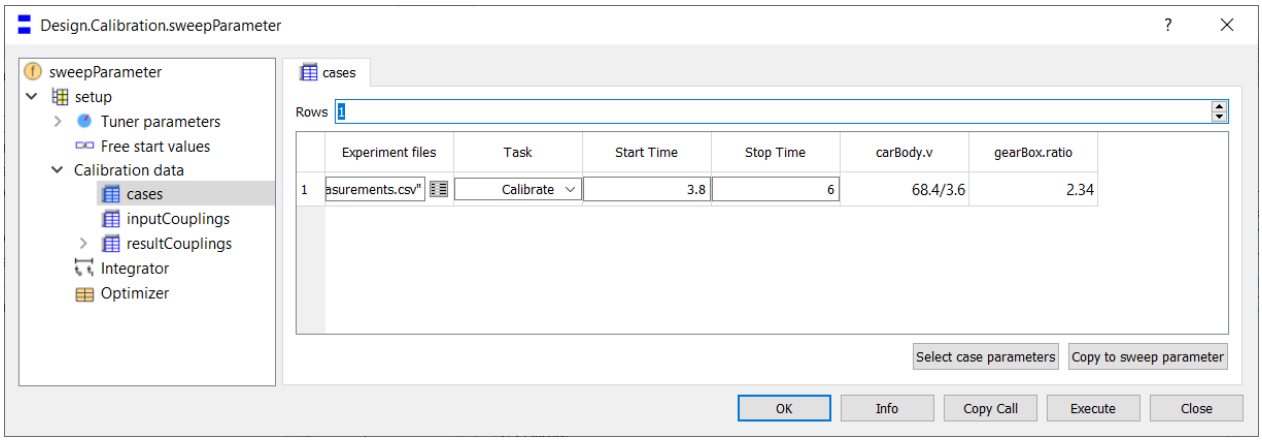


Enter minimum and maximum values and number of points (0.5 for the minimum value, the default 1 for the maximum value, and 6 for the number of points is used for the plot below) and click **OK**.

Click on **cases** in the browser. It will show that we sweeps two cases.

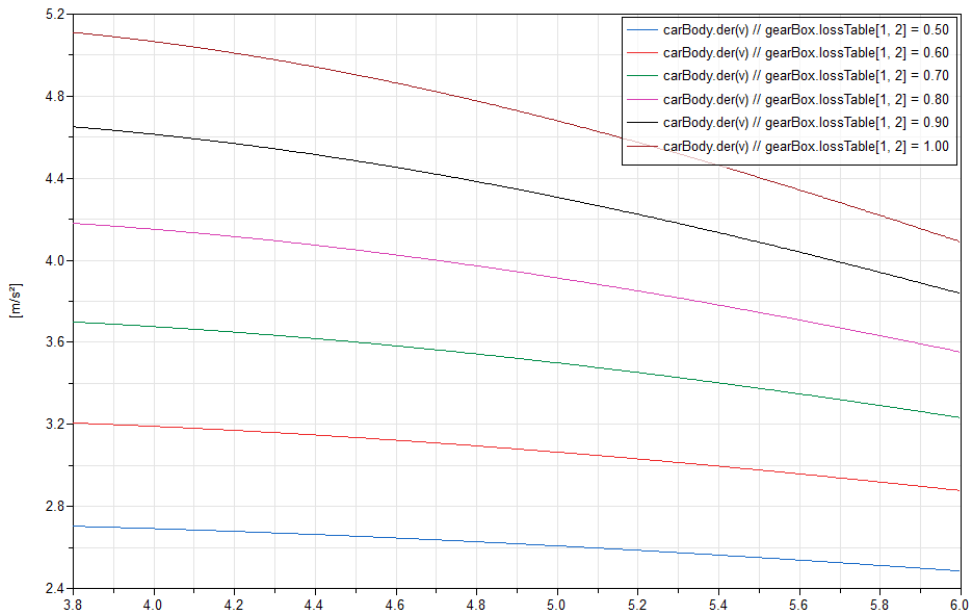


The Validate case is not interesting now, so decrease the number of rows to 1. The result will be:



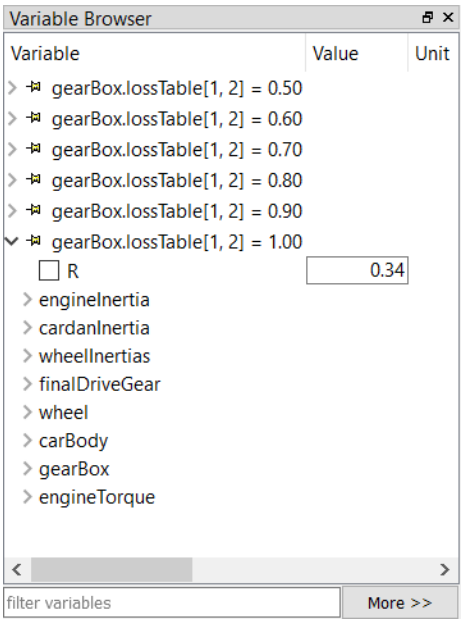
Click **Execute**. The result is plots of the result variables, which in this case is the acceleration. As expected, higher efficiency gives higher acceleration.

Simulated acceleration when sweeping gearbox loss (x-axis is time, y-axis is acceleation).

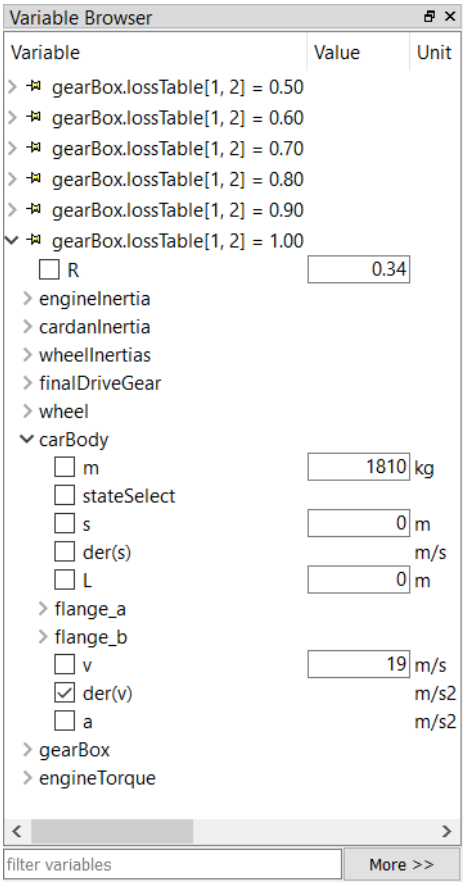


All results of the simulations are available for access in the variable browser.

The results from the parameter is available in the variable browser.

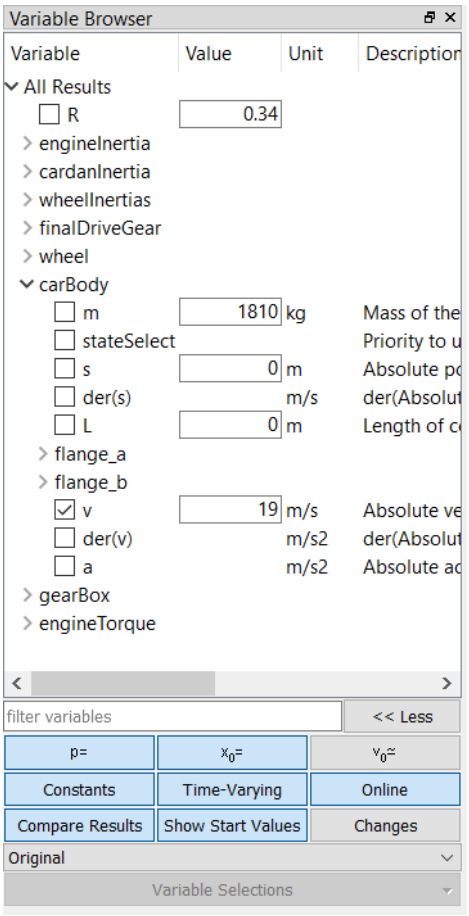


To look what signal is plotted, **carBody** can be expanded by clicking on the **>** before it. The result in the variable browser will be:



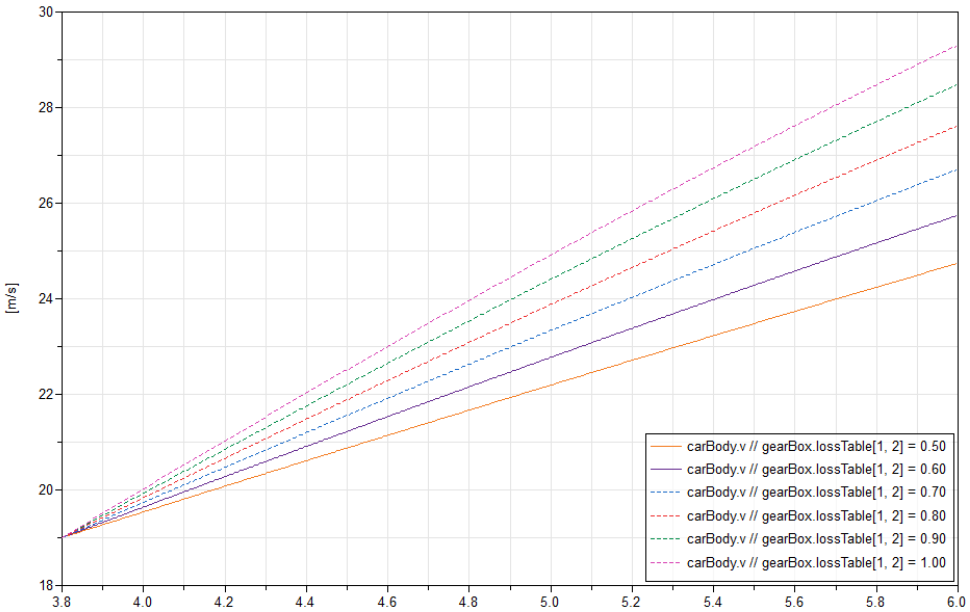
All Dymola's plotting facilities can be used to produce other plots from the sweep. It is e.g. easy to get a similar plot with the velocity of the car. Click on **More** and click on **Compare results**. Then select **carBody.v** – and deselect **carBody.der(v)** to not get too many curves:

Setting plot to compare results.



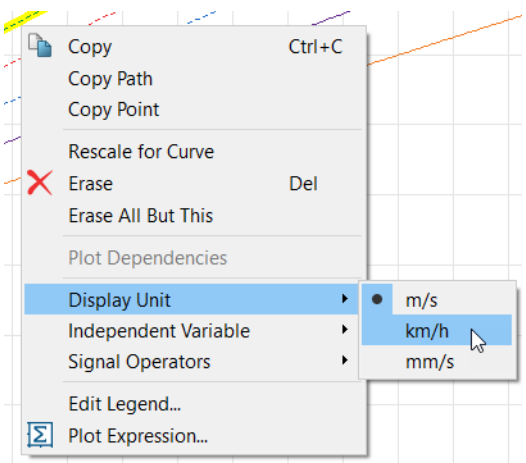
and the plot becomes (after having moved the legend square to the bottom left, see later):

Plotting the velocites from the sweep.



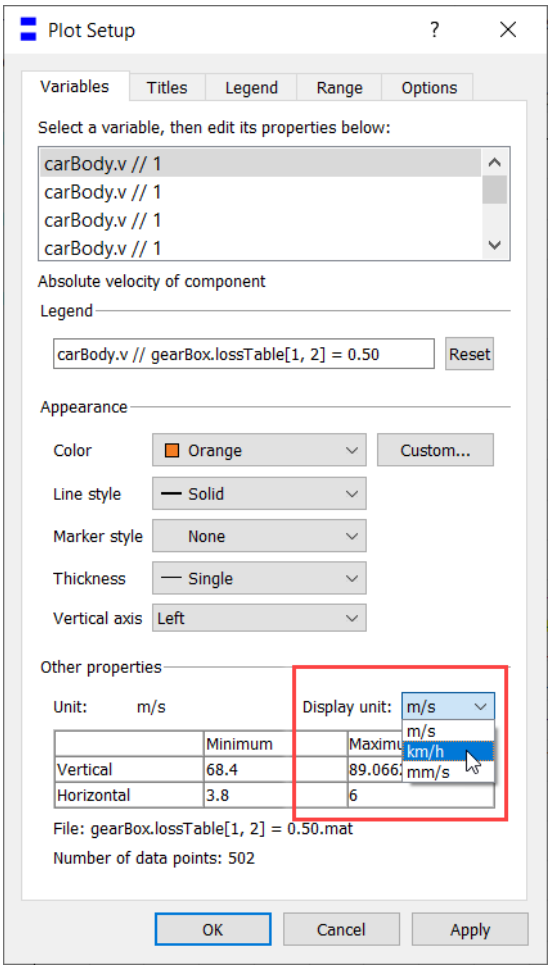
The velocites are in m/s, but it is easy to get them in km/h. The easiest way is to right-click on a curve and use the context menu command **Display Unit** to select **km/h**.

Using context menu to select display unit.



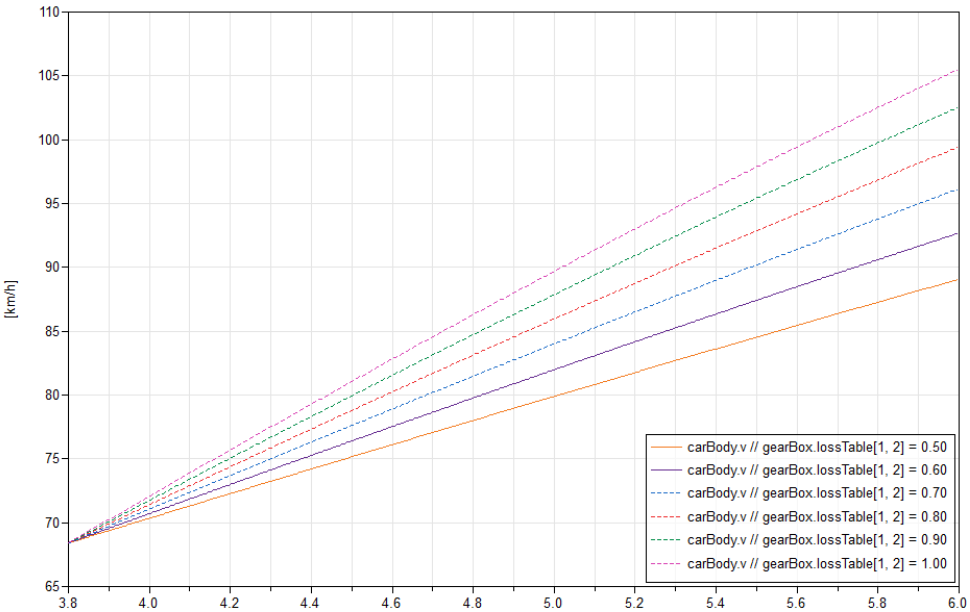
Another way is to use the command **Plot > Setup** and use the **Display unit** drop-down menu in the **Variables** tab:

Changing display result.



Whatever method used, the plot becomes now

Plotting velocities in [km/h]. The x-axis is time.



The plot window has a lot of other possibilities, e.g. zooming, displaying values etc., please see chapter "Simulating a model", section "Model simulation", sub-section "Plot window interaction". (The **Setup** menu above can be used for many things as well, e.g. to change the placement of the curve legend using the **Legend** tab.)

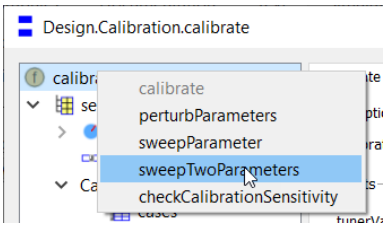
In the beginning of this example we de-selected one of the two cases available in the demo, the validation case. If both cases have been selected, two plots would have been the result, and the corresponding signals should also have been visible in the variable browser.

7.5.2 Sweep two parameters – sweepTwoParameters

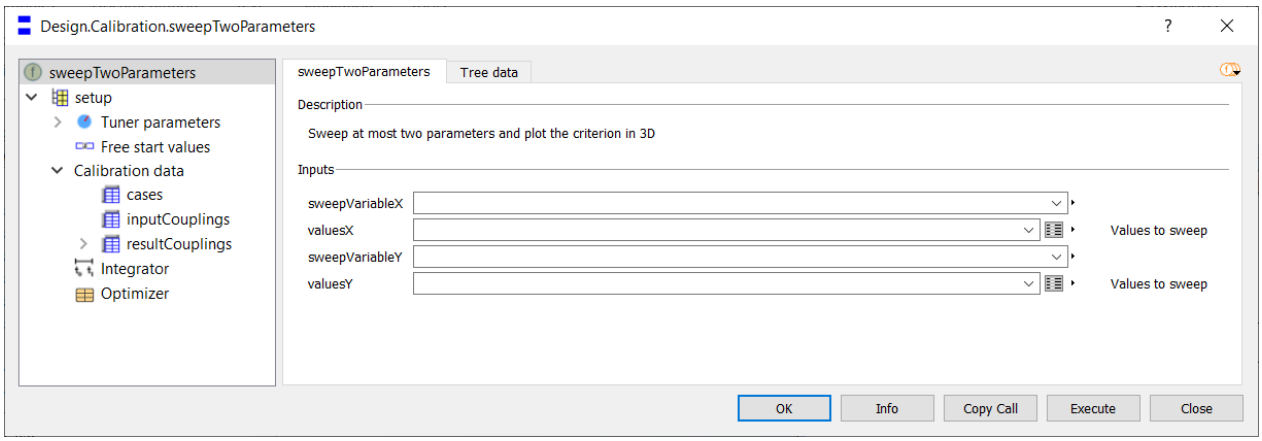
The function sweepTwoParameters sweeps two tuners and produces a 3D plot of the criterion.

As an example open the model (if not already open) `Design.Calibration.Examples.SimpleCar` by using the command **File > Libraries > Design** and then using the package browser to open `Examples.SimpleCar`

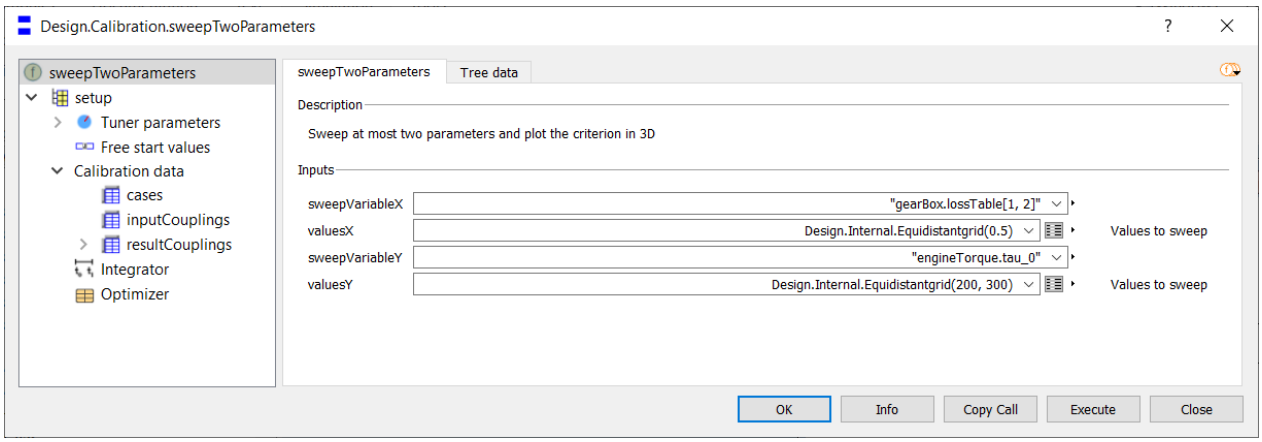
Now select the command **Simulation > Commands > Calibration with validation**. When the sweepParameter window has opened, select **calibrate** (at the top in the tree browser *in that window*), right-click and select **sweepTwoParameters**.



The menu changes

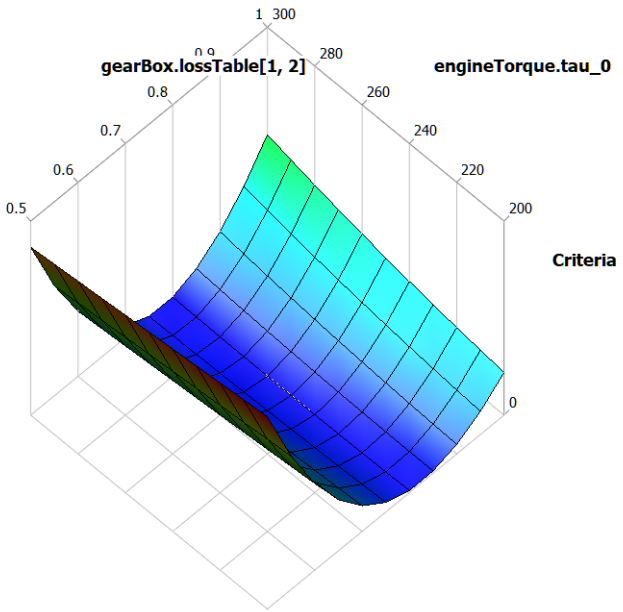


Since we have just two tuners, we select the efficieny as **sweepVariableX** and 11 values in the interval 0.5-1. We select tau_0 as **sweepVariableY** and 11 values in the interval 200-300. The result of this will be (for a more detailed description on how to select start values etc please see previous section if needed).



Click **Execute** and Dymola produces the plot below:

The criterion for different values of the tuners.

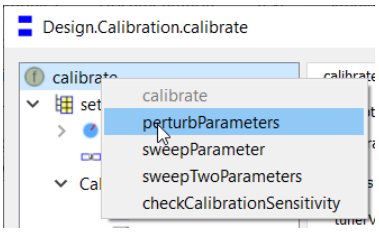


7.5.3 Response to parameter perturbations - perturbParameters

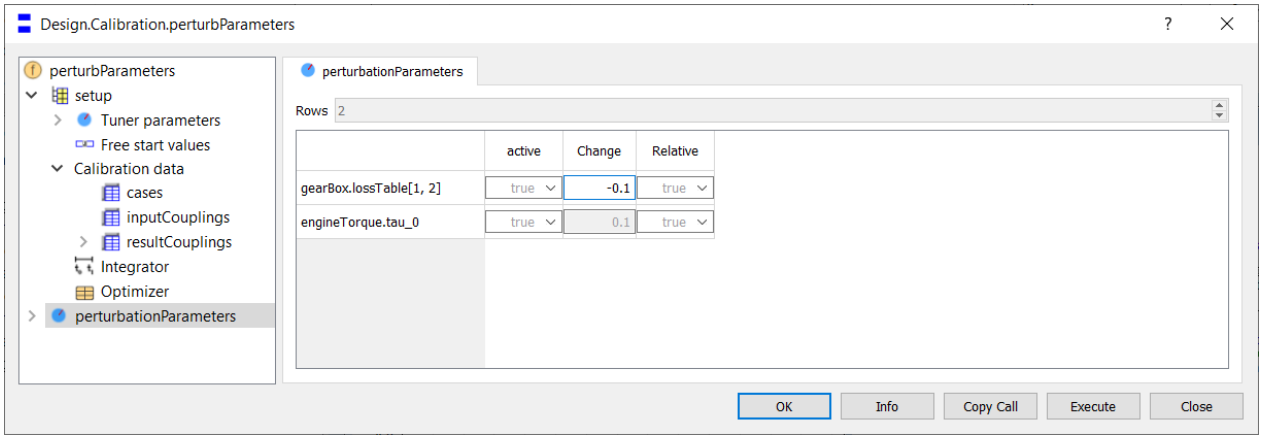
The function perturbParameters plots the responses to perturbations in the tuners.

As an example open the model (if not already open) `Design.Calibration.Examples.SimpleCar` by using the command **File > Libraries > Design** and then using the package browser to open `Examples.SimpleCar`

Now select the command **Simulation > Commands > Calibration with validation**. When the sweepParameter window has opened, select **calibrate** (at the top in the tree browser *in that window*), right-click and select **perturbParameters**.

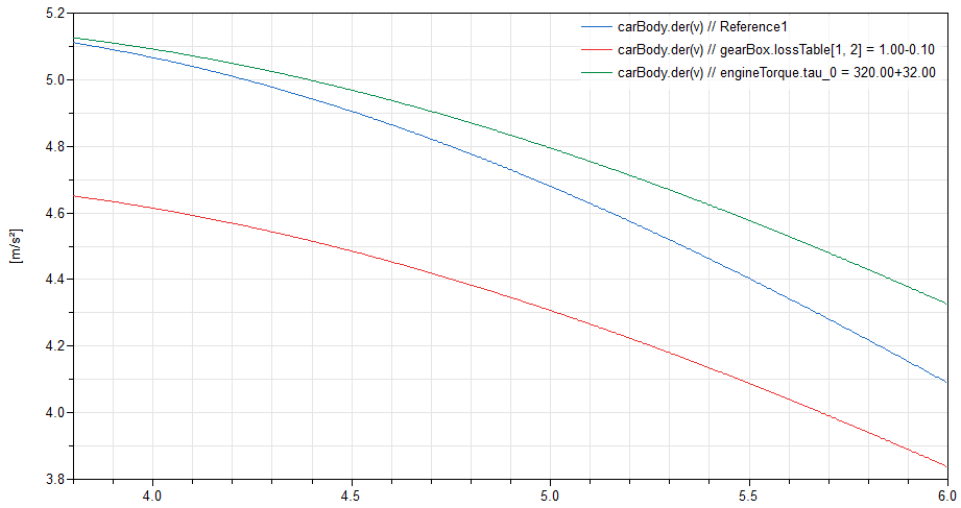


The menu changes. The function exploits the setup of calibration, but needs some additional input, perturbationParameters, which by default are the same as the tuner parameters of setup. When executing the function it perturbs the parameters in turn. The default pertubation is 10%. Note that the efficiency has a nominal value of 1 meaning a default perturbation to 1.1, which is not a physical value. Thus, we change it to –10% to get an efficiency of 0.9. The menu after the change look like:



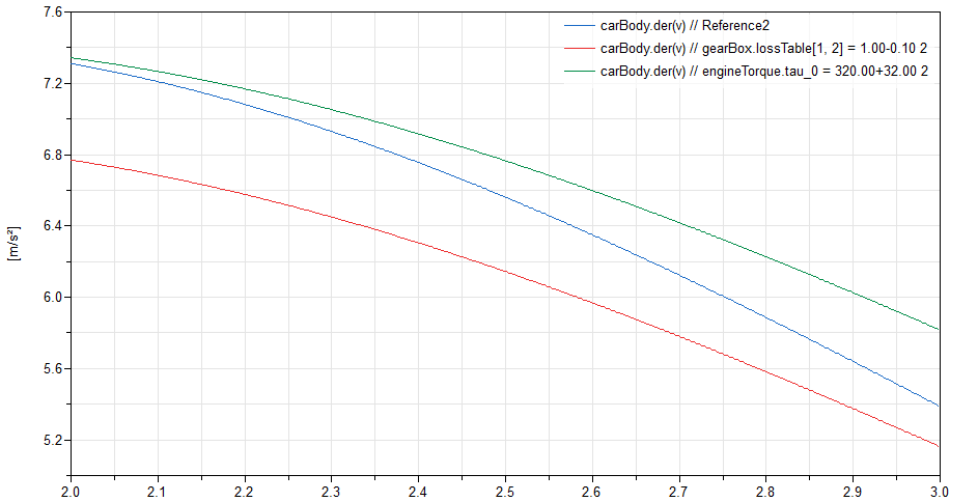
Click on **Execute**. The results are plots of the result variables as shown below (you get two plot windows; this is the first one).

The acceleration when perturbating the tuners.



Both tuners influence the acceleration. The responses for the validation case are also plotted, in a second plot window.

The acceleration for the validation case when perturbing the tuners.



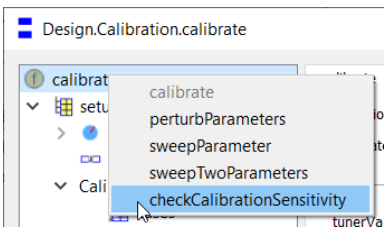
7.5.4 Check if tuners can be calibrated – checkCalibrationSensitivity

When tuning parameters from measurements, a basic question is "Which parameters can be estimated from the measurements available?" Changing a parameter to be estimated must of course influence the output. However, this is not enough. Two or several parameters may influence the result in a similar way such that it is not possible to estimate them individually.

Assume that our nominal model is the correct model and we had used it to produce a "measurement" file. If we make small perturbations of the values of some parameters and use the "measurement" file for calibration, we would like the result of the calibration procedure to be that the perturbed parameters are tuned to their original values. The function checkCalibrationSensitivity checks if this is the case. If not, it lists tuners that do not influence the criterion and linear combinations of parameters where changes of the appearing parameters do not influence the criterion, if the linear expression remains constant.

As an example open the model (if not already open) `Design.Calibration.Examples.SimpleCar` by using the command **File > Libraries > Design** and then using the package browser to open `Examples.SimpleCar`

Now select the command **Simulation > Commands > Calibration with validation**. When the sweepParameter window has opened, select **calibrate** (at the top in the tree browser *in that window*), right-click and select **checkCalibrationSensitivity**.

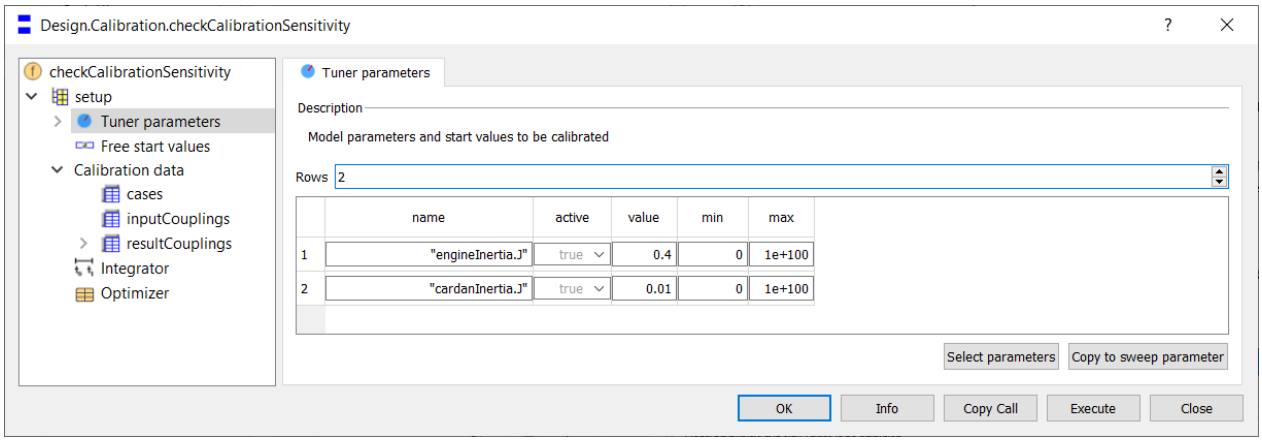


The menu changes. Click **Execute**. Dymola outputs a positive message in the Command window:

The result of checking calibration sensitvity.

```
The calibration criteria are sensitive for small variations
around the nominal values in all tuner parameters and in all
their linear combinations.
```

Let us try some other tuners. Can we tune the engine and cardan inertia? Click on **Tuner parameters** in the browser in the checkCalibrationSensivity window. Select them as tuners by removing the present tuners by setting the **Rows** to 0, then use **Select parameters** to select the new tuners. The result in the menu will be:



Click **Execute**. Dymola outputs the message

```
The calibration criteria are insensitive for small variations
around the nominal values in the following linear parameter
combinations:

-engineInertia.J-0.1826*cardanInertia.J
```

The message says that if we change the two values, but keep

```
-engineInertia.J-0.1826*cardanInertia.J
```

constant, then the criterion does not change. In other words we cannot tune the inertias individually, but we can tune the combination given. The engine and the cardan are rigidly coupled. It means that the inertia for those to bodies sensed from the engine is

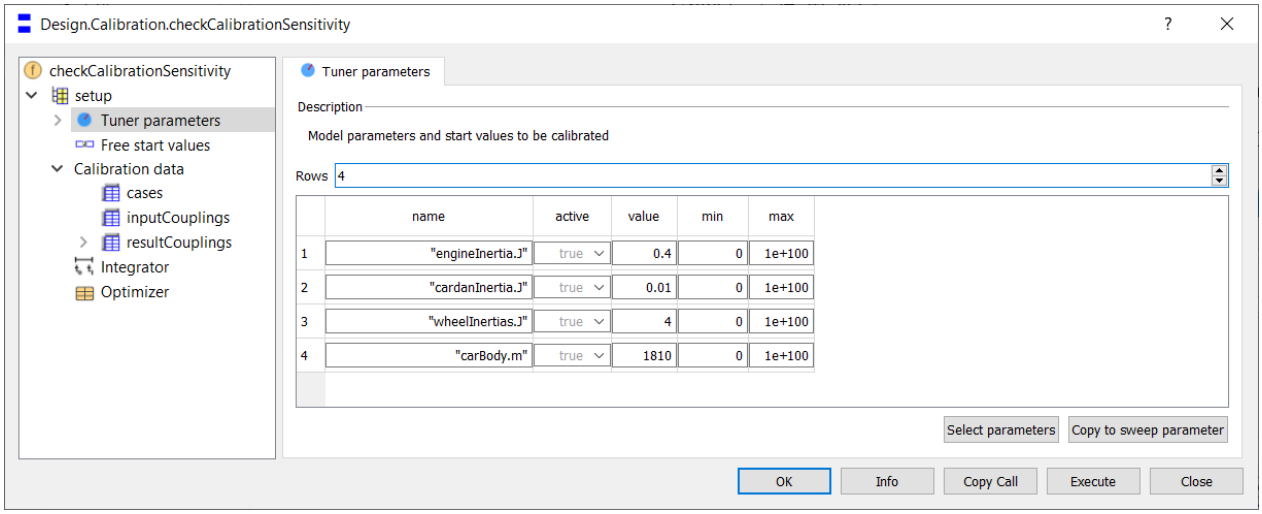
$$J_e + J_c / i^2$$

where J_e is the inertia of the engine and J_c is the inertia of the cardan and is i is the gear ratio. Using $i = 2.34$, we get

$$J_e + J_c / i^2 = J_e + 0.1826 J_c$$

This is consistent with what Dymola told us. In fact the engine, cardan, wheels and the car body are rigidly connected. It means that we can only estimate a total inertia for example

reduced to the engine side or a mass equivalent reduced to car body. Let us specify the inertias and the car mass as tuners.



Clicking **Execute** gives the expected answer

```
The calibration criteria are insensitive for small variations
around the nominal values in the following linear parameter
combinations:

-engineInertia.J-0.0018*carBody.m-0.1826*cardanInertia.J-
0.0153*wheelInertias.J
```

If we multiply by –1, this is the total inertia reduced to the engine side.

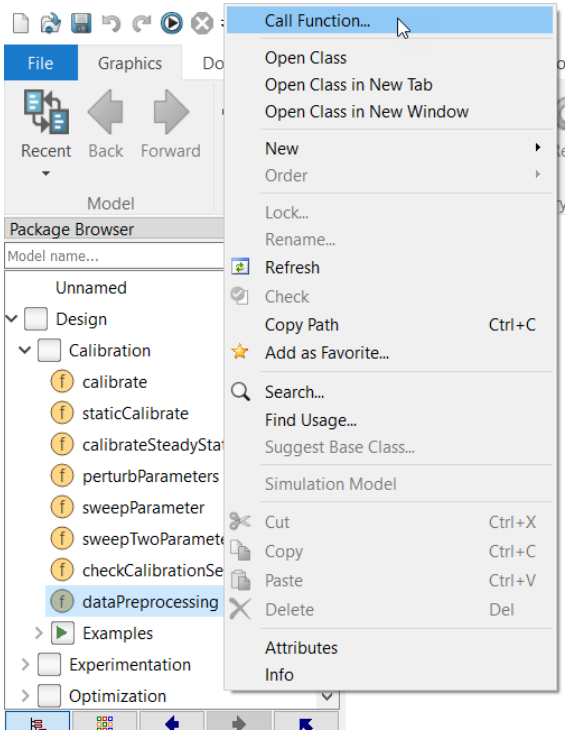
7.6 Data Preprocessing

The quality of the calibration process is directly related to the quality of the measured data used as input to the calibration tool. Any factor that perturbs the data will cause directly distortion of the final result of the calibration tool. It is important then to preprocess the data, that is, to adjust the data eliminating noise, zones where the model is not valid and erroneous or not representative measurements.

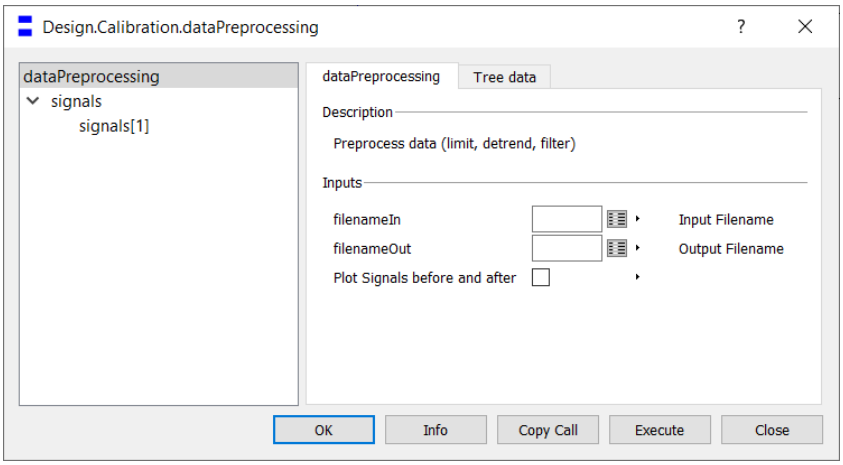
The Design.Calibration package incorporates the function `dataPreprocessing` with this objective: preprocess data for calibration.

7.6.1 Setting up for preprocessing

Select `dataPreprocessing` in the package browser under `Design.Calibration`. Right-click and select the command **Call function...**



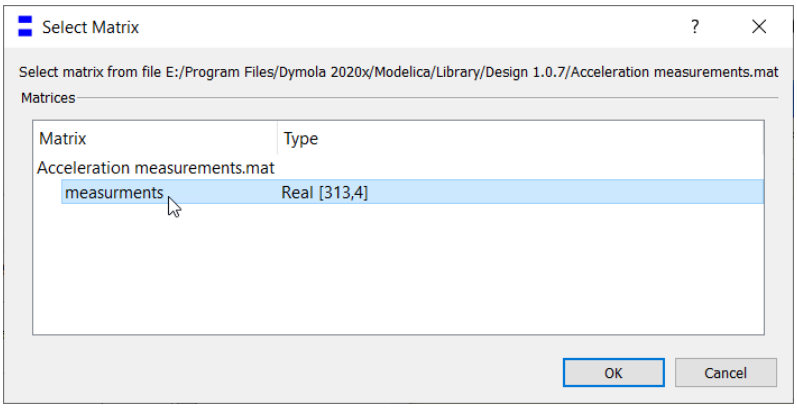
The dialog of dataPreprocessing pops. We select now the file we want to process and the file that will contain the output.



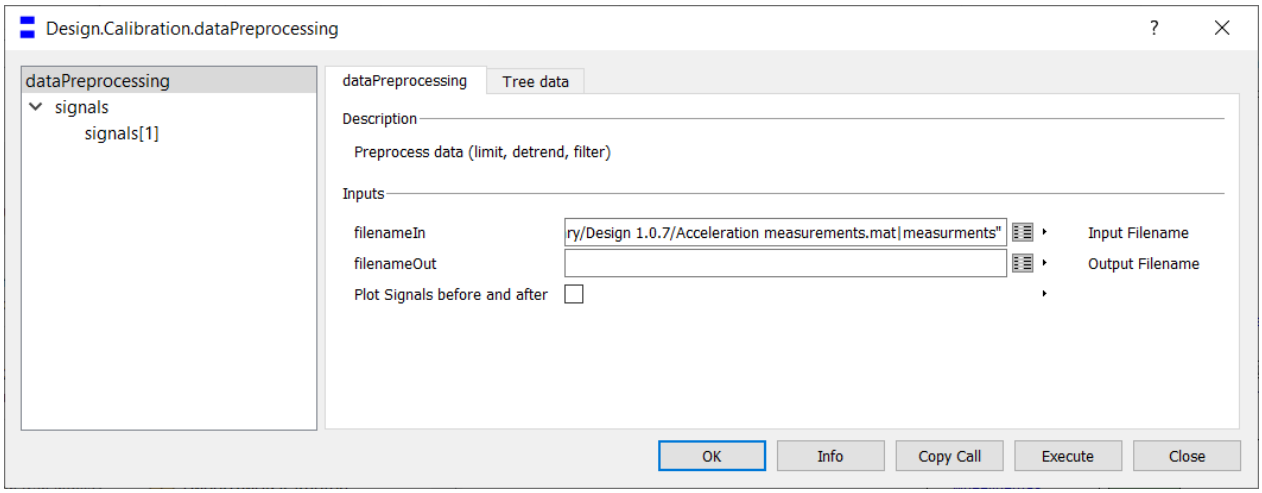
To introduce an experiment file, click on the **Edit** icon after the in the **filenameIn** input field. A file browser pops up. Use it to select the file

`…/Design 1.1.3/Acceleration measurements.mat.`

(As an example for Dymola 2023, the full path may be `E:/Program Files/Dymola 2023/Modelica/Library/Design 1.1.3/Acceleration measurements.mat`.) A menu for selection of appropriate matrix will pop up. In this case there is only one selection possible:



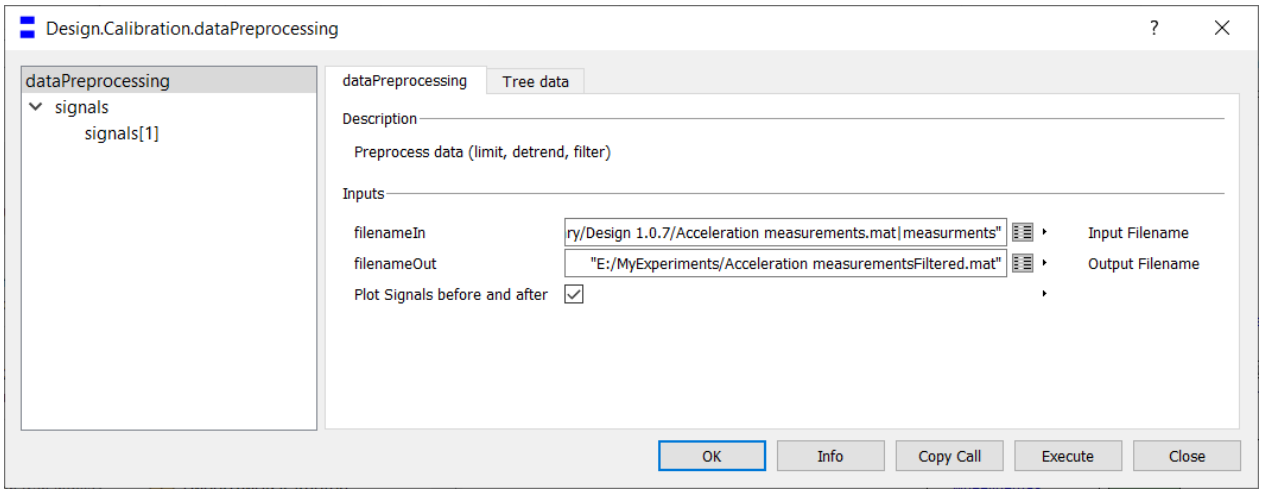
After clicking **OK**, the window will look like:



The same procedure has to be repeated to select an output file. In this case, the file does not exist. To not try to write into a folder that may be write-protected, create a suitable path on your machine – note that the file does not have to be present, it will be created. As an example, we choose as name for this example

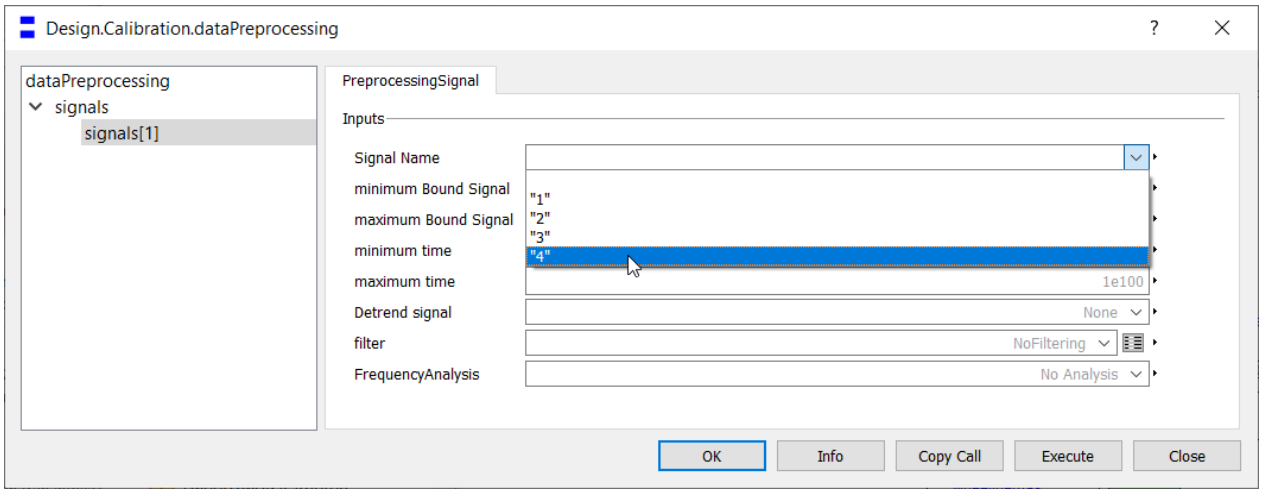
`E:/MyExperiments/Acceleration measurementsFiltered.mat`.

and check the field **Plot Signals before and after** to obtain a plot of the original signal and the result of the preprocessing.

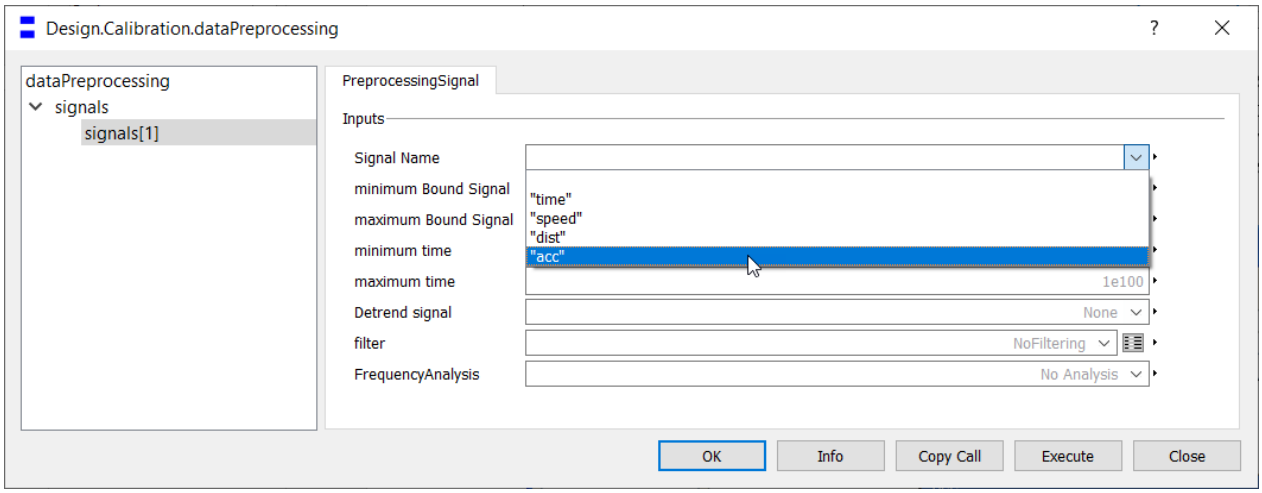


It is possible to overwrite the original file; there is no restriction for that. This means that the original data is gone, and no possibility of recovery is possible. A good practice is always to change the name slightly, since we might want to adjust later on the preprocessing parameters.

We select now a signal from the file. Since the input filename is a .mat file, we don't have access to the name of the variables, but we know the position of the acceleration signal, that is, number four in the matrix. Choose **4** from the combo box in **Signal Name**.



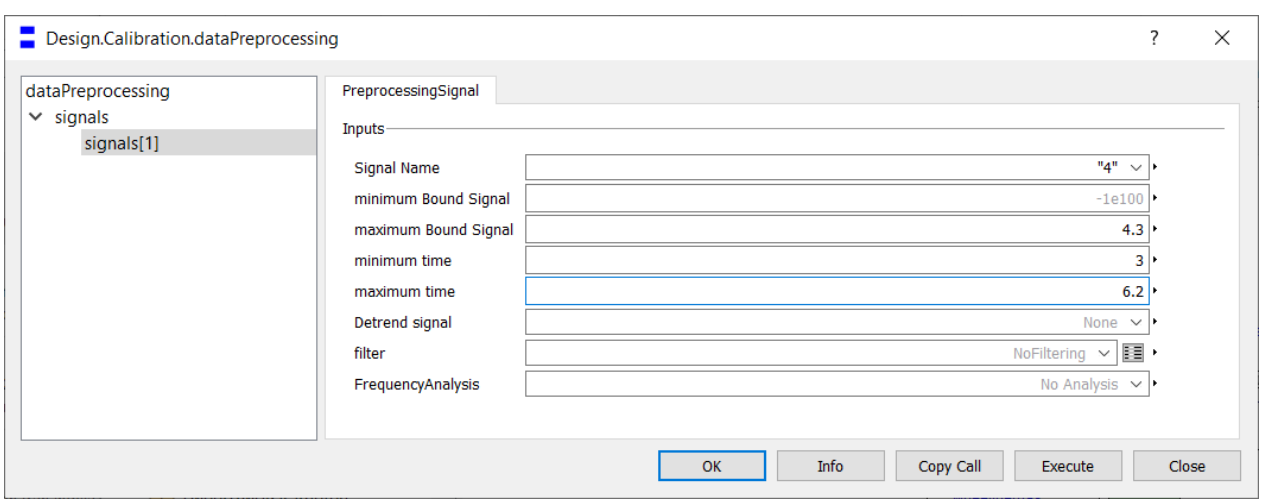
The dataPreprocessing tool assumes for .mat files that time is in the first column always. This is a cornerstone of the function, since all functionalities are relying on time. This is very important. If we instead choose to process a .csv file, we get the names in the combo box. We can simply then select "acc". And dataPreprocessing will seek the keyword "time" and "Time".



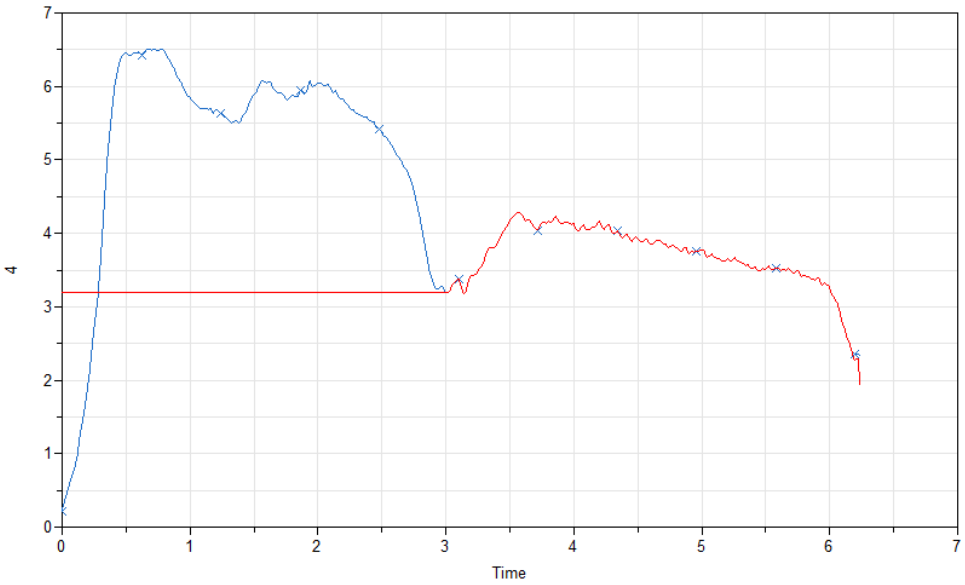
We are ready now to run the preprocessor function. We start with limiting and detrending functions.

7.6.2 Limiting and detrending signals

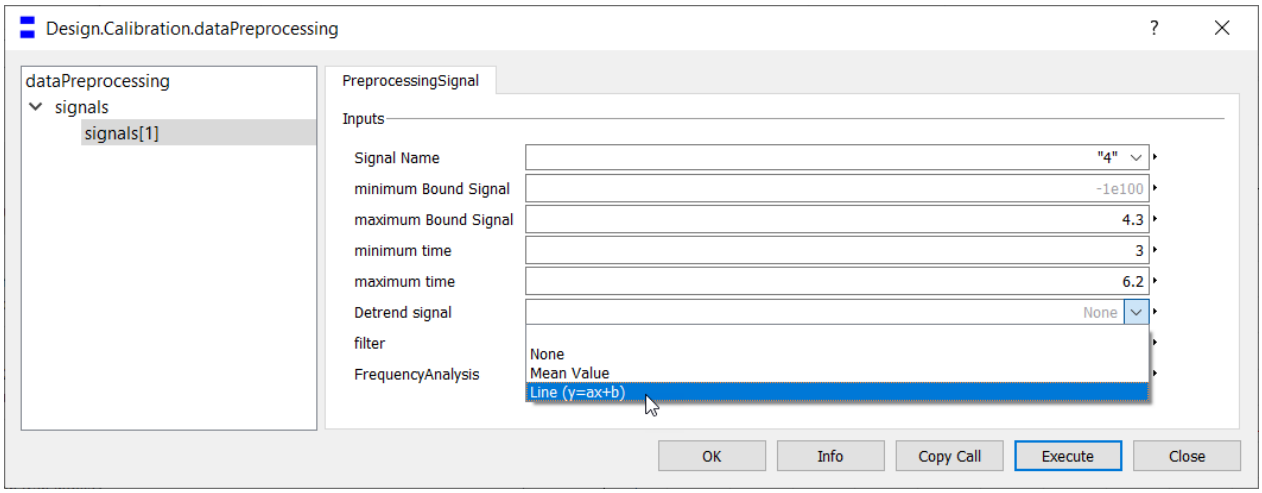
Limiting and detrending the signals is also very important. To limit a signal in time and amplitude it is enough to write the desired values in the fields **minimum Bound Signal** and **maximum Bound of Signal** for the amplitude and **minimum time** and **maximum time** for time axis. The data outside of these limits is taken away and interpolated or extrapolated linearly.



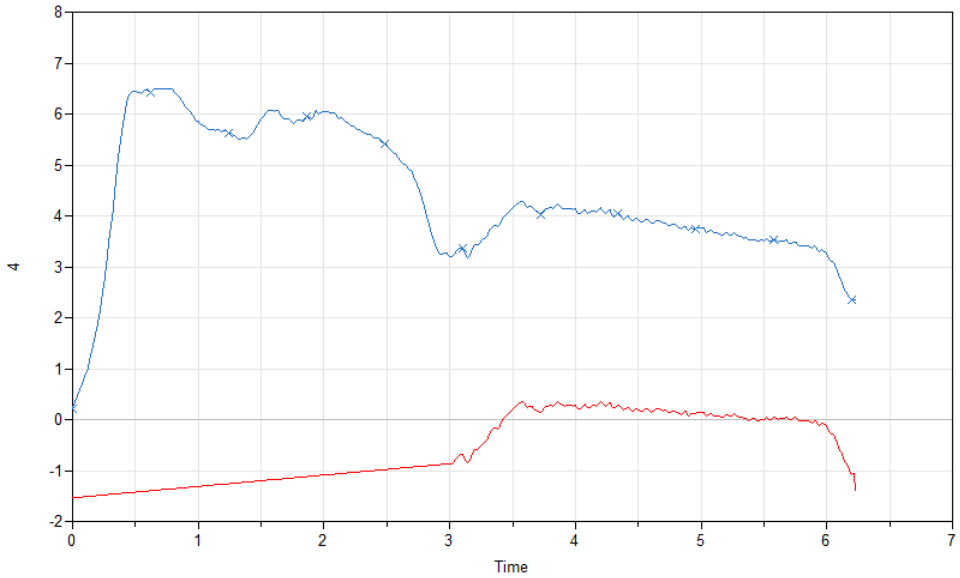
Just to demonstrate this feature, take as limits the interval [-1e100,4.3] for the amplitude and [3,6.2] for time. Press **Execute** and the result is presented.



All values outside of the range have been substituted by linear interpolation. Now, we choose **Line(y=a*x+b)** in the combo box **Detrend signal**. This will fit in least squares sense a straight line and subtract it form the data.



Press **Execute** and the result follows.



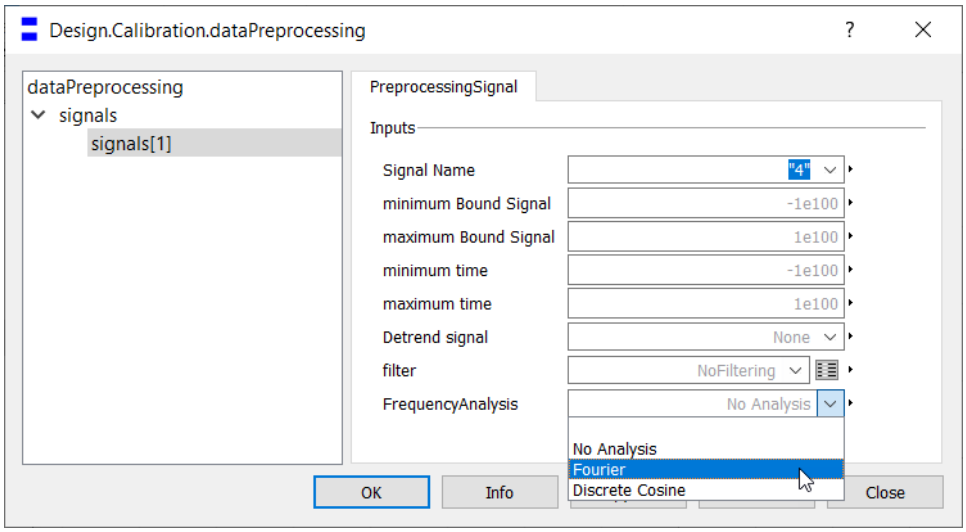
We observe the resulting curve. The other possibility for detrending is **Mean Value**, which subtracts the mean value of the function. Reset the values of the limits and set detrending to **None**. We are now into frequency analysis and filtering of signals.

7.6.3 Analyzing Signals: is there any noise?

Let us analyze the function in the frequency domain. The point now is to filter out noise. Such a noisy perturbation is normally easy to get in the measured data and complicates later on the calibration process unnecessary.

To be able to plot the original curve against the manipulated without applying any limiting and detrending, let us close the dialog and call the dataPreprocessing function again. Redo the file selection in the dataPreprocessing part of the dialog, and select, in the signals[1] tab the signal name “4”.

Now, select **Fourier** from the combo box of **FrequencyAnalysis**.

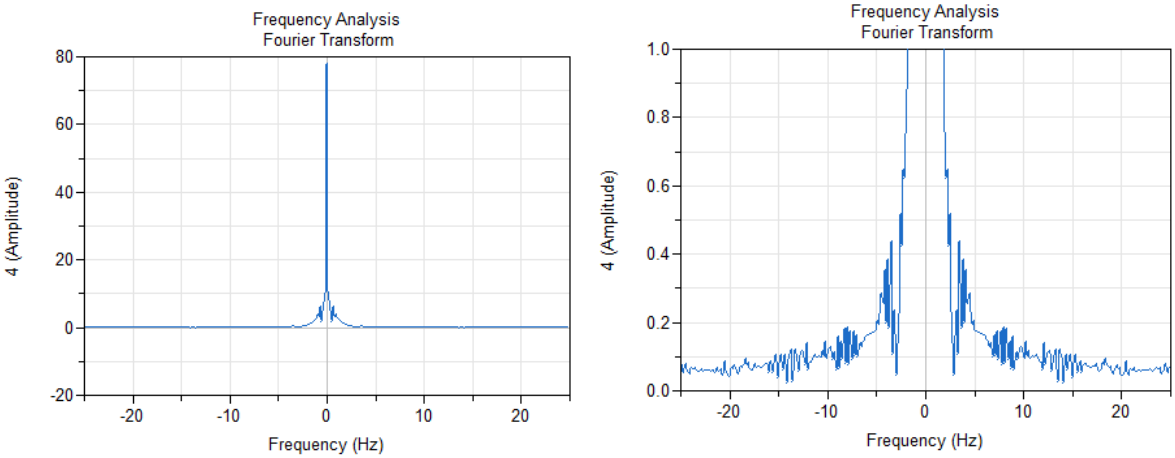


We are about to perform the discrete Fourier transform (DFT) of the acceleration signal. The DFT is defined as follows. Assume we have N samples of a function $x_n = x(t_n)$ at $t_n = nT_s$, where T_s is the sampling time. The DFT is a set of complex numbers c_k such that

$$x_n = \sum_{k=1}^{N} c_k \exp\left(\mathrm{i}\,\omega_k n T_s\right)$$

for all sampling x_n points and frequencies $\omega_k = \dfrac{2\pi k}{NT_s}$, and i the imaginary unit. The coefficients c_k can be calculated explicitly by matrix-vector multiplication or by more effective algorithms in case of large amounts of data.

The frequencies are discrete equidistant points distributed in the interval $\left[0, \dfrac{2\pi}{T_s}\right]$. Since the complex exponential function $\exp\left(\mathrm{i}\,\omega_k n T_s\right) = \exp\left(\dfrac{\mathrm{i}\,2\pi k n}{N}\right)$ is periodic, we choose a representation in the interval $\left[-\dfrac{\pi}{T_s}, \dfrac{\pi}{T_s}\right]$, to have the highest frequencies farther at the boundary, instead of in the middle of the graph. Now, we press **Execute** and obtain the graph at the left side. The right side graph is a zooming.



Since the coefficients are complex numbers, we present their modulus. In the log of the command window we observe also the following report

```
Processing signal 4

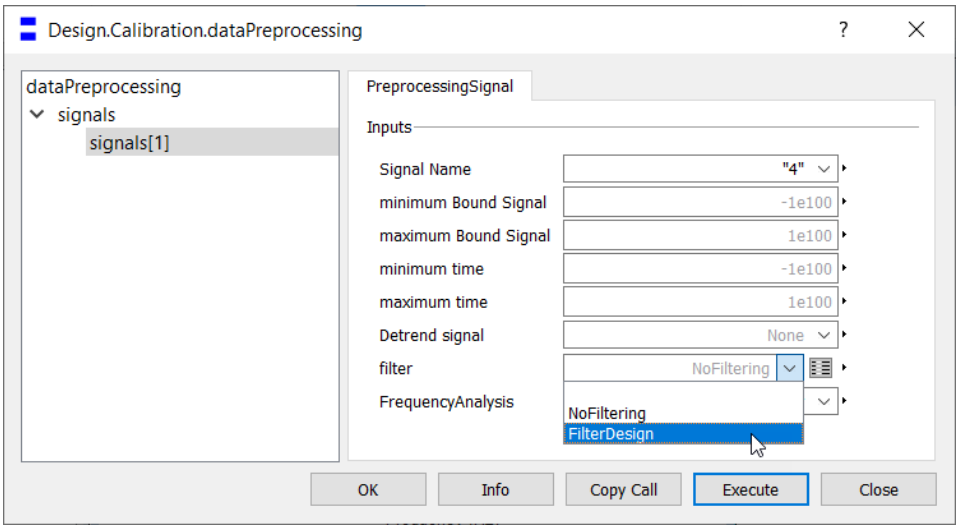
Signal 4 has the 99.5341% of its power under 2.21519 Hz
```

This is an important piece of information. The tool detected that the energy of the signal is almost condensed in the interval $[-2.2159, 2.2159]$. In the graphs before (right) we observe the behavior of the coefficients, and it is less smooth and more erratic outside of the interval reported by the tool.

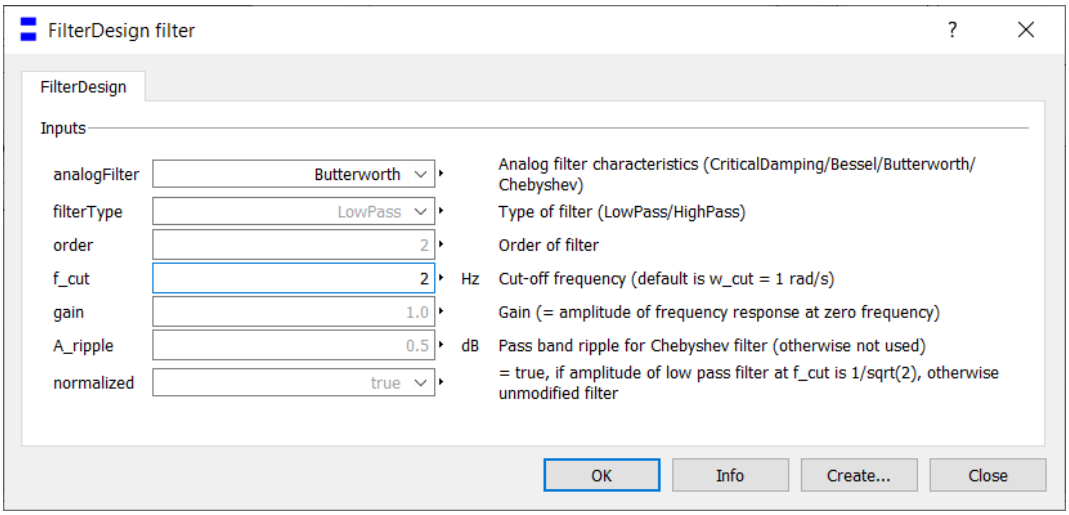
We can therefore suspect that additive noise is present in this range of frequencies. We can now design a filter and get rid of these noisy oscillations.

7.6.4 Filtering signals

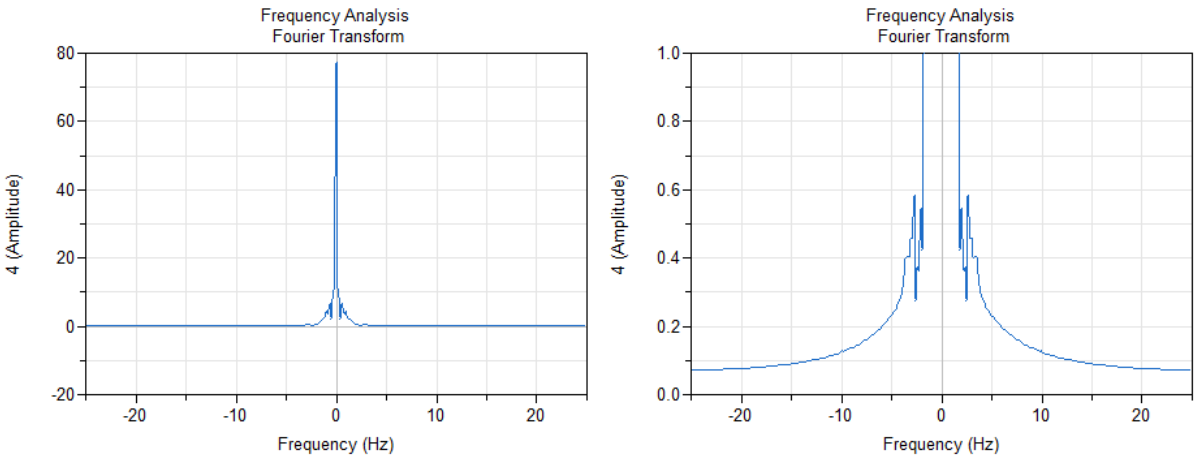
Now, click on **filter** combo box and choose **FilterDesign**.



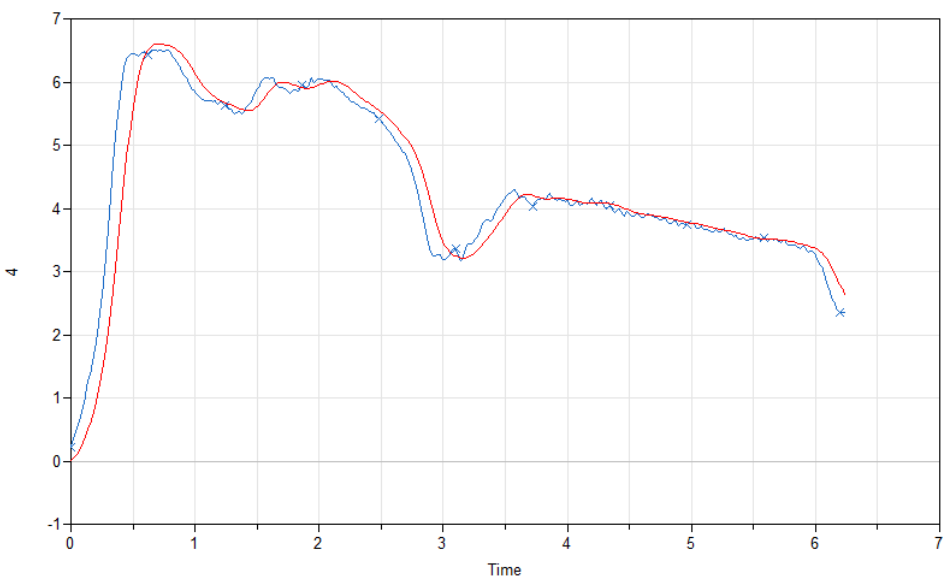
The GUI for filter Design from the LinearSystems library pops up now.



We choose a Butterworth filter in **analogFilter** and as cut frequency we choose 2 Hz. It is enough to choose 2 Hz and not exactly 2.2159 since the filter is not ideal and we will smooth out the spectrum of the signal around those frequencies too. The type of the filter has to be low pass since the signal is clustered around zero frequency. Press **OK**. Then **Execute** in the main GUI. The resultant spectra are presented.



We observe how high oscillatory modes are smoothed out. This means that the signal in time is also smoother. The result is presented in the next picture.



The filtered signal (red) has less noise than the original one (blue). This makes the calibration process easier. The filters are constructed using the Linear Systems library (opening “Types” and then “AnalogFilter”). These are discretised versions of continuous systems, with a discretisation in such a way that the ramp response is exact. The possible filters are four: Critically damped, Bessel, Butterworth and Chebyshev.

7.7 Static calibration

Two cases of static calibration exist.

A typical purpose of a static calibration is to tune static characteristics of components such pipes, valve, throttles, pumps, nonlinear resistors, frictions etc. Such models are completely static (always in steady-state). The function staticCalibration should be used in this case.

The other purpose of static calibration is to handle models in steady-state, either by giving steady-state initial conditions or by specifying a suitable stop time (where the user is assumed to guarantee that the solution has reached steady-state). The function calibrateSteadyState should be used in this case.

Dymola's GUI supports setting up such calibrations without building a corresponding test rig model. The GUI allows simple redefinition of a variable to be an input. The measured data for such an input and of course also for an original input can then be specified to have a common value for all cases or to have a case dependent value read from a file.

Assume that we want to find a static relation from the variable v to w of the model and that we have measurements for v and w and all inputs of the model. First we need to decide a parameterized shape or in other words we need to come up with a function w = f(p, v) where p is a parameter vector. In many cases we can use polynomials. In general, it is nontrivial to come up with a good function that fits the data well. However, in this case it is possible to use the model and the measured data to back calculate w for each case. Dymola supports the setup of such a calculation is in a straightforward way very similar to the setup of the transient calibration. Having w makes the relation much more explicit and easier to visualize and inspect. Just by plotting w against v, we can get a good estimate of the chances to get a good result. If the plot shows that the points seem to be lying on a line, the chance is much better than when the plot looks like a random scatter. Such a plot may also give us good insight in what kind of functional relation we should use. Classic pen and paper approaches as plotting in lin-log or log-log diagrams can be used to find out if exponential or potential relations could be useful.

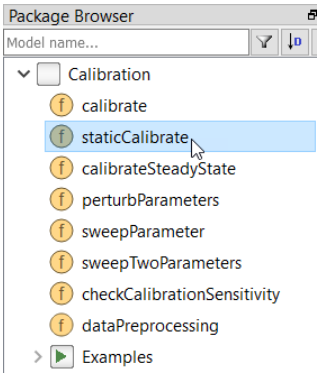
This document uses calibration of a resistor model to illustrate how a static calibration task is setup. More advanced uses are described in H.Olsson, J. Eborn, S.E. Mattsson and H.Elmqvist: Calibration of Static of Static Models using Dymola. Proceedings of the 5th International Modelica Conference, Vienna, Austria, 2006, vol 2, pp. 615-620 (available for download from www.Modelica.org).

7.7.1 The staticCalibrate function

The staticCalibrate function should be used for completely state-less (static) models. For steady-state models, please see the function calibrateSteadyState.

To load Design.Calibration, select **File > Libraries** and click **Design**.

The staticCalibrate function.



Example

(The first example below corresponds to the command **Simulation > Commands > Resistor example 1** in the package Design.Calibration.Examples.Resistor. However, you have to select **Calibration data** in the browser in the pane to the left in the window that pops and browse yourself for the Resistor measurements.csv file; see just below for the path.)

Assume that we have measurements of the voltage across and the current through a resistor. The file `Program Files\Dymola 2023\Modelica\Library\Design 1.1.3\Resistor measurements.csv` contains the measurement data.

The first row of the file includes the column headings and then the data follow.

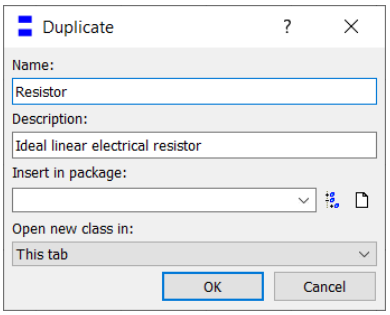
Data for a session.

	A	B
1	Voltage[V]	Current[A]
2	0	0
3	1.4	0.00031
4	3.2	0.00071
5	6.5	0.0014

We now want to estimate the resistance. Thus we need a resistor model. There is one in the Modelica Standard Library: Modelica.Electrical.Analog.Basic.Resistor

Since it is a model from the Modelica Standard Library, it is a read-only model. To allow us to store the calibration setup in connection with the model, we make a copy of it. It can be done in the following way: Select the resistor model of the Modelica Standard Library in the package browser. Right-click and select **New > Duplicate Class...**. Let us call the model Resistor as proposed by the menu. Click **OK**.

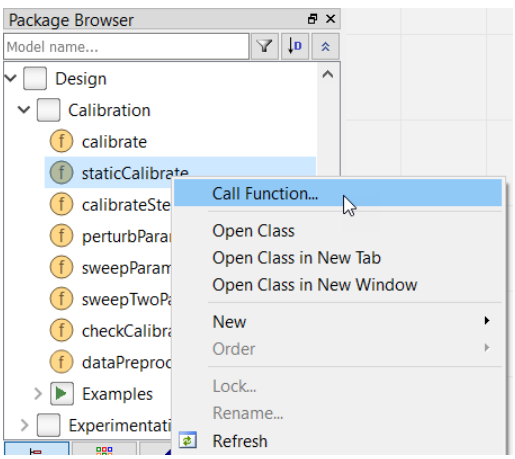
Making a resistor model for calibration.



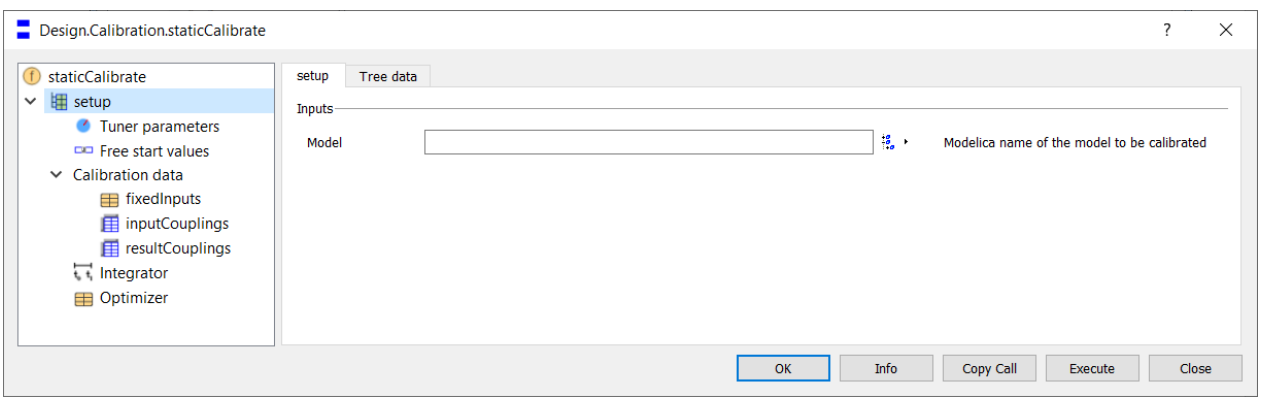
We now have a resistor model. Save the new model.

To set up the calibration, select staticCalibrate in the package browser. Right-click and select **Call Function...**.

Start the setup.

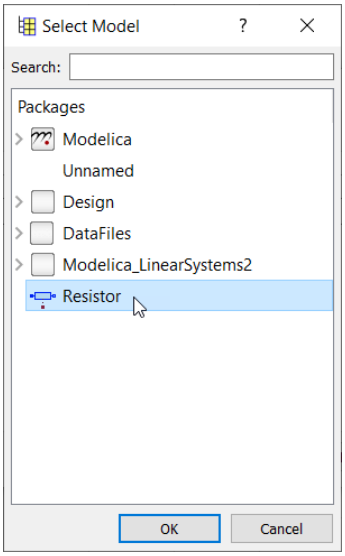


A menu pops up. In the tree in the menu, click **setup**. The following is the result:



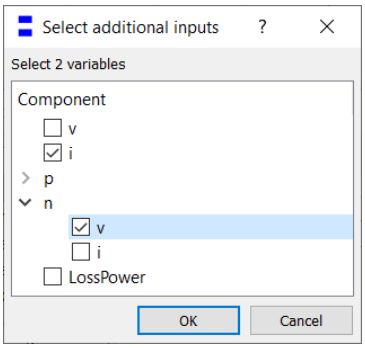
To specify the model to be calibrated, click on the little triangle to the right of the **Model** input field, and then on the **Edit** icon. A package browser pops up. Use it to select the model.

Selecting the model to calibrate.



Click **OK**. The model is now translated in order to gather information to support the further setup of the calibration task. Dymola finds that the resistor model is incomplete (the model has two more variables than equations) and pops a variable browser.

Add inputs to the model.



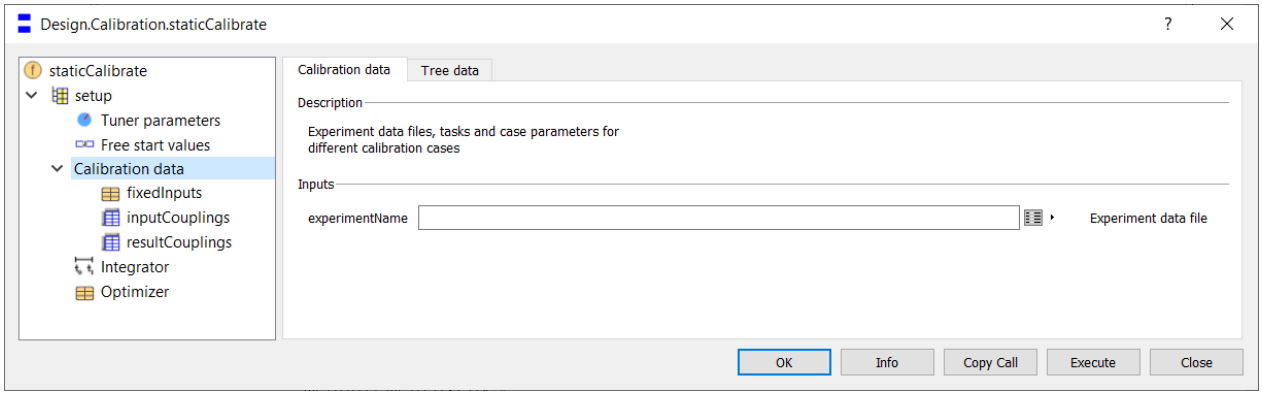
The resistor model cannot be simulated as it is. We need to provide sources driving the component. In real life we need a test bench or a rig. We could have built such a rig model in Dymola using the GUI for model building. We then would have to define a new model, drag in the resistor component and source components and connect them. The availability of appropriate source elements with compatible connectors may be a problem. In this case we could have constructed a test-circuit with a current source driving the resistor being grounded at the n-pin and measuring the resulting voltage across the resistor. The static calibration feature of Dymola provides a powerful solution to this problem. The translate procedure determines which variables that are not uniquely defined and displays them in a browser and

asks us to turn 2 variables of them to inputs in order to make the problem well-posed. In this simple way it is specified which variables that are to be driven from external source.

In this case we tick the current, i, through the resistor, and the voltage n.v to be given.

When clicking **OK**, the model is translated assuming the specified variables to be inputs in addition to other possible inputs of the model component. This rigging is found to be OK. (There will be a warning that a parameter does not have a value, but this is not important here.)

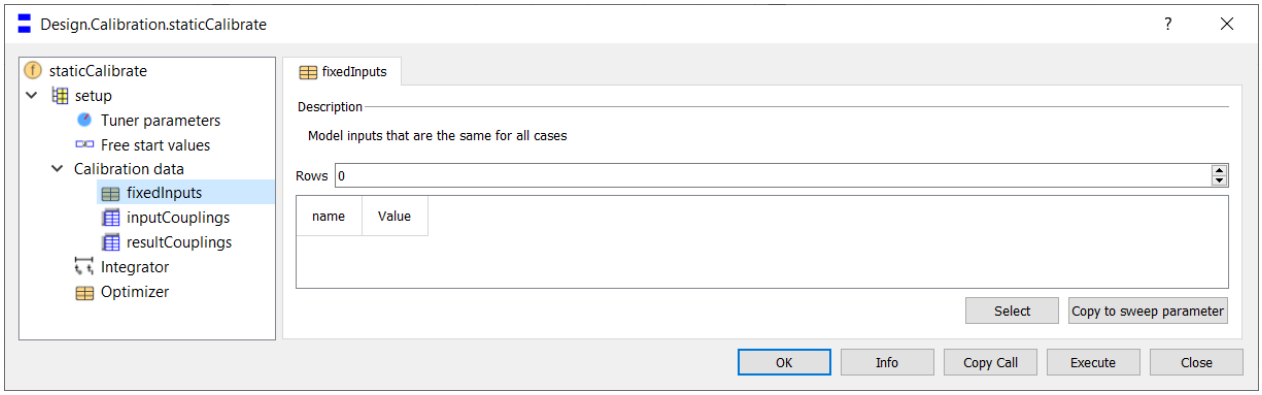
The next task is to specify how the measurements are stored. Click on **Calibration data** in the tree browser to the left.



For static calibration it is assumed that all measurements are stored in one file. Click on the **Edit** icon of **experimentName**. A file browser pops up. Use it to select the file `Program Files\Dymola 2023\Modelica\Library\Design 1.1.3\Resistor measurements.csv`.

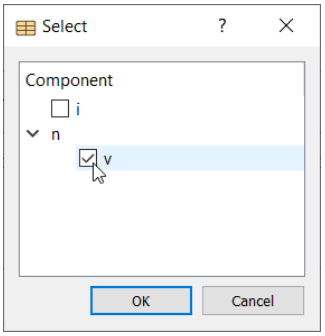
Fixed Inputs

We now need to specify those inputs that have the same value for all cases. Click on **fixedInputs** in the tree browser.

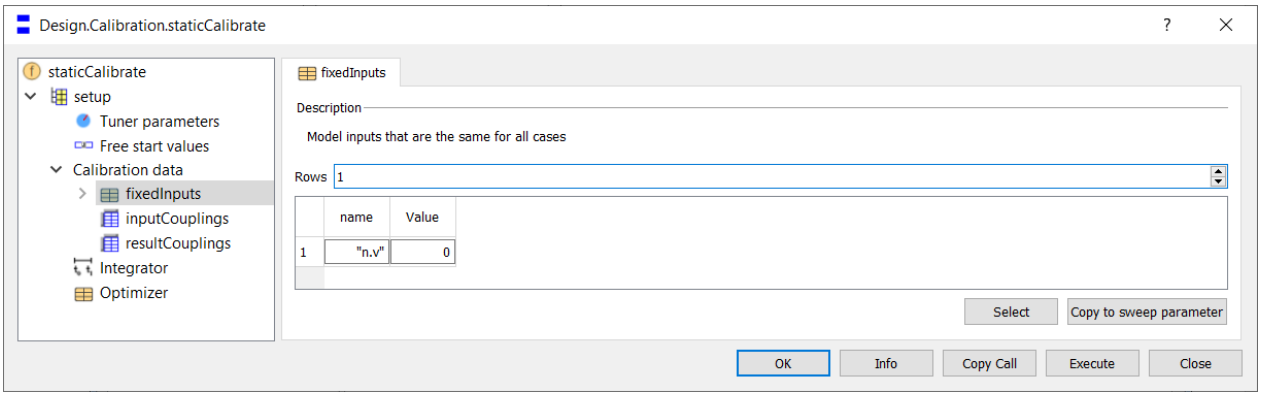


For the resistor calibration we have the grounding of the n-pin to specify. Click on **Select** and a browser including all inputs appears.

Specify grounding.

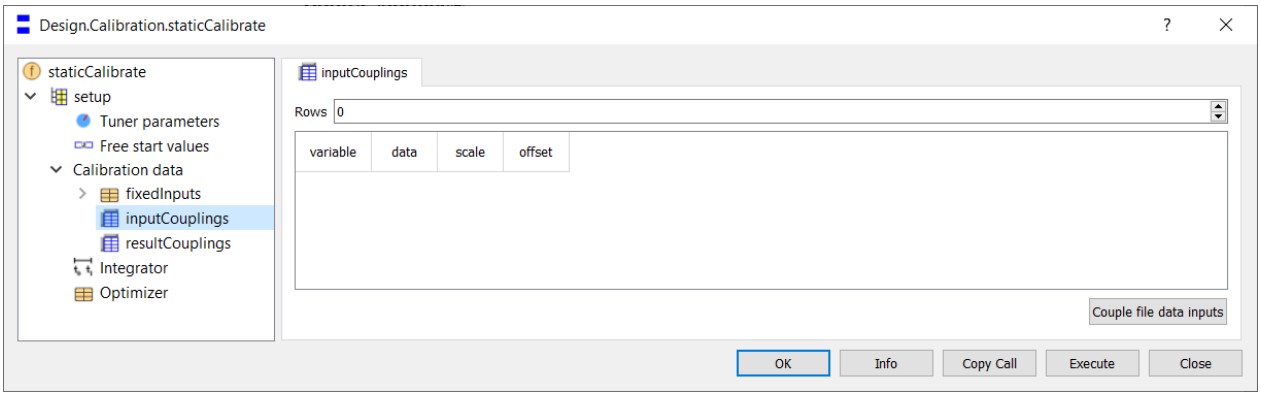


Tick **n.v**. Click **OK**. (Since the value should be 0, no input of value is needed here.)



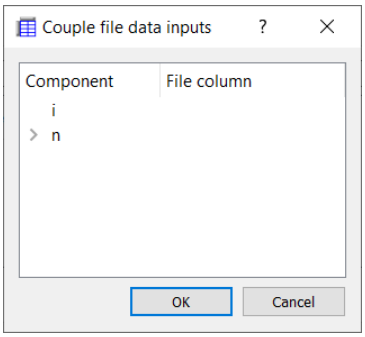
Input Couplings

To couple case dependent sources to the inputs, click on **inputCouplings** in the tree browser.



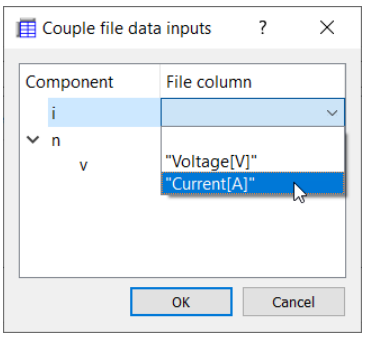
Click **Couple file data inputs** and a browser including all inputs is popped.

Menu to couple inputs.

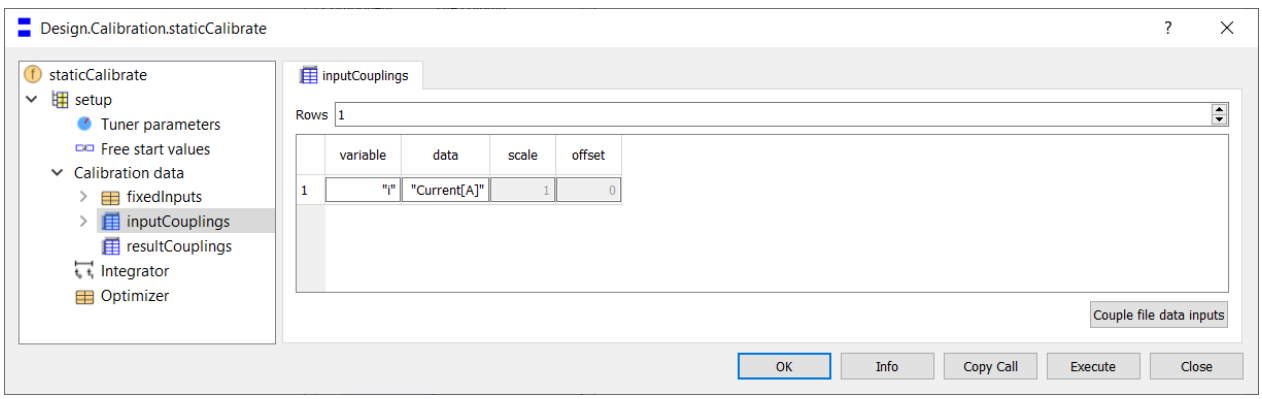


The right column allows us to specify where to find the source of the inputs in the measurement file. Here we should specify the input to i. Click on i or in the right column. A selector appears in the right column. Select “Current [A]” as it is called in the csv file.

Couple current as input.



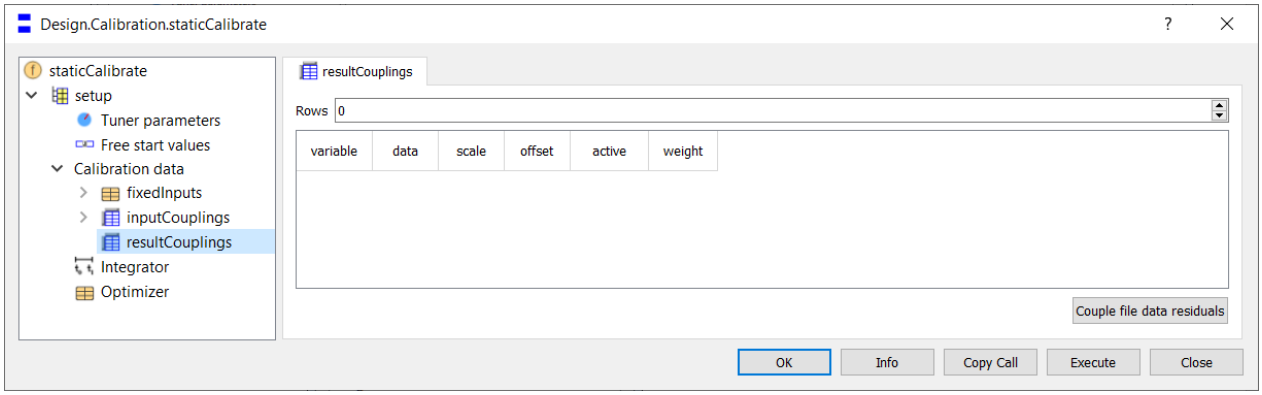
Click **OK**. The result of coupling file data to the inputs is displayed as



The input of the model as well as the measured current are given in the SI unit, A, so there is no need for scaling. In case of different units being used, the menu supports rescaling (variable = data*scale + offset). For example, if the measurements of the current had been given in mA, we had needed to downscale them by a factor of 1000 by setting scale to 0.001.

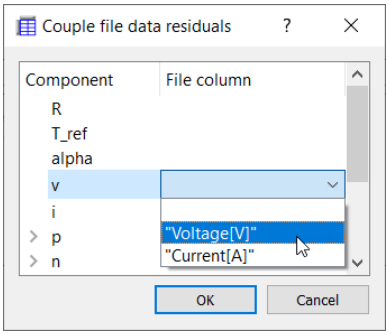
Result Couplings

The measurements of the voltage across the resistor are to be used for the calibration criterion. To specify this, click on **resultCouplings** in the tree browser.

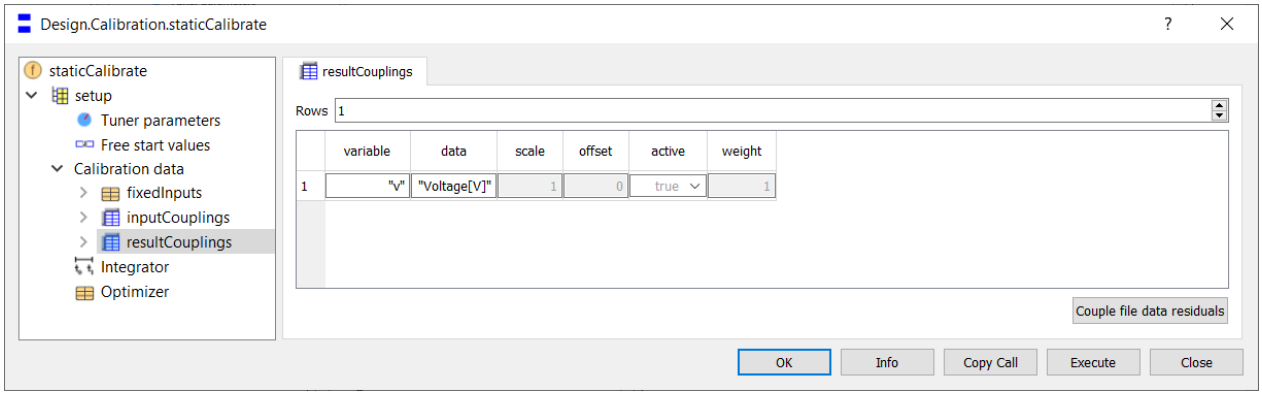


Couple output and data.

Click **Couple file data residuals**. A browser similar to that for connecting file data inputs is popped.



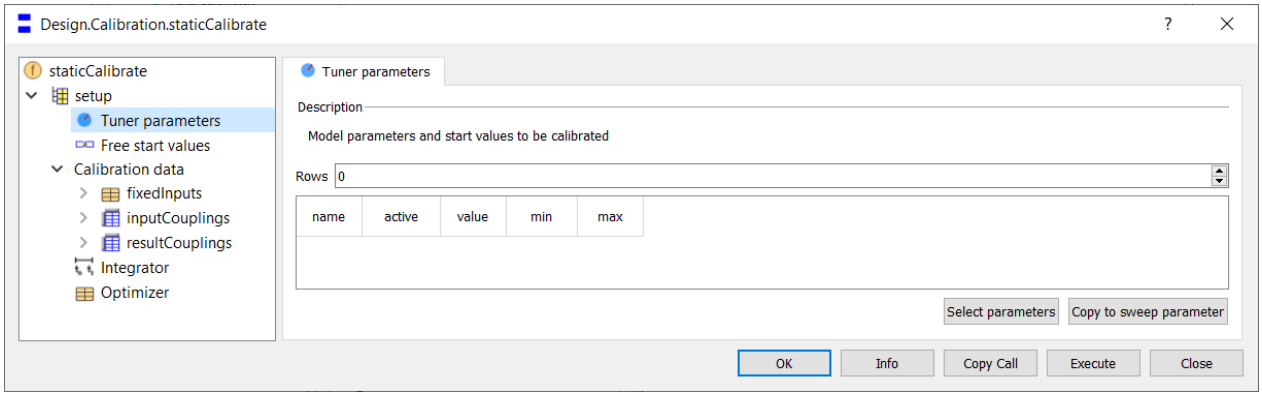
Select **v** to be compared with “Voltage [V]” in the csv file. Click **OK**. These measurement data are given V so no need for scaling.



We have now specified the instrumentation or the rigging of the experiments.

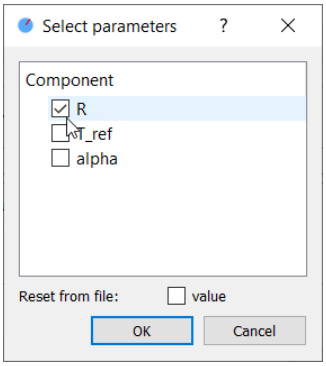
Tuner parameters

For the calibration task we need to specify which paramers to tune. Click on **Tuner parameters** in the tree browser.



Click **Select parameters**.

Select resistance for tuning.



The parameter of interest for us is R representing the resistance. Tick it and click **OK**.

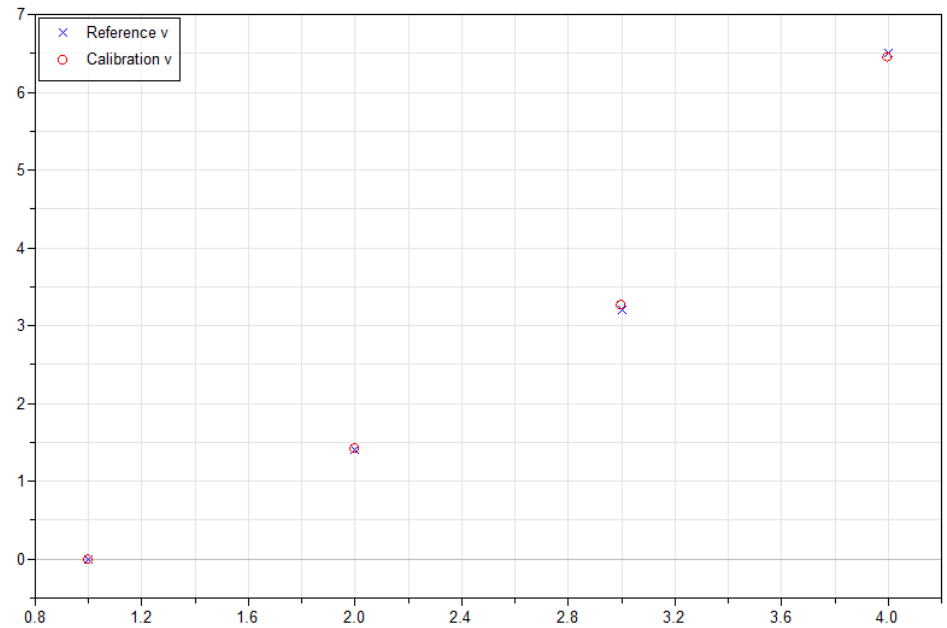
It is time to do the first calibration. Click **Execute**. After 13 fast iterations we obtain the result.

Tuning result.

```
R = 4611.36,
```

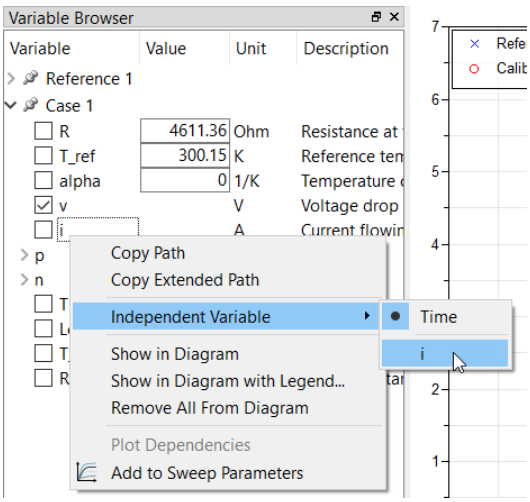
with the error being 0.0083 and a plot comparing measured reference and simulated voltage from tuned model for the different cases (the legend of the plot has to be moved to see the last point).

Comparing measured and calculated voltage drop after tuning.



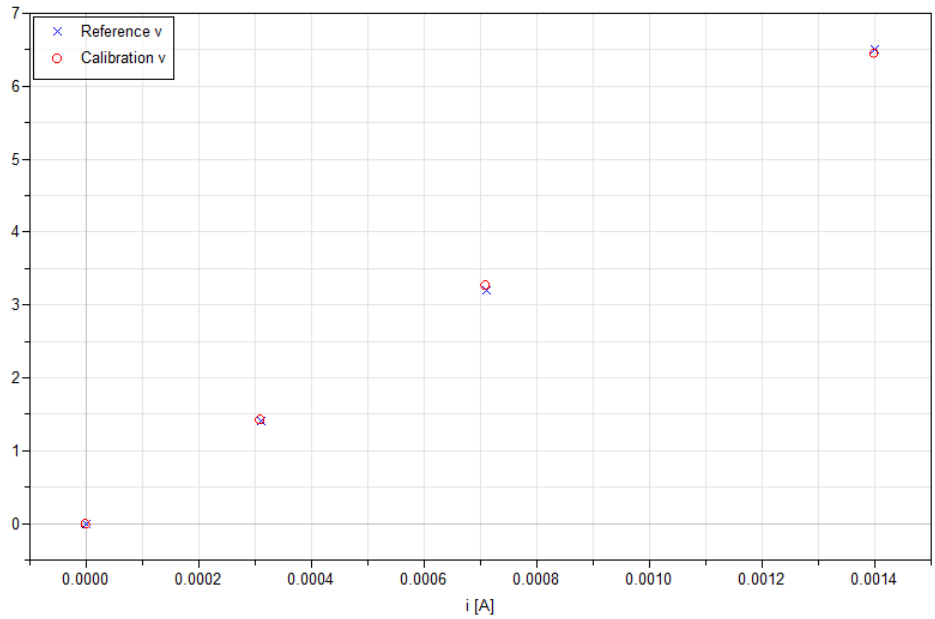
The fit is rather good. To get better insight, it may be of interest to plot versus current. In the variable browser we find two results; the Reference and the simulation result Case. For these two results v is plotted. Put the cursor on either i, right-click and select **i** as independent variable.

The results are easily accessible in the variable browser.



This plot now becomes

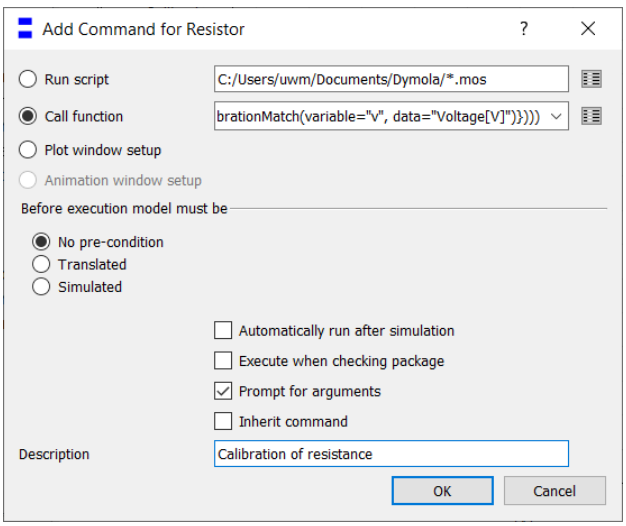
Current v s voltage drop.



Saving the setup for reuse

After an execution of a command we can save it in the model for later reuse as described above. Select **Simulation > Commands > Add command**. A menu pops up:

Menu for saving the setup.



Tick **Prompt for arguments** and enter a description, which will be used in the commands menu. Do not tick that the model must be translated before execution, because this would mean translation of the model Resistor without the additional inputs and consequently Dymola would generate error messages that the model is singular. (More about this menu can

be read in chapter “Simulating a model”, section “Editor command reference when working with a simulation”, sub-section “Main window: Simulation tab”.)

Direct calculation of R from the measurements

There is another way to get insight. The measurements can be used to uniquely back calculate R for each case. Make a copy of

```
Modelica.Electrical.Analog.Basic.Resistor
```

as described above. Call it ResistorBase. We need to free R; change the declaration of R

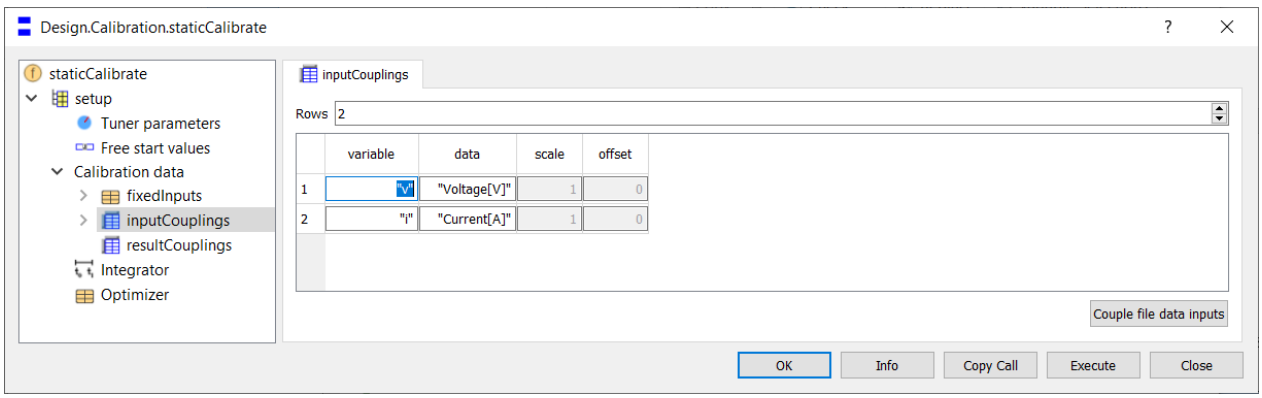
```
parameter Modelica.SIunits.Resistance R(start=1) "Resistance";
```

to

```
Modelica.SIunits.Resistance R "Resistance";
```

Take ResistorBase and set up a calibration task as done above,

- Select **staticCalibrate** in the package browser, right-click and select **Call function...**
- Select the model to be ResistorBase.
- Select the additional inputs as previously, but also select **v (v, i** and **n.v** should be selected).
- Select **Calibration data** and specify experimentName as previously.
- Specify fixedInputs and give their values as previously.
- Specify inputCouplings and scaling as previously, but couple also v to “Voltage [V]” (no scaling). The result is:



- Do not specify any resultCouplings.
- Do not specify any tuners.

Clicking **Execute** produces a result file. Plot R and select i as independent variable. It gives the plot below.

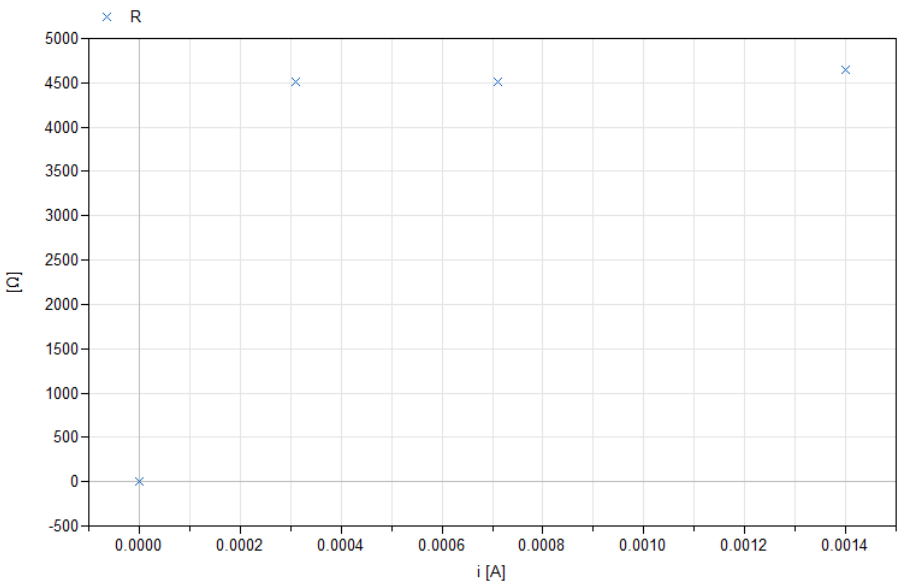
You can also define the setup by starting from the setup of the previous calibration.

- Select the model to be ResistorBase.

- Select the additional inputs as previously, but select also **v (v, i** and **n.v** should be selected).
- Select inputCouplings and select v and couple it to “Voltage [V]”. (You will now have the result as the previous figure.)
- Select **resultCouplings**. We want to deselect v. Decrease the number of rows to zero.
- Deselect tuners by decreasing the number of rows to zero (or by clicking **Select parameters** – no parameter is yet selected – and click **OK**).

Clicking **Execute** produces two results Reference and Case 1. Plot R and select i as independent variable. It gives the plot.

Obtained resistance values from direct calculation using the measurements.



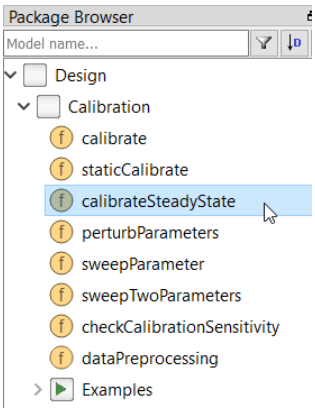
First, we realize that it is not possible to get a unique R when the current is zero. The measurements indicates that the resistance is independent of current. If they had indicated dependencies, this kind of plot can give us hint about useful parameterizations for the relation between voltage and current for our component.

7.7.2 The calibrateSteadyState function

The calibrateSteadyState function should be used for steady-state models. For completely state-less (static) models, please see the function staticCalibrate.

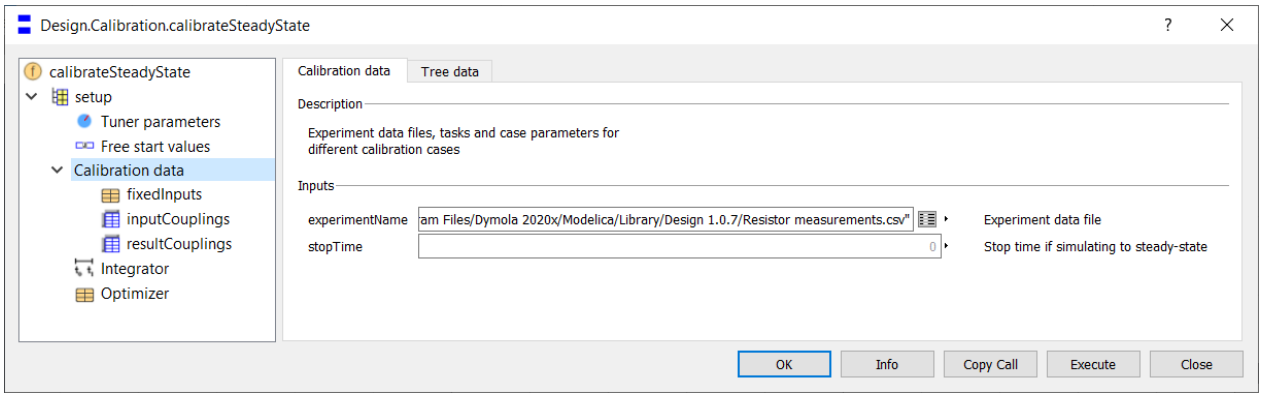
To load Design.Calibration, select **File > Libraries** and click **Design**.

The calibrateSteadyState function.



Example

The resistor example above can be solved using calibrateSteadyState as well; the only difference in the menus compared with the above example is the **Calibration data** menu:



Here also **stopTime** can be specified if a steady-state model should be treated. The time should be specified by the user when a model is not initially in steady-state. The time should be stated in such a way that steady-state conditions can be guaranteed at that time.

8 DESIGN OPTIMIZATION

8 Design Optimization

Design optimization is available in two ways:

- As an older implementation "Optimization 1.x" in the Design library; the Design.Optimization package.
- A new implementation, the new library Optimization 2.x. Optimization 2.x has largely been re-implemented and improved and is not backwards compatible with the older Design.Optimization package ("Optimization 1.x").

Both libraries are available and can be used at the same time; existing optimization setups work without changes. However, wanting to take advantage of the new features, Optimization 2.x has to be used.

Given the fact that the new library has existed for a number of Dymola versions, the documentation of the old package (that previously was the content of this chapter) is from this version removed. For the documentation for this Design.Optimization package, see the corresponding chapter "Design Optimization" in the manual "*Dymola User Manual Volume 2*" in previous Dymola versions (Dymola 2020x and earlier).

You can open the new Optimization library with the command **File > Libraries > Optimization**. The documentation is available in the library, and the pdf manual is also available by the command **Tools > Help Documents**.

9 MODEL MANAGEMENT

9 Model Management

The model management package includes version management, automatic documentation of model dependencies, encryption of models, model and library checking, model comparison, and functions for accessing the model structure. There is no licensing for Model Management.

Note!

In Dymola 2017 FD01, the Model Management product (MOC) was integrated into the Dymola standard configuration, meaning that Model Management is from that version available without licensing.

9.1 Version management

9.1.1 Short guide with new features included

If you are familiar with version management, and just want a short guide to use the latest features (Dymola 2017 and later) please go to section “Short guide to version management with new features included” starting on page 1191.

9.1.2 The context of version management

In developing model components for a complex system such as a vehicle, many different kinds of competence are needed. Experts in engines, transmissions and chassis etc. are needed to develop a drive train. Because several people are involved in the process, it becomes essential to break up or decompose the overall problem into modular units during development.

As more people are involved in the process, the development is geographically and chronologically distributed because it is natural to have centers with specific core-competencies. This implies that the modular units developed must be seamlessly integrated to solve the overall problem, and the partitioning should be able to reflect the organizational structure of the model development teams.

In order to increase quality and reduce development time, tools should be made available to

- Provide a structure for organizing, storing and retrieving information (models, simulation results, documentation, and experiment data).
- Support the exchange of information and simplify reuse of models throughout the organization.
- Ensure that correct information is available to each user (versions of libraries, corresponding experiments).

A version control system provides means to track changes to a set of files. A “commit” operation associates a developer and documentation with each change to the common storage of files. The Modelica text of two versions can be compared, and it is possible to back up to any previous version.

The underlying version control system must be able to support multiple concurrent developers working on the same set of models. Extensive locking of files is undesirable in a collaborative environment, and more recent tools also support concurrent development of closely related parts (with appropriate safety nets). A single physical person may have multiple roles in the development or use of the library; one role as a developer for new features of the library, and one role fixing bugs in a release version of the library.

Traceability is essential for maintaining quality over time. Tool enforcement to document modifications before they become publicly available gives the opportunity to review changes and improves quality. The development history (documentation of changes) may also be needed for tracing model incompatibilities, for example.

Model testing should be integrated with model development, which implies that the version control system must be able to handle test scripts, support utilities and binary test data. Regression testing, where models are simulated and compared with known good simulation results, is very powerful in detecting involuntary changes to model libraries. A failed regression test may cause either a change of a model, or the revision of the test itself.

Multiple libraries are often used together. In this case, version compatibility across libraries becomes essential. It must be possible to “tag” releases of multiple libraries to indicate compatibility at the project level.

Dymola supports storing, retrieving, etc. of models in version control systems such as CVS (Concurrent Versions System), SVN (subversion) or Git. We have deliberately chosen to build on existing version control systems, which offers greater flexibility and better integration than a proprietary system. Because of the textual representation of models in the Modelica language, existing text-based tools can be used, for example, to compare versions. To browse changes in large systems, support in the graphical environment of Dymola would be needed.

The use of public libraries has increased in industry over several years. More recent is "open source development", which can be described as the loosely organized development (typically of software) by several geographically separated parties. Public websites, such as SourceForge, support Open Source development with web-based tools and CVS/SVN. The Modelica Standard Library is maintained as a project on a server.

9.1.3 Introduction to version management

If you are familiar with version management, skip this section. This section describes the principles, for details please see later detailed description.

General principles

If a version management system is in place (including a suitable local directory structure), the user will always work with a *local copy* of the files in the *repository* (with one exception). Some typical work flows will be:

Making changes to an existing file that the user has not worked with before

Typically, a user intends to add some code to a file that is included in the version management system. If the user has not worked with this file before, the following will be the work flow:

1. The user *checks out* the appropriate file by using a command in the version management system. What will happen is that the version management system will create a *local copy* of the appropriate file on the user's local hard disk.
2. The user makes changes in this file copy. When the user saves the changes, the changes are however only saved locally, that is, on the user's own local directory.
3. When the user finds it appropriate the changes can be made available to all users of the version management system. This is done by a *commit* command in the version management system. The basic idea of a commit is to save the changed file in the repository (that can be located in another computer, e.g. a server). However, since many users can work on the same file the operation must be done in two steps. The first step will be that the version management system compares the version of the file initially checked out (copied) to the hard disk of the user and the present version on the repository. Two alternatives occur:
   a. The file version is the same. That means that no changes have been made in the file on the repository since the user checked out the local copy. The second step is easy – the local copy will now be saved in the repository (with an updated version number). The old file on the repository will not be overwritten – it will always be possible to *revert* to an older version if necessary.
   b. If the versions are not the same it means that *some other user* already has updated that file in the repository. The user gets at warning and can compare the file in the repository and the local file and take proper actions, e.g. merge the changes. The resulting file can then be saved in the repository (with an updated version number). The old file on the repository will not be overwritten – it should always be possible to *revert* to an older version if necessary.

Making changes to existing files that the user has been working with before

Being on vacation a couple of weeks, the user is now requested to make another change in a file that the user has been working with before. In this case the user already has the local copy on the hard disk and need not to (can't) check it out, but since other users might have changed files during the vacation, the user *updates* the files using commands in the version management system. An updating command for a certain file means that the version management system compares the version of the local file to the version of the corresponding file in the repository. If the repository holds a file that is newer, the local file is deleted and a new (updated) local file is created. Now the user can start working with that file. The user continues to work like in point 2 and 3 above. Please see those.

Creating new files that should be included in the version management system (available for other users)

Sometimes the user wants to create a new file that should be available to all users of the version management system. What the user has to do is the following:

1. The file cannot be created in "empty space"; it must be created in a directory that has been checked out by a command in the version management system. In most cases such a directory is already available (when creating a new model in Dymola it can be stored where other models are already present that are handled by the version management system). In other words the user has to create a local file in a directory that has been previously defined to hold local copies of files from the repository. Please be cautious – you must ***not*** try to create any file ***directly*** in the repository!
2. The existence of the file must be made recognized by the version management system – otherwise it cannot handle it. This is done using a command in the version management system (Dymola uses the command **Add Model**). Please note that yet no such file is in the repository, but now it is possible to create such a file.
3. Now a *commit* command can be done for this file, which in this case will mean that the file is copied to the repository. Now everything is set, and the user can continue to work as if the file existed in the repository from the very beginning. (This was by the way the exception to the rule to only work with a local copy; in this case the copy was made before making the "original"…)

These are the principles; of course there is more to it. Comments can (should) be made when committing files, files can be compared without committing; version history can be reviewed etc etc. See the following sections for details.

Please note that an ordinary user never works directly with the files in the repository! That is the job of the version management system – the user only works with the local copies – and the version management system.

Deploying a version management system

If the version management system is *not* in place, the user must install one. The following will just give the basics; the details are described in later sections. Only basic principles are described (local repository), not e.g. the handling of (distributed) remote servers etc. The idea is to give an idea of what might be done.

- External software might have to be installed. This is the case for SVN and Git, but not for CVS.
- A repository has to be set up, that is, the directory structure that should be used by the management system as the repository must be defined. This is done by certain commands. Definition of environment variables might be included.
- A local copy of the repository structure has to be checked out. This will be where the local files will be modified – the "working directory".

9.1.4 Scope of implementation

This is a description of minimal support for version management in Dymola. The strategy is to provide a relatively thin layer on top of an existing version management system, such as, CVS (Concurrent Versions System) SVN (Subversion), or Git.

The added value for the user, compared to using existing graphical user interfaces e.g. WinCVS or TortoiseSVN, is:

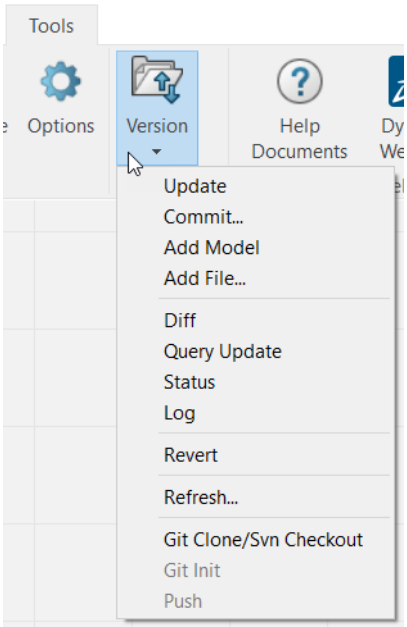
- Commands are integrated in the Dymola environment. No need to swap between different applications. Some information is easily accessible in Dymola, e.g. version number and date.
- Some steps have been automated. For example, Dymola knows the filename of the current class, knows if there are files which have been modified, etc.
- Files are automatically reloaded into Dymola after updates from the repository.

However, there is no need to provide a comprehensive version management environment in Dymola. More complex tasks are better performed in specialized tool such as WinCVS, TortoiseCVS, TortoiseSVN or RapidSVN.

9.1.5 Supported features

Dymola provides a graphical user interface to the most basic CVS, SVN, and Git commands, where the principal automatic step is to provide the correct file name in which the model the user is located.

The primary commands which can be reached using **Tools > Version** are (in CVS terminology):



Update

Updates your local copy of the file with changes from the repository. If your file has been changed since it was last updated, your changes are merged with the changes made to the repository. After a successful update the file is reloaded into Dymola.

If conflicts arise during the merge, this is noted in the message window, and the file is not reloaded into Dymola.

See section “Query Update” on page 1173 for an explanation of the status code displayed in the message window.

For Git the Update command works as the Git command Pull.

Commit...

Updates the repository with changes you have made in your local file. Your file is first checked to make sure that you have an up-to-date copy. You are then asked to enter a description of the changes, which is later available through the Log command.

Note that for Dymola 2017 and later, SVN and Git commits the entire directory, not specific files.

Add Model

Makes a new model’s file known to the underlying version management system. The user must then perform a Commit on the model.

Add File...

Makes an arbitrary file known to the version management system. The user must select the file using a file browser.

Diff

Displays the textual differences between your local file and the corresponding version in the repository.

Query Update

Displays which files in the model's directory are

- Locally modified compared to the corresponding version in the repository (marked by "M" before the filename).
- Changed in the repository compared to the version that was checked out ("U" or "P").
- Caused a conflict during an Update operation, or which could potentially create a conflict because it is both locally modified and changed in the repository ("C").
- Added but not yet committed ("A").
- Unknown to the version management system ("?").

Local files are not updated. The repository is not changed.

Status

Displays version status of the file. The information includes:

- If the file is up-to-date, needs an Update, or has been locally changed.
- Revision of your local file and the repository file.
- A list of all symbolic tags and which revisions they refer to.

Log

Displays log messages which were entered every time the file was committed, and a list of all symbolic tags and which revisions they refer to

Revert

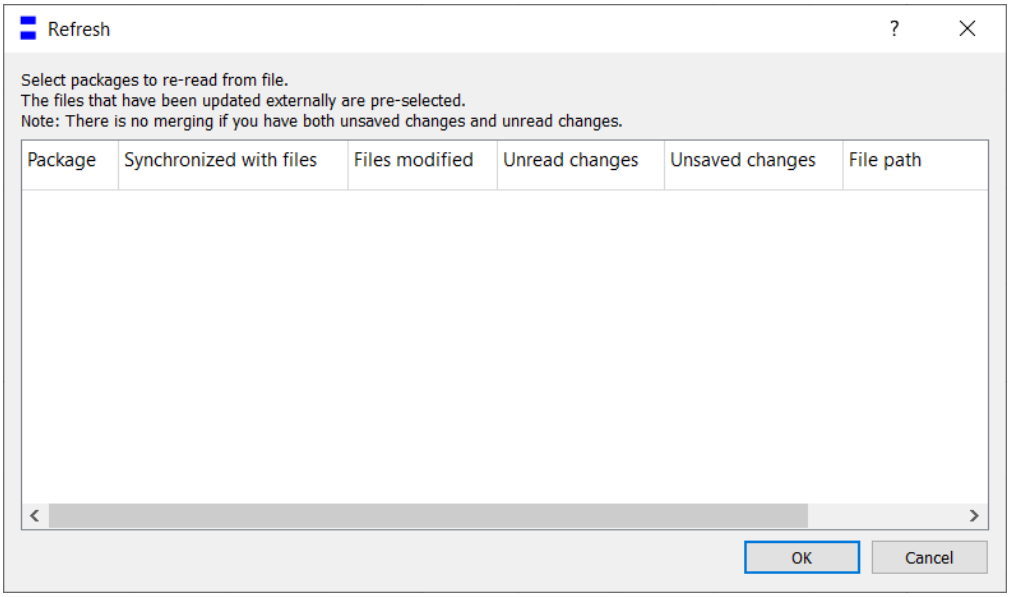
Deletes your local file and retrieves the latest version from the repository. All changes to your local file are lost.

All version management systems operate on files. An environment which would allow version management of individual models even when several models are stored in the same file could be implemented on top of external tools, but would be quite complex. However, Dymola can easily map from model to the corresponding filename, and also knows when a model is part of a larger package comprising several files (in which case updates probably should be made on all files).

Also note that CVS can update special “keywords” in the Modelica text, which can be used to automatically insert information in the model documentation. They include version number, date of last change, and a log describing all changes. An example of this is given below.

Refresh...

The command can be used to refresh selected files. A menu is displayed:



Git Clone/Svn Checkout

See section “Short guide to version management with new features included” starting on page 1191.

Git Init

See section “Short guide to version management with new features included” starting on page 1191.

Push

Git command, commits to remote Git repository. For Git Pull, see the Update command above.

Conflict handling after update

If several users have modified the file, the “update” command will attempt to merge the changes. If they have modified the same lines of code, CVS will detect a conflict. After a conflict the original modified file is kept as backup, and the merged file contains both sets of changes marked by special indicators inserted into the text. It is then up to the user to resolve the conflicts.

An important issue here is that Dymola cannot use the file until conflicts have been resolved. Initially we do nothing, i.e., require that the user edits the Modelica file with some external text editor to delete conflicting lines and their indicators. At some future point in time Dymola could be extended to parse Modelica text with CVS conflict indicators, and the resolution could be handled from within Dymola (which of course has better support for analyzing the conflicts). An intermediate step is to rename the file with conflicts and restore the backup; this will at least maintain consistency between the Dymola environment internally and the corresponding file externally.

It should be noted that merge conflicts arise from a people management problem, and are rare in practice. Normally people working on a project do not edit the same code.

Version management of non-model files

The discussion of version management is naturally focused on Modelica code, but the facilities also handle parameter sets, experiments and trajectories in large projects.

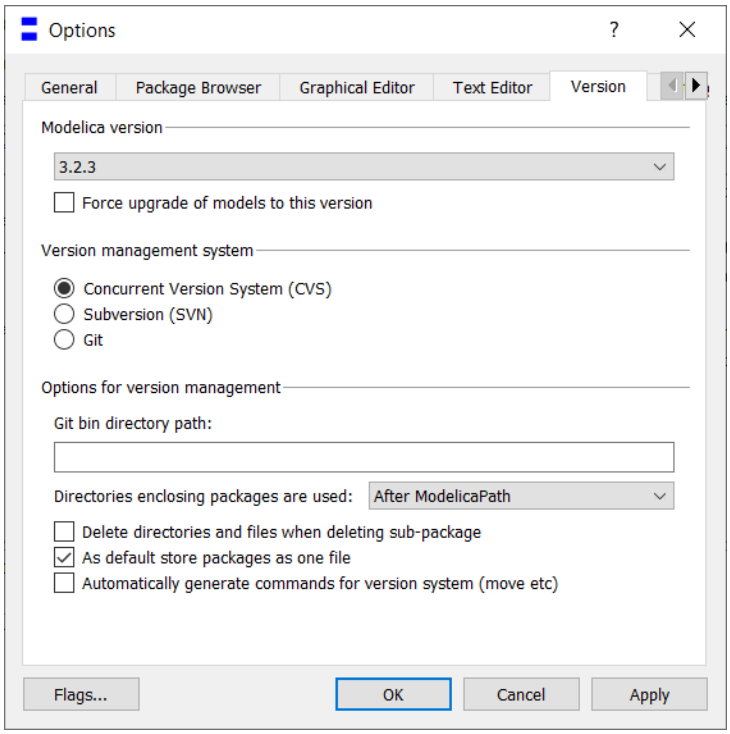
Parameter sets and experiments can be represented by Modelica models. In this case the experiment extends from the top-level model and provides experiment-specific parameters through modifiers of the extends clause. An alternative is to set up the experiment with a Modelica script file (extension .mos). Trajectories are represented by binary files (extension .mat).

Files which are not Modelica text can be stored in hierarchical Modelica packages. These files are added to the repository using the **Add File...** command. Operations performed on the package will then also operate on the contained .mos and .mat files. Operations supported include **Update**, **Commit** and **Status**.

9.1.6 Selecting version management system

Dymola supports three version control systems, CVS, SVN, and Git, and generates the appropriate external commands to perform operations on the version control system. Which system is used is set in **Tools > Options > Version**.

Setting version management system.



For options for versions management, see section "Short guide to version management with new features included" starting on page 1191.

The settings for version management system are stored between sessions.

9.1.7 Version management using CVS

Version management support in Dymola assumes that there exists a functional CVS environment. In its simplest form there exists a CVS repository on a local disk. More advanced installations maintain a CVS server on a separate UNIX system; one such setup is the use of the SourceForge server to maintain the Modelica standard library. Two examples are given below.

It is worth pointing out that Dymola and the underlying CVS system supports development of libraries maintained at several different servers concurrently. For example, the Modelica standard library may be maintained at SourceForge, other libraries proprietary to the company, and still others by the user on a local disk. In this fashion version management also facilitates effective distribution of updates as they become available from the vendor.

Location of the CVS command

Location of the CVS command.

Note: in several places the user is asked to execute the CVS command. The file cvs.exe is located in the Dymola distribution, typically \dymola <version>\bin. A command for

initialization of a repository directory `CVS_Repository` with the full path written might be e.g.

```
"C:\Program Files\Dymola 2023\bin\cvs" -d \CVS_Repository init
```

In order to avoid long paths the path to cvs.exe (in this case `C:\Program Files\Dymola 2023\bin`) can be added to the environment variable PATH. This is done the following way:

- Use the Windows **Start** Button, select **Settings**.
- Search for "System variables", select the proposed **Edit the systems environment variables**.
- In the **Advanced** tab of System Properties, click **Environment variables…**.
- In the **System Variables** pane, select the variable **Path**. Click on **Edit**.
- Click **New** and enter `C:\Program Files\Dymola 2023\bin`
- Click **OK** in three consecutive menus.

Please note:

- You need to have administrator rights to edit environment variables.
- You must open a new DOS command window after changing the environment variable!

Now the path can be omitted in the commands. For conciseness we will use that form in the following; the command above will now be `cvs -d \CVS_Repository init`.

Local CVS repository

To set up a local CVS repository, first choose the machine and disk on which you want to store the revision history of the source files. CPU and memory requirements are modest, so most machines should be adequate.

To create a repository with a set of configuration files, run the "cvs init" DOS command to set up an empty repository in the designated directory.

```
mkdir \CVS_Repository
cvs -d \CVS_Repository init
```

These steps complete the initialization of the CVS repository. The "init" command is careful to never overwrite any existing files in the repository, so no harm is done if you run cvs init on an already set-up repository.

Note that if you use a Windows drive letter, you must write a slightly longer repository specification because the "cvs" command interprets the colon after the drive letter:

```
cvs -d ":local:c:\CVS_Repository" init
```

The CVS repository is initially empty. It is necessary to create one or more directories which act as top-level directories for further development. For example, we may create a "models" directory:

```
mkdir \CVS_Repository\models
```

To use a CVS repository it is necessary to initially perform a "checkout" operation:

```
cvs -d \CVS_Repository checkout models
```

This command creates a "models" sub-directory with all models currently stored in the corresponding top-level directory in the CVS repository. It also creates extra directories called "CVS" at each level, which are used to maintain CVS status information. The files inside the "CVS" directories should never be manipulated by hand.

Access to servers via CVS

Projects maintained at SourceForge (http://www.sourceforge.net) or other servers can be accessed via CVS. To access a Modelica area via CVS, you set up your CVSROOT when the files are initially checked out, and do a "cvs login" with an empty password. After that the usual CVS commands work as expected.

If you work against a single CVS repository it may be convenient to set the CVSROOT environment variable to the value below, as an alternative to using the -d command line switch:

```
:pserver:anonymous@myproject.sourceforge.net:/cvsroot/modelica
```

To use it you must first login and then check out using these DOS commands:

```
cvs login              // enter password
cvs checkout models    // check out models library
```

These will checkout a Modelica library in the current directory.

9.1.8 An example of file management using CVS

In this example we will demonstrate the basic version management operations provided by Dymola. It is divided into several different steps to setup a local CVS repository, to create a new model, and to make changes to an existing model.

Setting up the CVS repository

A local CVS repository is set up (we choose to start from the directory `C:\MyWorkspace` in this example and we assume Dymola 2023\bin to be located according to the path used below), and then a new top-level directory called "models" is created. Finally the new top-level directory needs to be checked out in the current working directory.

DOS commands.

Execute these DOS commands in the directory `C:\MyWorkspace`:

```
mkdir \CVS_Repository
"C:\Program Files\Dymola 2023\bin\cvs" -d \CVS_Repository init
mkdir \CVS_Repository\models
"C:\Program Files\Dymola 2023\bin\cvs" -d \CVS_Repository
checkout
  models
```

The first command creates an empty folder `CVS_Repository` at the root level (`C:\CVS_Repository`).

The second command declares the folder `CVS_Repository` as a CVS repository and creates a folder `CVSROOT` inside it. (Inside that folder a number of files are created that are used by the cvs system.).

The third command creates an empty folder `models` in the folder `CVS_Repository` – the resulting folder is `C:\CVS_Repository\models`. (This is one of the few cases when the user tampers with the CVS_Repository folder - when the folder system for the cvs handling is created.)

The forth command will give the answer `cvs checkout: Updating models`. The command creates a (very specific) copy of the folder `models` in the folder `MyWorkspace` (the result is `C:\MyWorkspace\models`). Please note that inside this folder a new folder `CVS` is created. This folder is part of the cvs handling – each folder that contains files that should be handled by the cvs system will contain such a folder! This folder should never be tampered with.

The result of these commands is that we have a folder `C:\MyWorkspace\models` where we should put the Dymola models that we create. This folder is handled by the cvs system, so cvs commands can be applied to the files inside it.

(Shorter paths in the commands above can be used if the environment variable PATH has been modified; please see above.)

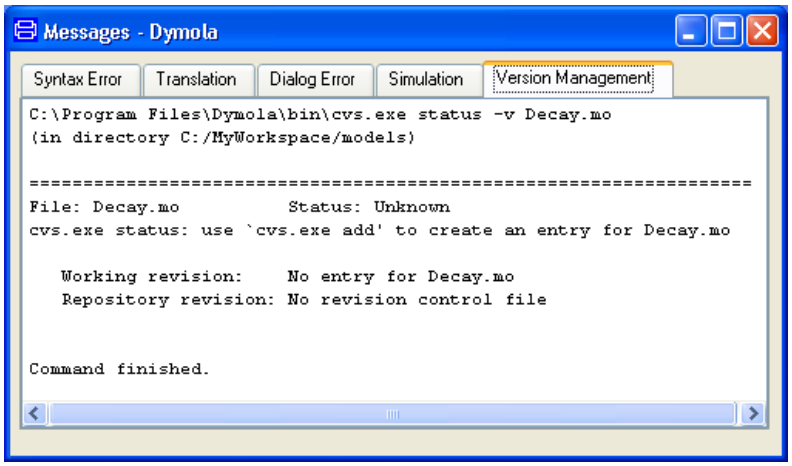
Creating a new model

We can start by creating a model in Dymola and saving it in the `C:\MyWorkspace\models` folder (***not*** in `C:\CVS_Repository\models`!). For our example we will use the simple model:

```
model Decay
  Real x(start=2);
equation
  der(x) = -x;
end Decay;
```

Initially the model is unknown to the version management system. For example, using the command **Tools > Version > Status** returns this information in the message window:

Information in the Message window.



Informaton in the Message window.

Next we perform the command **Tools > Version > Add Model** to make the model's file known to the version management system. The message in the Message window will be the following:

```
C:\Program Files\Dymola 2023\bin\cvs.exe add Decay.mo
(in directory C:/MyWorkspace/models)
cvs.exe add: scheduling file `Decay.mo' for addition
cvs.exe add: use 'cvs.exe commit' to add this file permanently
Command finished.
```

Informaton in the Message window.

The information from the command **Tools > Version > Status** is now different, but there is no file in the repository yet (not until we commit the file).

```
C:\Program Files\Dymola 2023\bin\cvs.exe status -v Decay.mo
(in directory C:/MyWorkspace/models)

===================================================================
File: Decay.mo           Status: Locally Added
   Working revision:    New file!
   Repository revision: No revision control file
   Sticky Tag:   (none)
   Sticky Date:         (none)
   Sticky Options:      (none)
Command finished.
```

Informaton in the Message window.

When we perform **Tools > Version > Commit...**, the user is asked to enter a log message describing what changes are committed. The lines beginning with CVS are generated to help us remember the nature of the commit.

```
This is the first version of our test example.
CVS: ----------------------------------------------------
CVS: Enter Log.  Lines beginning with `CVS:' are removed
automatically
CVS: Added Files:
CVS:      Decay.mo
```

```
CVS: ----------------------------------------------------------
```

When having entered the log message, save the changes using **Tools > Save** or **Ctrl + S**, then close the window.

The message after the commit operation has finished looks like this:

```
C:\Program Files\Dymola 2023\bin\cvs.exe commit Decay.mo
(in directory C:/MyWorkspace/models)
RCS file: \CVS_Repository/models/Decay.mo,v
done
Checking in Decay.mo;
\CVS_Repository/models/Decay.mo,v  <--  Decay.mo
initial revision: 1.1
done
Command finished.
```

The output from **Tools > Version > Status** now contains more information, in particular the version number of the file and the date it was last changed in the repository.

```
C:\Program Files\Dymola 2023\bin\cvs.exe status -v Decay.mo
(in directory C:/MyWorkspace/models)
===================================================================
File: Decay.mo           Status: Up-to-date
   Working revision:    1.1      Fri Oct 04 09:34:02 2005
   Repository revision: 1.1   \CVS_Repository/models/Decay.mo,v
   Sticky Tag:   (none)
   Sticky Date:         (none)
   Sticky Options:      (none)
   Existing Tags:
        No Tags Exist
Command finished.
```

It is also possible to view the change log with **Tools > Version > Log**. The change log contains all messages entered during commit operations.

```
C:\Program Files\Dymola 2023\bin\cvs.exe log Decay.mo
(in directory C:/MyWorkspace/models)
RCS file: \CVS_Repository/models/Decay.mo,v
Working file: Decay.mo
head: 1.1
branch:
locks: strict
access list:
symbolic names:
keyword substitution: kv
total revisions: 1;     selected revisions: 1
description:
----------------------------
revision 1.1
date: 2005/10/04 09:34:02;  author: Dag;  state: Exp;
This is the first version of our test example.
=============================================================
Command finished.
```

The output from both **Tools > Version > Status** and **Tools > Version > Log** contain information specific to the underlying CVS system, which is beyond the scope of this report. For non-expert users it would be beneficial to filter the raw output.

Changing an existing model

Starting with the model created above, we now modify it by adding a time constant Ti. The revised Modelica text looks like this:

```
model Decay
  Real x(start=2);
  parameter Real Ti=1;
equation
  der(x) = -x/Ti;
end Decay;
```

After changing the model, it must be saved using **File > Save > Save**.

The **Tools > Version > Diff** command will display the differences between the model stored in the repository and the current model. Changed lines are indicated by "!", added lines by "+" and any removed lines by "-" (this is the so-called "context diff" format).

```
C:\Program Files\Dymola 2023\bin\cvs.exe diff -c Decay.mo
(in directory C:/MyWorkspace/models)
Index: Decay.mo
===================================================================
RCS file: \CVS_Repository/models/Decay.mo,v
retrieving revision 1.1
diff -c -w -r1.1 Decay.mo
*** Decay.mo     2005/10/04 09:34:02 1.1
--- Decay.mo     2005/10/04 09:43:12
***************
*** 1,6 ****
  within ;
  model Decay
    Real x(start=2);
  equation
!   der(x) = -x;
  end Decay;
--- 1,7 ----
  within ;
  model Decay
    Real x(start=2);
+   parameter Real Ti=1;
  equation
!   der(x) = -x/Ti;
  end Decay;
Command finished.
```

The **Tools > Version > Query Update** command is used to quickly list which files have been locally modified (indicated by "M") or need to be updated from the repository ("U"). Files that are not included in the version handling will be marked with "?".

```
C:\Program Files\Dymola 2023\bin\cvs.exe -qn update
(in directory C:/MyWorkspace/models)
```

```
M Decay.mo
? dsin.txt
Command finished.
```

The **Tools > Version > Query Update** command does not operate only on the file of the model. Instead it operates on the entire directory and all sub-directories; this makes it particularly useful to concisely review the status of all files in a complex model hierarchy.

The model is then committed to the repository with **Tools > Version > Commit**, as shown above. If we review the log with **Tools > Version > Log**, we see that the new revision comment is also listed. The listing also shows the number of changed Modelica text lines.

```
C:\Program Files\Dymola 2023\bin\cvs.exe log Decay.mo
(in directory C:/MyWorkspace/models)
RCS file: \CVS_Repository/models/Decay.mo,v
Working file: Decay.mo
head: 1.2
branch:
locks: strict
access list:
symbolic names:
keyword substitution: kv
total revisions: 2;     selected revisions: 2
description:
----------------------------
revision 1.2
date: 2005/10/04 09:44:57;  author: Dag;  state: Exp;  lines:
+2 -1
Added time constant Ti.
----------------------------
revision 1.1
date: 2005/10/04 09:34:02;  author: Dag;  state: Exp;
This is the first version of our test example.
=============================================================
Command finished.
```

This concludes the demonstration of how models are edited in co-operation with the version management facilities in Dymola.

Use of revision information

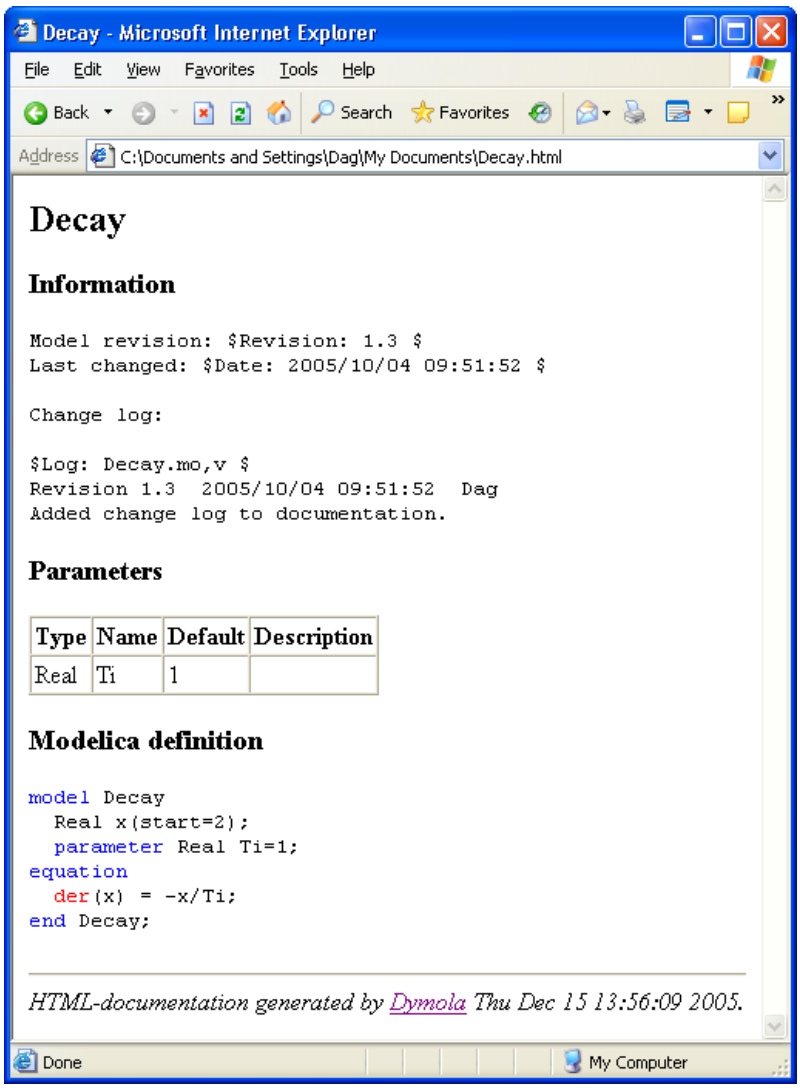
The underlying CVS system supports expansion of particular keywords, for example to automatically document the revision or commit date of the model. We could for example enter this text in the revision part of the documentation layer of our model:

```
<PRE>
Model revision: $Revision$
Last changed: $Date$

Change log:

$Log$
</PRE>
```

The keywords indicated by $ will be expanded at the next commit operation. Since this means that the committed file actually will be changed due to the expansion of the keywords, the local copy needs to be updated after the commit. The result after an update is shown in the following example of HTML documentation:



9.1.9 Version management using SVN

Version management support in Dymola assumes that there exists a functional SVN (subversion) environment. In its simplest form there exists a SVN repository on a local disk. More advanced installations maintain a SVN server on a separate UNIX system.

It is worth pointing out that Dymola and the underlying SVN system supports development of libraries maintained at several different servers concurrently. For example, the Modelica standard library may be maintained at svn.modelica.org, other libraries proprietary to the company, and still others by the user on a local disk. In this fashion version management also facilitates effective distribution of updates as they become available from the vendor.

SVN editor setup.

Some SVN operations require input from the user, for example a log message when a file is committed. To enable this feature the user must set either of the environment variables SVN_SETUP, EDITOR or VISUAL to the name of a text editor. Windows "notepad" will be sufficient for most uses.

Location of the SVN command.

Note: in several places the user is asked to execute SVN commands. The files svn.exe and svnadmin.exe should be available from the Windows Command Prompt or Dymola command input line if you have performed the default installation of SVN (see section "References" on page 1192). Please also see the example below.

Local SVN repository

To set up a local SVN repository, first choose the machine and disk on which you want to store the revision history of the source files. CPU and memory requirements are modest, so most machines should be adequate.

DOS command.

To create a repository with a set of configuration files, run the "svnadmin create" DOS command to set up an empty repository in the designated directory. For example,

```
svnadmin create \SVN_Repository
```

DOS commands.

The SVN documentation suggests that you populate the repository with three directories called "branches", "tags" and "trunks". The easiest way to do that is to create these directories locally and then import them:

```
mkdir models
cd models
mkdir branches
mkdir tags
mkdir trunk
cd ..
svn import models file:///SVN_Repository/models -m "Initial
  import"
```

A user name might be requested after the last command. An empty one might be sufficient. SVN will report that it has imported the directories as revision 1. It is worth noting that SVN manages directories as well as files, whereas CVS only manages files directly and implicitly creates directories as needed.

DOS command.

These steps complete the initialization of the SVN repository. Remove the local "models" directory to start over.

```
rmdir /s models
```

DOS commands.

To use a SVN repository it is necessary to initially perform a "checkout" operation to create a local copy with files that can be modified

```
svn checkout file:///SVN_Repository/models/trunk  models
```

This command creates a "models" sub-directory with all models currently stored in the corresponding top-level directory in the SVN repository. It also creates extra directories called ".svn" at each level, which are used to maintain SVN status information. The files inside the ".svn" directories should never be manipulated by hand.

9.1.10 An example of file management using SVN

In this example we will demonstrate the basic version management operations provided by Dymola. The example shows the first steps from the CVS-based example above.

Please note that a SVN client must be downloaded and installed first. We assume that this is done. Dymola must be restarted afterwards since the environment variable PATH is modified by the installation. The same goes for any Windows Command Prompt. (If the DOS commands do not work check that the path to svn.exe has been added to the environment variable PATH. For handling of the environment variable PATH, please see section "Location of the CVS command" on page 1176. The handling is analogue but this path is of course different.

Please note that the location when executing the DOS commands should be the directory `C:\MyWorkspace` if the example should be the same as the CVS example above. We use a directory "testmodels" instead of "models" in this example.

Of course SVN must be selected in Dymola as the system to use, please see section "Selecting version management system" on page 1175.

We must define an environment variable that defines the text editor to use. We choose to use notepad, and choose to define this in the EDITOR environment variable. Please see the section "Location of the CVS command" on page 1176 for a description how to edit environment variables in MS Windows.

DOS commands.

Now setup the SVN repository with initial directories, remove the initial local models directory and check it out. In this example the DOS commands (we assume being in the directory `C:\MyWorkspace`) would be (please note that the svn import command is so long that two lines are needed in the print below):

```
svnadmin create \SVN_Repository
mkdir testmodels
cd testmodels
mkdir branches
mkdir tags
mkdir trunk
cd ..
svn import testmodels file:///SVN_Repository/testmodels -m
  "Initial import"
rmdir /s testmodels
svn checkout file:///SVN_Repository/testmodels/trunk testmodels
```

The first command creates a folder `SVN_Repository` at the root level (`C:\SVN_Repository`). The folder contains a number of folders and files and will work as the repository. This folder should not be manipulated directly by the user.

The second command creates an empty folder `testmodels` at the `MyWorkspace` level. The resulting folder will be `C:\MyWorkspace\testmodels`.

The third command changes the directory of where the commands are given in the DOS window (the location) to `C:\MyWorkspace\testmodels`.

The fourth, fifth and sixth command creates three empty folders (`branches`, `tags` and `trunks`) at this level – as example the first command will create the folder `C:\MyWorkspace\testmodels\branches`.

The seventh command changes the directory of where the commands are given in the DOS window (the location) to `C:\MyWorkspace`.

The eight command (occupying two lines above) imports the directory structure consisting of the folder `testmodels` and the folders inside it to the SVN repository (please note that there has to be a space before "Initial import"). The visible result of the command will be an answer that folders have been added and that the first version is archived. (The folder structure inside `C:\SVN_Repository` will however not be changed, the folder structure is handled differently in SVN compared to CVS.)

The ninth command removes the folder `testmodels` and including folders from the directory `C:\MyWorkspace`. The structure was imported into the SVN system and is not needed anymore. (Please compare with the next command.) You have to answer `Y` to acknowledge that the command should be executed.

The last command checks out the folder `testmodels` and including folders. The folder `C:\MyWorkspace\testmodels` will be recreated (however without the three previous folders inside). This folder is now included in the SVN version handling! This might be seen looking at the folder using explorer – the folder has a sign on it. Looking inside the folder a folder `.svn` is present. This folder should not be tampered with.

The result of these commands is that we have a folder `C:\MyWorkspace\testmodels` where we should put the Dymola models that we create. This folder is handled by the svn system, so svn commands can be applied to the files and folders inside it.

Let us assume that we entered username "Dag" when importing the folder `testmodels` in one of the commands above.

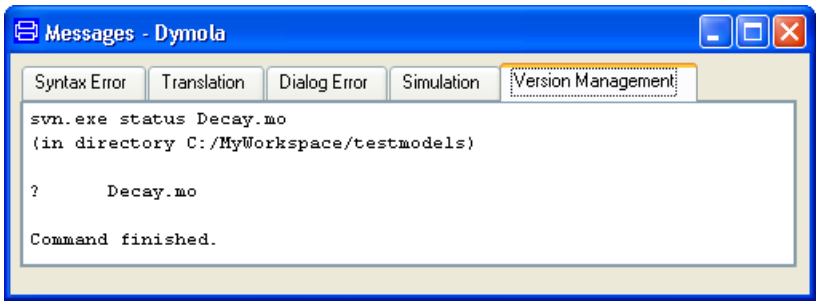
Creating a new model

We can start by creating a model in Dymola and saving it in the `testmodels` directory. For our example we will use the simple model:

```
model Decay
  Real x(start=2);
equation
  der(x) = -x;
end Decay;
```

Store the model in `C:\MyWorkspace\testmodels`. Initially the model is unknown to the version management system. For example, a **Tools > Version > Status** command returns this information in the message window:

Information in Message window.



Next we perform the **Tools > Version > Add Model** command to make the model's file known to the version management system. The system will respond with the message:

```
svn.exe add Decay.mo
(in directory C:/MyWorkspace/testmodels)

A       Decay.mo

Command finished.
```

We can now perform a **Tools > Version > Query Update** command to get some more information. The system will respond with the message:

```
svn.exe status -verbose --show-updates
(in directory C:/MyWorkspace/testmodels)

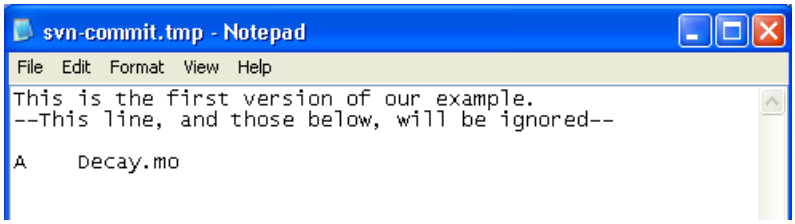
A               0       ?   ?           Decay.mo
                1        1 Dag          .
Status against revision:      1

Command finished.
```

"Dag" is the name user name entered when executing the DOS commands above.

When we perform **Tools > Version > Commit...**, the user is asked to enter a log message describing what changes are committed. Since we use notepad, the window after entering the comment might look like:

Inserting a commit comment.



The lines at the end (starting with "--This line") are generated by SVN to help us remember which file is committed. After the text has been entered (e.g. like the first line in the window above) save the changes using **File > Save > Save** or **Ctrl + S**, then close the window.

The message after the commit operation has finished looks like this:

```
svn.exe commit Decay.mo
(in directory C:/MyWorkspace/testmodels)

Adding         Decay.mo
Transmitting file data .
Committed revision 2.

Command finished.
```

It is also possible to view the change log with **Tools > Version > Log**. The change log contains all messages entered during commit operations.

```
svn.exe log Decay.mo
(in directory C:/MyWorkspace/testmodels)

------------------------------------------------------------
r2 | Dag | 2005-12-15 11:59:22 (Thu, 15 Dec 2005) | 2 lines

This is the first version of our example.

------------------------------------------------------------

Command finished.
```

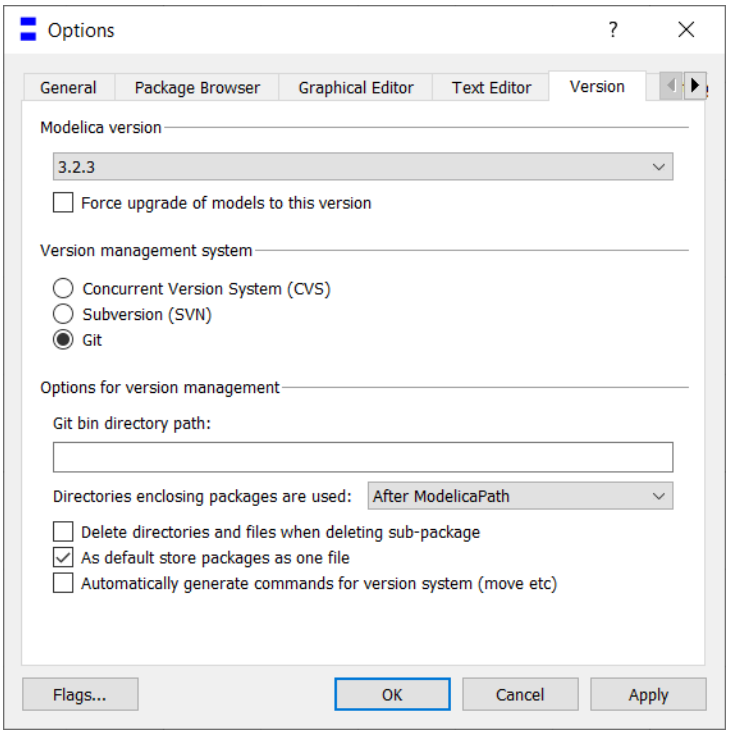
The output from both **Tools > Version > Status** and **Tools > Version > Log** contain information specific to the underlying SVN system, which is beyond the scope of this report. For non-expert users it would be beneficial to filter the raw output.

Changing an existing model

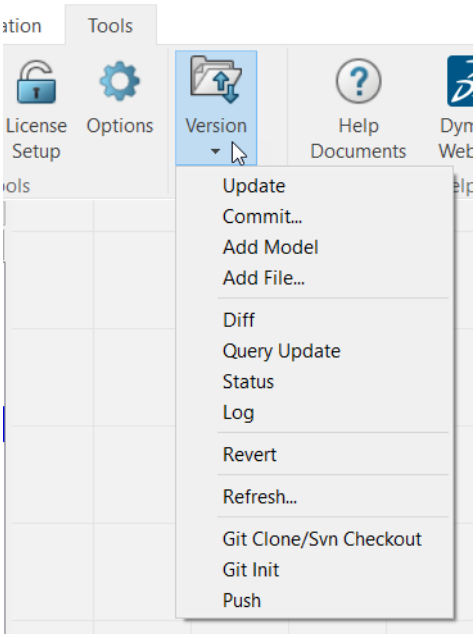
Changing the model follows the same pattern as for the CVS-based example above. The main difference is that SVN log message are different from those produced by CVS.

9.1.11 Version management using Git

Dymola 2017 and later supports Git. Git can be selected as version management system by ticking **Git** in the **Version** tab reached by the command **Tools > Options**:



The command **Tools > Version** displays the available Git commands:



9.1.12 Short guide to version management with new features included

The following is a short guide to get started with version management in Dymola; in order to keep track of changes during development.

The guide assumes that you are already familiar with the basics of version management. Note that the terminology differs between different systems.

The steps are:

1. Figure out which version management you already are using – or going to use (in the latter case we do not recommend CVS). Select this in **Tools > Options > Version**. For Git you also have to provide the path to the bin-directory. In case you use an unsupported system or do not want to use the support in Dymola go to step 3.
2. Now you need a local copy of a repository of changes. If you do not already have one:
    a. In most cases there is an existing repository for changes stored on a server (or you create one first) – use **Tools > Version > Git Clone/Svn Checkout** to connect to it. The first line is the URL – for example https://github.com/HansOlsson/GitModelicaTest.git for git or https://svn.modelica.org/projects/Modelica/trunk for svn. The second line is only needed if you want to give the local copy a different name.
    b. To experiment you can create a local repository – but you should move to a server later on. For svn follow the instructions in

https://tortoisesvn.net/docs/nightly/TortoiseSVN_en/tsvn-qs-guide.html – and use a file-url. For git just use **Tools > Version > Git Init**.

c. If you are starting a new project: plan ahead – e.g. have a directory containing your library even if there is initially only one; and do not use the version number in the path (so name your directory ”MyLibrary” not “MyLibary 1.2.0” – and use a better name than “MyLibrary”).

3. We now recommend some important changes of settings in **Tools > Options > Version**: The first three options below can also be used without using a version management system in Dymola.

   a. If you use version management for more than one related package they are normally versioned and stored together. A simple alternative is to just open one of them and rely on **Tools > Options > Version > Directories enclosing packages are used: “Before ModelicaPath”**. The default **“After ModelicaPath”** also works, except if you are developing a library already present in Dymola (such as Modelica Standard Library). Another alternative would be to add the directory containing those libraries to ModelicaPath using **Tools > Library Management > Modelica Path**; and store the setting using **Tools > Options > Settings > Modelica Path**; the benefit is that you can add entries in **File > Libraries**.

   b. Dymola normally creates backup files for deleted mo-files. With a version management system that is not needed, so: enable **Tools > Options > Version > Delete directories …**

   c. For version management it is usually preferable to have a finer granularity so that each model is stored in its own file. Disable **Tools > Options > Version > As default store packages as one file**. (Note: Some classes cannot be stored in a separate file, and this only controls the default for each package you create later on.)

   d. For version management you want to keep track of files being added, moved, and deleted automatically. Enable **Tools > Options > Version > Automatically generate commands…** This applies both to model-files and images added for equations in the documentation layer. Note: This option does not work for CVS, and must be enabled already when you add/rename/delete the class and is executed when you save your library. This also changes save to always saving entire libraries, not just individual files. If you rename a package (stored as directory) – please save before renaming any classes inside it. You can also add files manually using **Tools > Version > Add file**.

4. You can either use Dymola’s commands in **Tools > Version** or an external tool for seeing your changes.

9.1.13 References

The primary reference to the CVS version management system is

- Per Cederqvist et al. (1993): “Version Management with CVS”.

CVS binaries for several platforms and documentation (including Cederqvist et al.) are available for downloading from the official CVS homepage:

```
http://www.cvshome.org/
```

The primary source on Subversion is the homepage. The SVN command line tools used by Dymola are available here.

```
http://subversion.tigris.org/
```

Graphical user interfaces to SVN are available for downloading. Two of the more popular are TortoiseSVN (an extension to Windows Explorer)

```
http://tortoisesvn.tigris.org/
```

and RapidSVN

```
http://rapidsvn.tigris.org/
```

which is a free-standing application.

9.2 Model dependencies

Dymola can export documentation of models and packages in HTML format. The HTML documentation contains information extracted from Modelica classes. For example, model parameters and functions inputs and outputs are tabulated for easy reading without any need to understand the Modelica text.

Dymola can also make tables of cross-references in HTML. Such a table clearly shows dependencies to other packages, and in some cases incorrect references can be found. The following is an example from the package Modelica.Blocks.Examples:

These classes have been referenced in this package.

Class	Referenced From
Modelica.Blocks.Continuous.CriticalDamping	InverseModel
Modelica.Blocks.Continuous.FirstOrder	InverseModel
Modelica.Blocks.Continuous.Integrator	PID_Controller
Modelica.Blocks.Continuous.LimPID	PID_Controller
Modelica.Blocks.Examples.BusUsage_Utilities.Part	BusUsage
Modelica.Blocks.Examples.BusUsage_Utilities.Interfaces.ControlBus	BusUsage
Modelica.Blocks.Logical.And	LogicalNetwork1
Modelica.Blocks.Logical.Not	LogicalNetwork1
Modelica.Blocks.Logical.Or	LogicalNetwork1
Modelica.Blocks.Logical.Pre	LogicalNetwork1

Modelica.Blocks.Math.Feedback	InverseModel
Modelica.Blocks.Math.Gain	BusUsage
Modelica.Blocks.Math.InverseBlockConstraints	InverseModel
Modelica.Blocks.Sources.BooleanConstant	ShowLogicalSources
Modelica.Blocks.Sources.BooleanExpression	ShowLogicalSources
Modelica.Blocks.Sources.BooleanPulse	ShowLogicalSources
Modelica.Blocks.Sources.BooleanStep	BusUsage, ShowLogicalSources
Modelica.Blocks.Sources.BooleanTable	LogicalNetwork1, ShowLogicalSources
Modelica.Blocks.Sources.IntegerStep	BusUsage
Modelica.Blocks.Sources.KinematicPTP	PID_Controller
Modelica.Blocks.Sources.SampleTrigger	ShowLogicalSources
Modelica.Blocks.Sources.Sine	BusUsage, InverseModel
Modelica.Mechanics.Rotational.Components.Inertia	PID_Controller
Modelica.Mechanics.Rotational.Components.SpringDamper	PID_Controller
Modelica.Mechanics.Rotational.Sensors.SpeedSensor	PID_Controller
Modelica.Mechanics.Rotational.Sources.ConstantTorque	PID_Controller
Modelica.Mechanics.Rotational.Sources.Torque	PID_Controller
Modelica.Slunits.Angle	PID_Controller
extends Modelica.Icons.Example	BusUsage, InverseModel, LogicalNetwork1, PID_Controller, ShowLogicalSources

The column to the left shows all classes that have been referenced, for example in import statements or as the type of a component; extends clauses are specially marked. The column to the right shows the class in this package which contains some kind of reference. To see what the reference is, click on the link and view the Modelica text.

9.2.1 Cross-reference options

The generation of cross-references is controlled by options in **Tools > HTML**, the **Advanced** tab. To be able to select the **Advanced** tab, **Custom** must first be selected in the General tab.

Per file

Generate cross-references to classes in HTML documentation in each HTML-file. This is typically a sub package in a larger library.

Top level

Generate cross-references to classes in HTML documentation for top-level package. Because this often is quite large, the cross-references are stored in a separate file which is liked from the top-level HTML file (near the end).

Full name

Generate HTML cross-references to classes using full name (the default). When checking consistency of referencing to classes, it may be useful to disable this option, because inconsistent naming will show up as multiple cross-reference entries.

9.3 Encryption in Dymola

9.3.1 Introduction

There are a number of closed simulation packages on the market where you are not able to see the details of the models. Modeling is an art in the sense of describing the relevant aspects of the object under observation. It is thus very important to be able to see what assumptions and approximation the author of a model made.

Dymola is open to view all and possibly modify the details of models by showing graphical representations and, if all details are wanted, the Modelica code itself. However, Dymola also supports concealment of model details, if, for example, a supplier wants to protect proprietary information when shipping models.

A classical way of protecting software is to distribute only executable programs or object code and no source code. This approach is not useful for Modelica models. To achieve robust and efficient simulation, it is important that Dymola can make a global analysis and manipulation of all equations. It is thus highly desirable to give Dymola access to the equations in their original form. Encryption of the textual Modelica representation of the model supports concealment of internal parts such as the equations, while still allowing Dymola internally to access the equations as if the model was not encrypted.

There are also other aspects of protecting models and model libraries. Prevention of unauthorized modification of models, but still having unrestricted viewing and use is supported by including checksums. Another aspect of library protection is to ensure authorized use. In this case, any use of the library is controlled by options in a license file.

9.3.2 Visible and concealed classes

The basic idea of the protection of models is to hide some information while making it possible to use the model components.

A protected library typically consists of parts that are open, and other parts that are protected.

Protected parts may require different degree of information hiding, e.g.:

- The model is regarded as a “black box”. The icon, its connectors and parameters as well as documentation are available to the user to allow use of the model as a component, but model structure and equations are concealed.
- The model is completely concealed from external use.

Dymola supports concealment by encrypting models or libraries and the use of protected code sections, and special annotations to allow more information to be revealed. The special annotations are grouped in "Protection" group (similar to e.g. "Diagram").

There are several kinds of classes in an encrypted library starting from the most open:

- Example classes that are completely open, such that a user can duplicate it and use it as a basis for their own work. They can still refer to concealed classes.
- Classes that that can be viewed completely (including the entire Modelica text), but cannot be copied.
- Classes where the diagram is visible (but not the text).
- Classes where only the interface is visible. (This is the normal case).
- Concealed classes are completely hidden for the users, who shall not be aware of the existence of such components at all. They are not shown in the package browser and they cannot be inspected.

A class or a component is defined as concealed if one of these conditions is fulfilled:

- It is declared in the protected section of an encrypted class.
- Its lexically enclosing scope is concealed.
- It has the Protection annotation: `hideFromBrowser=true`.

Dymola supports encryption on file basis, which means that all parts of an encrypted package must be stored in one file. Storing an encrypted package in several files or in subdirectories would reveal structural information. Instead it is possible to reveal the contents of encrypted packages.

9.3.3 Developing encrypted libraries

To allow visible components to be used in the normal way to compose models, set parameters and initial values, the developer of such components must make a careful design. The public part must provide all necessary parameters, necessary control of initialization and variables to inspect and plot. Nested modifiers cannot be used to modify concealed parameters.

Instead new parameters have to be declared and propagated down the hierarchy. Parameters for initial conditions need to be introduced and propagated to start values or used in "initial sections".

The procedure for developing an encrypted library is:

- The developer maintains an unencrypted library, which is easy to modify and easy to maintain in a version control system. All parts which should be concealed in the finished library must be declared as protected or using the protection annotation.
- When the unencrypted library has been finished for release, double-click on the package to show it in Dymola.
- Select **File > Save > Encrypted File…** which produces the encrypted file myPackage.moe.

- The encrypted file, i.e., the .moe file, is distributed. The original .mo files for the encrypted parts are never distributed outside of the development group.

It is worth pointing out that external decrypting of a .moe file is not supported by Dymola, but all development work must be performed in the original unencrypted .mo file. In Dymola all encrypted files are by definition read-only.

9.3.4 Using encrypted components

Dymola must not reveal any concealed information when encrypted components are used to compose a model and as well as at simulation (unless the library developers has decided otherwise). It means that some commands or operations are disabled or have modified effects or results. Also some diagnostics and error messages must be less informative.

Let us first discuss the use of encrypted components in when building models.

File menu

An important and basic restriction is that encrypted components are read-only and cannot be modified. The **File > Save** commands **Save**, **Save All**, **Save As…**, and **Save Total…** are not available for encrypted components. The **Duplicate** command is only available if duplication is explicitly enabled.

The commands **File > Print**, **Tools > Image** and **Tools > Animation** are not changed in the meaning that they output what is visible on the screen.

The command **File > Open > Open…** reads encrypted files in the usual way, when the file type "Encrypted Modelica files (*.moe)" is selected. This file-type is visible for all users – not only the ones who can encrypt models.

Package and component browsers

Concealed classes are never shown in the package browser. The component browser does not show components or extends of a concealed class.

Editor (graphics and text)

Encrypted models are read-only and concealed models are never visible in the editor. Dymola implements the following restrictions on what is shown in the graphical and textual layers of the editor.

- The icon layer is empty for concealed classes. Also, it does not show protected connectors (regardless of encryption). Note that these rules for the icon layer also apply to icons as they are shown in the diagram layer of some other class.
- The diagram layer is as default empty for encrypted classes (not even public ones). However, models enclosed in a package called "Examples" or "Tutorial" are shown as default.
- Modelica text (declarations and equations) is as default empty if the class is encrypted.
- The documentation layer is empty for concealed classes, but is otherwise shown.

The window title says **Encrypted** instead of **Read-Only** for all encrypted classes.

Simulating an encrypted or concealed model

The aim of translating a model is to perform consistency checks and analyze and manipulate the equations to generate efficient code for simulation. This procedure is not affected by the fact that components are encrypted or concealed with the following natural modifications:

- Diagnostics and error messages during translation and simulation do as default not reveal concealed information. Warnings and error messages are issued as for non-encrypted models, but they may be less informative. An extreme is "Error in ConcealedEquation".
- The generated simulation code as default prohibits storing, plotting or other access to simulation results for concealed variables by the use of their names.

9.3.5 Examples

Encrypted transfer function

To illustrate the basics of using and encrypted model component and how encryption changes error messages, let us develop a simple encrypted model and use it in some simple contexts.

The model Modelica.Blocks.Continuous.TransferFunction defines the transfer function between a scalar input, u, and a scalar output, y. Transfer functions may be realized in different ways. Assume that we have invented a new good way to realize transfer functions and that we have developed a new model MyTransferFunction that exploits our ideas. We have also decided to protect our intellectual property by encrypting the model MyTransferFunction before making it available to others.

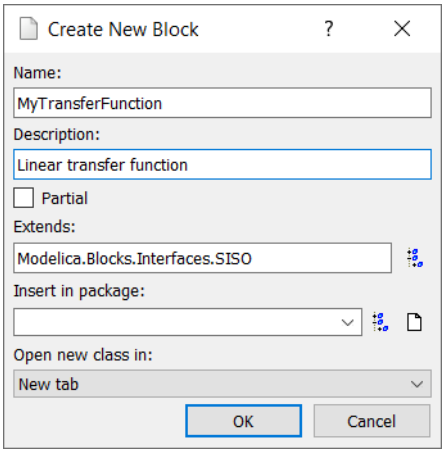
The model MyTransferFunction may look like

```
block MyTransferFunction "Linear transfer function"
  extends Modelica.Blocks.Interfaces.SISO;
  parameter Real b[:]={1} "Numerator coefficients.";
  parameter Real a[:]={1,1} "Denominator coefficients.";
protected
  Real x[size(a, 1) - 1] "State";
  parameter Integer na=size(a, 1);
  parameter Integer nb=size(b, 1);
  parameter Integer nx=size(a, 1) - 1;
  Real x1dot;
  Real xn;
equation
  [der(x); xn] = [x1dot; x];
  [u] = transpose([a])*[x1dot; x];
  [y] = transpose([zeros(na - nb, 1); b])*[x1dot; x];
end MyTransferFunction;
```

(The easiest way to create such a model for testing is:

1. Create a block using **File > New > Block**. Extend Modelica.Blocks.Interfaces.SISO. The dialog will look the following (please note that in order to encrypt this block only it has

to be top level, that is, not inserted into any package – if not the whole package should be encrypted):



2. (optionally) Copy relevant graphics from the icon layer of Modelica.Blocks.Continuous.TransferFunction.
3. Enter the code in the Modelica Text layer.
4. Use **File > Save > Encrypted File…** to encrypt the block.
5. Depending what should be tested below; either MyTransferFunction.mo (not encrypted) or MyTransferFunction.moe (encrypted) should be opened. Please remember that if encrypted models should be opened the **Files of type** must be changed in the file dialog.)

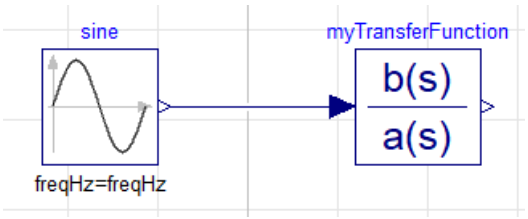
The code part is smaller than the corresponding code in the Modelica Standard Library (the initialization is simplified etc.). However, the important difference for us is that the output. The model in the Modelica Standard Library, declares the state x in the public sections as

```
output Real x[size(a, 1) - 1](start=x_start) "State of transfer
function from controller canonical form";
```

If it is possible to store the time trajectories of x, it is possible to find out how we realize the transfer function by simulating different transfer functions. In our MyTransferFunction the state is protected, to prevent users to store, plot or otherwise inspect the simulation results for the state.

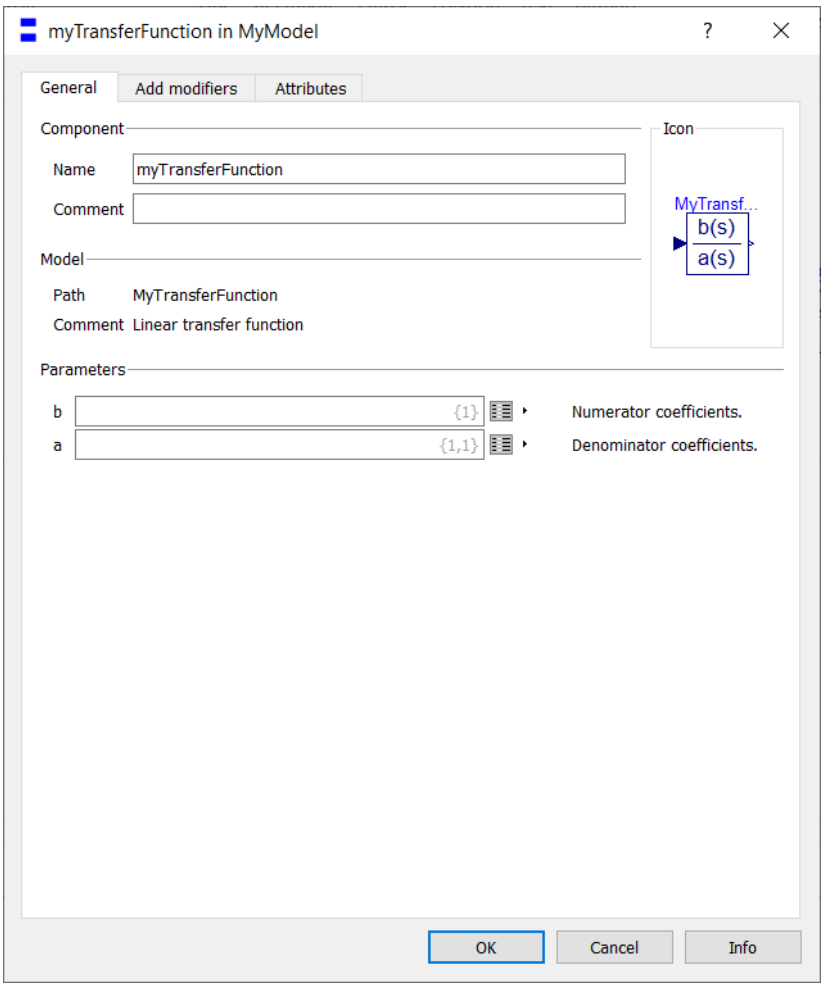
Let us test the model by creating a model MyModel (in this testing we create a model on top-level), and connecting the input to the source to the sine signal generator of the type Modelica.Block.Sources.Sine. Do not forget that the encrypted version of MyTransferFunction should be used. (You may have to unload the unencrypted model to be able to open the encrypted one.)

Example of use of encrypted model (TF).



Such a model is built in the usual way by dragging and dropping components and connecting them together. The encrypted model MyTransferFunction is available in the package browser for dragging but it cannot be displayed or inspected in the editor. The connectors are public and thus available for connection. Selecting the component and right-clicking pops the context menu in the usual way and selecting the alternative **Parameters...** displays the parameter window.

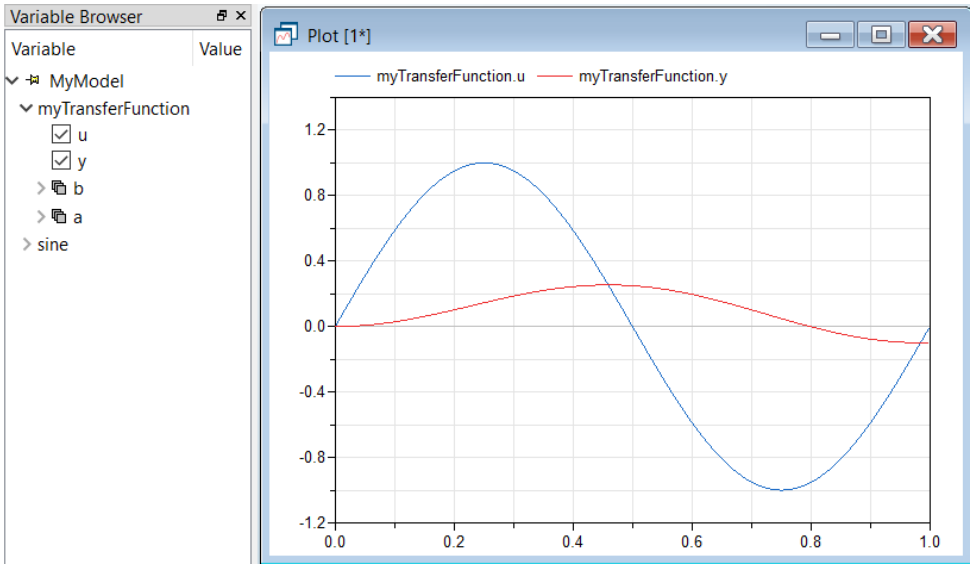
Parameters for component of encrypted model.



and it is possible to enter values for the coefficient parameters. (The parameter dialog looks the same for the encrypted and un-encrypted model – parameters protected are not shown in either case.)

The result of a simulation is shown below. Please note that the state x components are not available in the variable browser (that is the case also for the unencrypted model since the variables are protected, but for the unencrypted model also protected variables can be shown in the variable browser by the setting **Simulation > Setup > Output > (Store additional variables) Protected Variables**).

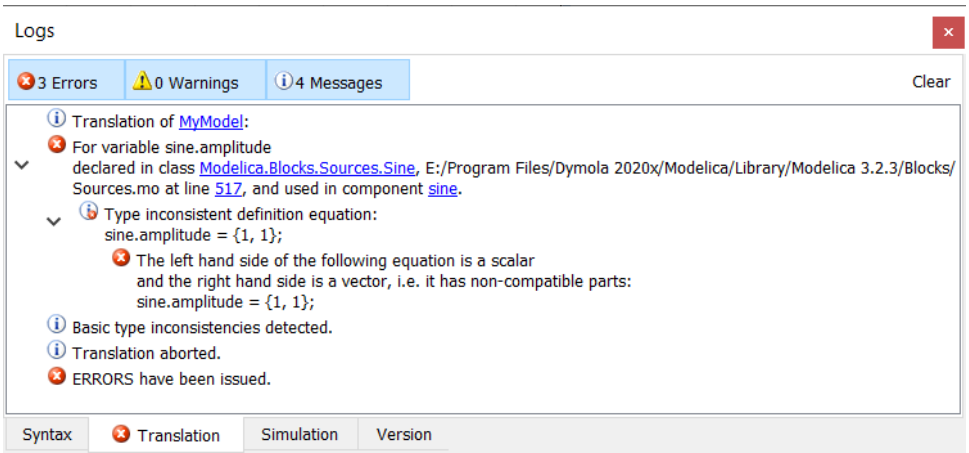
Plot for encrypted model – concealed variables are not selectable.



The sine generator may produce multiple output signals, while the transfer function assumes a scalar input. Let us see what happens if we let the sine generator produce two signals. This can be achieved by setting the value of its parameter amplitude to {1, 2}.

Translation gives the error message

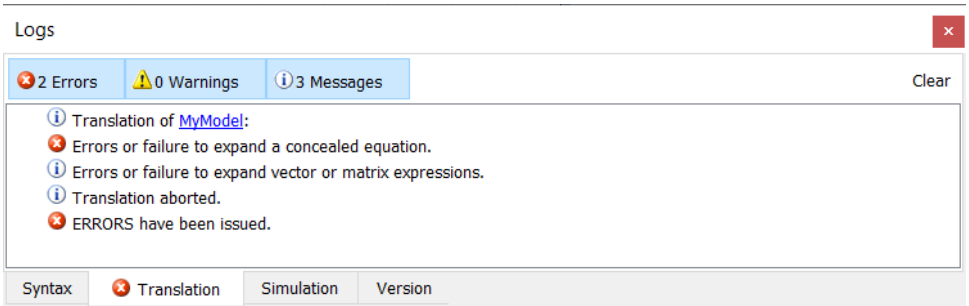
Error message containing no sensitive information.



This error message does not reveal any concealed information. In fact the same error message is given also when MytransferFunction is not encrypted.

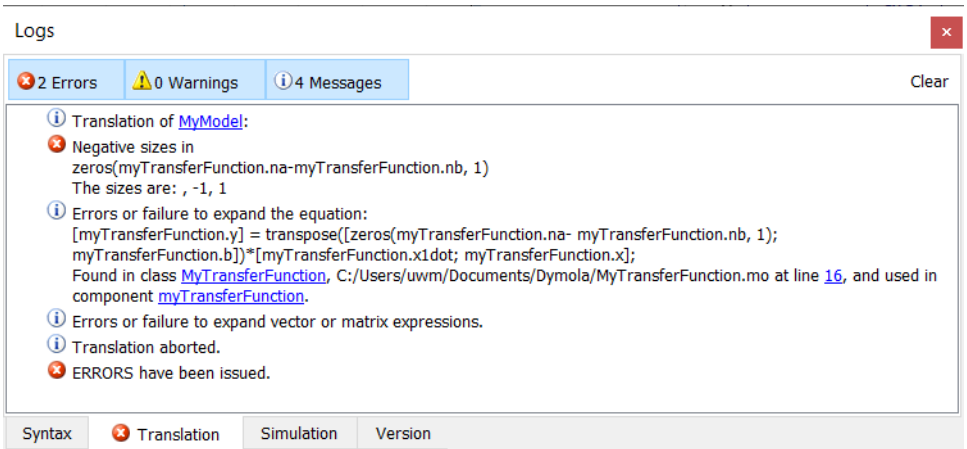
MyTransferFunction assumes that the transfer function is proper, i.e. the degree of the nominator polynomial is equal to or less than the degree of the denominator polynomial. As shown above the parameter a = {1, 1}. If we set b = {1, 1, 1} and translate, Dymola issues the error message for the encrypted block:

Error message for encrypted block.



For a non-encrypted MyTransferFunction the error message is more informative (to test this you have to unload the encrypted model and load the unencrypted one):

Corresponding error message for non-encrypted block.

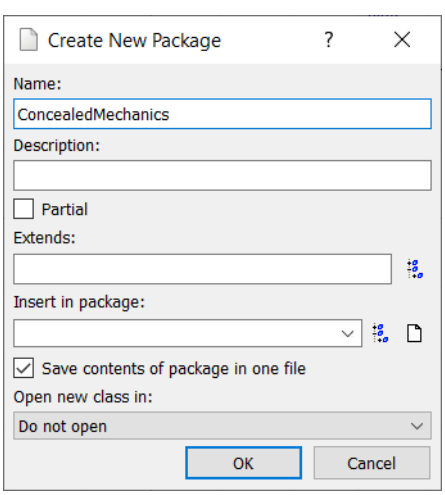


However, such an error message cannot be output for the encrypted version, because its reveals concealed information.

Coupled clutches

We will use the example Modelica.Mechanics.Rotational.Examples.CoupledClutches and exchange components to illustrate various possibilities to provide or conceal information.

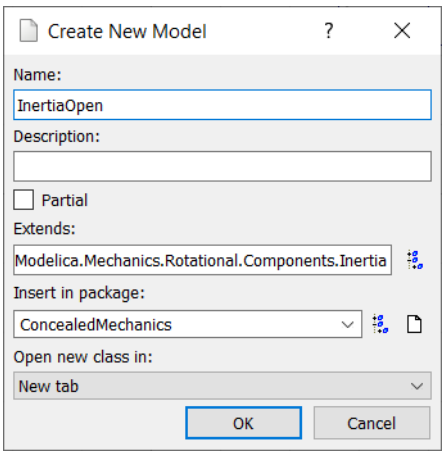
Let us make a package ConcealedMechanics where we put the components developed. We can use the command **File > New > Package**.



The package will later be encrypted.

Now, let us just make an identical copy of Modelica.Mechanics.Rotational.Components.Inertia, call it InertiaOpen.

It is most simply done right-clicking on the ConcealedMechanics package in the package browser, and then selecting **New > Model...** to insert it into ConcealedMechanics and extending from Modelica.Mechanics.Rotational.Components.Inertia. The dialog will look like:



The resulting code when looking at the Modelica Text layer will be:

```
model InertiaOpen
  extends Modelica.Mechanics.Rotational.Components.Inertia;
end InertiaOpen;
```

This model will reveal all public components (by right-clicking on the extended class in the Modelica Text layer) and select the command **Selected Class > Open Class in New Window** (you will also see some annotations etc. in the code but we don´t list those below, to have a cleaner code text)

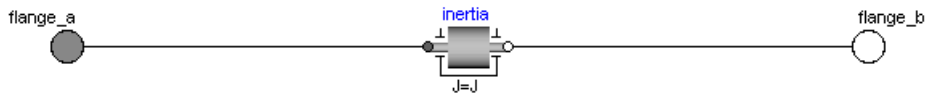
```
model Inertia "1D-rotational component with inertia"
  import SI = Modelica.SIUnits;
  parameter SI.Inertia J(min=0, start=1) "Moment of inertia";
  parameter StateSelect stateSelect=StateSelect.default
    "Priority to use phi and w as states";
  SI.Angle phi(stateSelect=stateSelect)
    "Absolute rotation angle of component";
  SI.AngularVelocity w(stateSelect=stateSelect)
    "Absolute angular velocity of component (=der(w))";
  SI.AngularAcceleration a
    "Absolute angular acceleration of component (=der(w))";
```

We could restrict this by putting phi, w and a in a protected section. If doing so, the name of the component also should be changed to e.g. InertiaProtected to reflect the change.

Another approach is to design a new interface and hide the internals of the model. In the package ConcealedMechanics, make a new model InertiaHidden extending from Modelica.Mechanics.Rotational.Interfaces.PartialTwoFlanges.

We drag in a component of class Modelica.Mechanics.Rotational.Components.Inertia and connect it.

Model to be encrypted.



To declare a parameter J we first right-click on inertia (that will pop the context menu of the component) and then select the **Parameters** alternative and set its parameter J to J. To propagate the parameter, right-click in the input field where J was entered (or click the triangle after the field) and select **Propagate…**, then click **OK** button two times.

To make the component inertia protected, we once again right-click on inertia and select the **Parameters** alternative. This time we select the **Attributes** tab and, in the Properties group, activate **Protected**. Click **OK** to validate. The resulting Modelica model is (you have to expand the "Dymola symbol" in the code to see the protected section, you will also see some annotations etc. in the code but we don´t list those below, to have a cleaner code text):

```
model InertiaHidden
  extends
Modelica.Mechanics.Rotational.Interfaces.PartialTwoFlanges;
protected
  Modelica.Mechanics.Rotational.Components.Inertia
   inertia(J=J);
public
  parameter Modelica.SIUnits.Inertia J "Moment of inertia";
equation
  connect(inertia.flange_a, flange_a);
  connect(inertia.flange_b, flange_b);
end InertiaHidden;
```

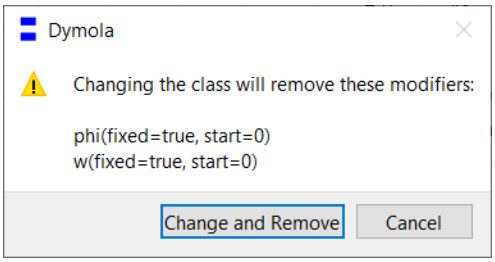
Save the package ConcealedMechanics and then encrypt it using the command **File > Save > Encrypted File...**

The demo model Modelica.Mechanics.Rotational.Examples.CoupledClutches is read-only but we can copy it. In the package browser, right-click on the demo and use the command **New > Duplicate class…**. A suitable name for the model is MyCoupledClutches. Since this is just a test, we can accept the model being a top-level model (not inserted in any package).

When changing MyCoupledClutches the encrypted version of the package ConcealedMechanics should be used; the non-encrypted package is now present in the package browser. Right-click on the package, select **Unload**, and accept it. Now the encrypted package can be opened (please remember that if encrypted models should be opened the **Files of type** must be changed in the file dialog).

Now, switch to the **Graphics** tab in Dymola to see the graphics, and right-click on J1 and select **Change Class…** Select ConcealedMechanics.InertiaOpen. In the same way, change the class of J2 to ConcealedMechanics.InertiaHidden. When doing that, an error message appears:

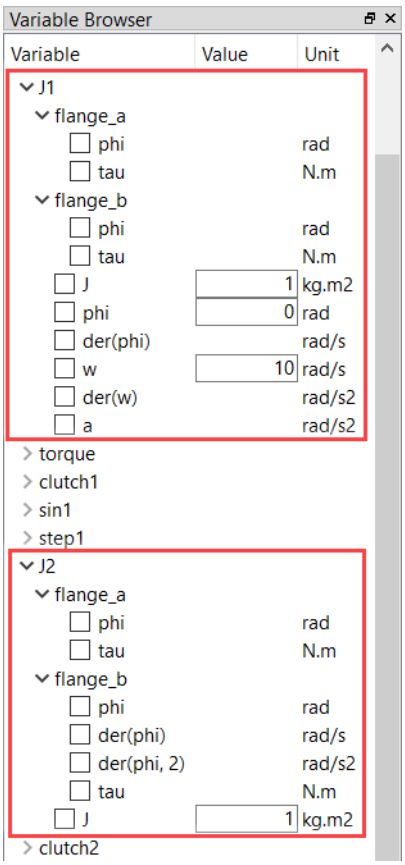
Warning message when changing class to a protected one.



The reason for this warning message is that since now a number of variables are not public anymore; it is not possible to e.g. select start values of them. Use **Change and remove** to accept this.

Now the simulation of the model CoupledClutches gives the following variable browser:

Concealed variables: J1 is public (non-encrypted). J2 is concealed (encrypted). This gives different plot possibilities.



For J1and J2 it is possible to plot the connector variables and set the moment of inertia J.

However, for J1 it is also possible to plot velocity and acceleration. What to do if we would like to plot the velocity of J2? The velocity can be made available by connecting a Speed-Sensor.

For J1 it is possible to set an initial value for w. For J2 the situation is more complex. By just looking at it we cannot tell whether there is some internal initialization. When translating the model Dymola issues a warning that initial conditions are not fully specified. The documentation of InertiaHidden needs thus to include documentation on initial conditions. In this case we know that there are no initial conditions stated for J2, so we may introduce an initial equation section in the CoupledClutches model containing for example

```
initial equation
  J2.flange_a.phi = ... "start angle";
  der(J2.flange_a.phi) = ... "start velocity";
```

to specify the initial position and velocity of J2.

9.3.6 Special annotations for concealment

General

Annotations for concealment were present in Dymola before the Modelica standard annotations were introduced. The annotations now co-exist, but the standard annotations should be preferred if creating a new library.

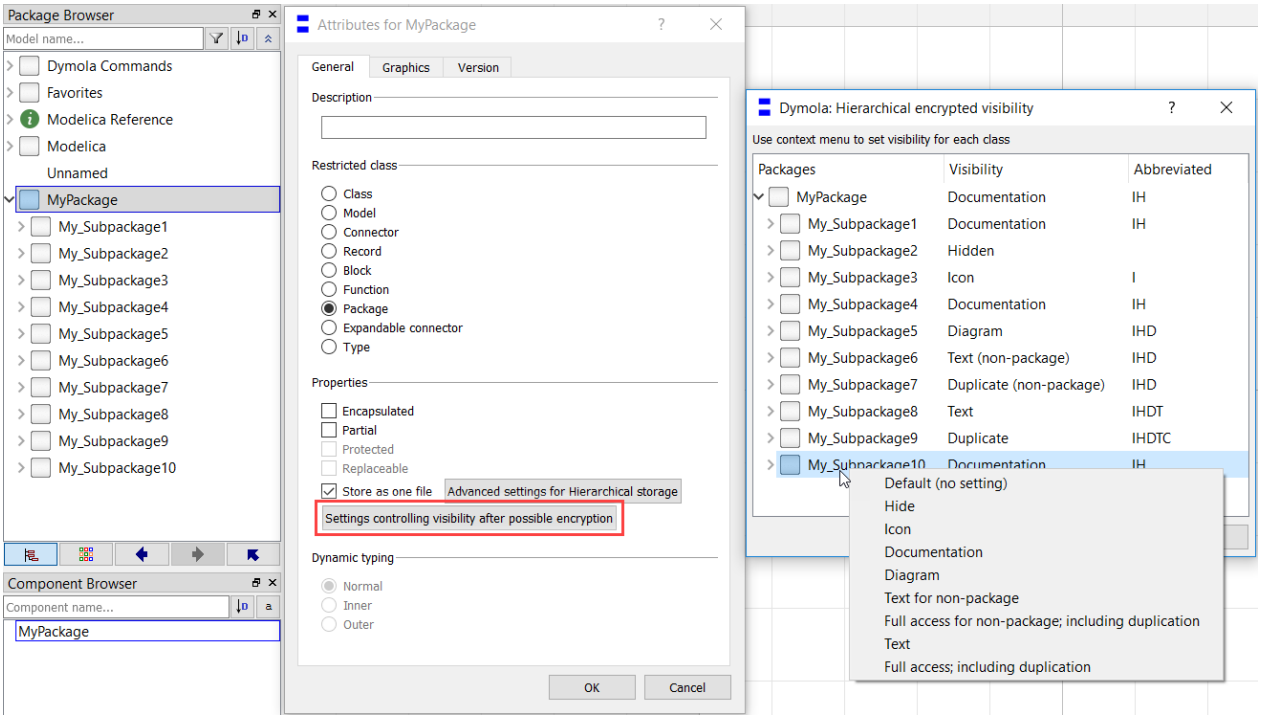
Important

It is important however to note that you cannot use the Dymola annotation `__Dymola_Protection()` together with the general Modelica annotation `Protection()` in the same file; the solution in such case is to embed the wanted Dymola annotation components in the Modelica one. For an example, see section "Licensing libraries" starting on page 1210, the last part of the "Introduction" section.

Working with Modelica standard annotations for concealment

The Modelica standard annotations for concealment are presented in the Modelica Language Specification; for Modelica 3.4 see chapter 18.9 "Annotations for Access Control to Protect Intellectual Property". The Modelica Language Specification can be found by the command **Tools > Help Documents**. (It can also be downloaded from https://Modelica.org/documents.) The description below are comments to this documentation, and the descripting of the Dymola GUI for handling these annotations.

It is possible to manage the protection level for a library hierarchy using a graphical user interface when adding the Modelica standard annotations for concealment. An example:



The starting point in the example above is a package `MyPackage` with 10 sub-packages. Right-clicking `MyPackage` in the package browser and selecting **Attributes** from the context menu gives the dialog to the right of the package browser above (in the middle of the figure).

Clicking, in this dialog, the button **Settings controlling visibility after possible encryption** gives the browser to the right in the image above. Each subpackage in this example has been given a different protection level by right-clicking it and selecting from the context menu, as shown for the last package.

The selected level will be implemented in the subpackage as an annotation, for example, for the subpackage `My_Subpackage5` in the figure above, the annotation will be

```
Annotation(Protection(access=Access.diagram))
```

Notes:

- The alternative **Default (no setting)** is typically what you select for a certain class that don´t need any additional protection setting; the protection level set by the enclosing class is sufficient. The default is the level **Documentation**. However, the corresponding annotation will not be inserted in the Modelica text; "(no setting)".
- Be careful with the **Hide** alternative; a class with this protection level is by default not seen in the package browser even before encryption. To see it, you must activate the setting **Show protected classes** in the simulation setup. You reach this setting by the command **Simulation > Options**, the **Package Browser** tab.

- The generation of HTML documentation is normally performed before encryption. Note that the correct protection levels must be set before generation HTML documentation, to not reveal too much information in the documentation.

The selected protection levels are applied when encrypting the package, for example, with the command **File > Save > Encrypted File…**.

Working with Dymola-specific annotations for concealment

Note that the Dymola-specific annotations do not have any GUI for working with them, you have to work in the Modelica text editor.

These specific Dymola-annotations are all grouped inside:

```
annotation(__Dymola_Protection(…));
```

The annotations are designed based on the following basic principles:

- Security by default – the default is to not reveal information for encrypted packages. You as the library developer have to enable each of these flags.
- It is more important to protect an entire package than individual classes in the package. The annotation "nested" is used for this purpose, setting e.g. `nestedAllowDuplicate = true` in a certain package means that a single non-package class in that package can be duplicated but not the whole package. This is useful when e.g. a large amount of examples should have a certain extent of protection, although a specific example should be possible to look at.
- Easy-to-use.
- Simple logic to make it easy to verify the behavior. Thus you can enable duplication, but hide diagrams even though this does not make sense.
- Only applied after encryption – thus they can be present in the original library.

The behavior can be summarized in the following table (the missing entries are not implemented):

Annotation for making encrypted classes visible.

Show/allow	Annotation		Default
	All classes	Non-packages	
Duplicate (1)	allowDuplicate	nestedAllowDuplicate	false
Diagram	showDiagram	nestedShowDiagram	false (2)
Text (3)	showText	nestedShowText	false
Icon			true
Documentation (4)	showDocumentation	nestedShowDocumentation	true
In package browser/ choices all matching (4,5)	hideFromBrowser	N/A	false
License check (6)	Library		None

The annotation applies hierarchically to all classes (unless overridden by a similar annotation).

The idea is to use either an all-classes or a non-packages annotation for a certain package, they should be considered as alternatives depending on the functionality wanted. **Note** that if single class(es) in a package should be protected, the top-level package itself should be annotated using non-package annotation, e. g. `nestedShowText=true` to allow all non-package text to be visible unless overridden locally. If using All classes annotation to annotate the top-level package, e. g. `showText=true`, the classes inside it cannot be protected, they will be visible by inspecting/duplicating the top-level package.

Notes:

1. **Please note** that any class being allowed to duplicate is also not protected anymore; any protection annotations in that class is useless since the source code is revealed anyway when looking at the copy of the class.
2. However, the default for Examples and Tutorial-packages is `showDiagram=true`.
3. The text window has copying disabled (unless duplicate is allowed), but there are ways of circumventing this.
4. If a class is hidden from browser or not shown in documentation layer it is treated as a protected class for HTML-generation. It is thus not possible to export to HTML unless a specific setting is applied (that is valid also before encryption).
5. The logic for the browser is reversed ("hide" contrary to the others "show").
6. License check of libraries: see next section.

In addition there are several top level settings (also inside __Dymola_Protection) as follows:

Show/allow	**Annotation**	**Default**
Plotting of protected variables	showVariables	false
Diagnostics with variable	showDiagnostics	false
Statistics (e.g. #states)	showStatistics	false
Flat-modelica	showFlat	false

Note that if several encrypted packages are used they must all enable e.g. statistics for the statistics to be shown.

9.3.7 Licensing libraries

Introduction

A licensing mechanism is available for "external" Modelica libraries, i.e. libraries which are not sold and licensed through the Dassault Systèmes channels.

Note that below describes third party licensing for node-locked licenses. Licensing of third party libraries is also supported for sharable licenses, however currently only on Windows. Third party library vendors, please contact your support channel/sales representative for additional information, and examples.

The licensing mechanism provides a number of functions:

- The library developer can create/administrate licenses without the help of tool vendor.
- It is possible to update the licensing information without updating the library, and to update the library without changing licensing information.
- It is possible to tie the library license to specified users, identified by license number or computer host-id. For sharable licenses, see note above.
- Start and expiration dates on licenses can be defined.

The licensing scheme is implemented on top of the encryption mechanism in Dymola that prevents the user from inspecting all source code. This is how the licensing mechanism works:

- In the library that will be protected, the developer adds an annotation with an arbitrary key. This key is used to associate the library with a separate authorization file, and will not be visible to the user after library encryption.
- The developer then makes a separate authorization Modelica file which contains the library key, some identification of the user that should be able to read the library (the licensed systems), and possibly start and expiration dates. This file is also encrypted in Dymola and distributed as needed. **Note**:
    - The encrypted authorization file must be located in the same folder as the encrypted corresponding library.
    - Dymola allows extension `.moe` or `.mo_lic` for the encrypted authorization file, 3DEXPERIENCE Dymola Behavior Modeling app *only* allows extension `.mo_lic` for the encrypted authorization file.
- When Dymola opens the library and sees the key, it also opens the corresponding authorization file, and checks if the user is permitted to open the library. The authorization file is never shown to the user, only used internally by the tool.

The full details of the licensing mechanism can be found in "Modelica Language Specification, Version 3.4", section 18.9.2 "Licensing". The document can be accessed by the command **Tools > Help Documents**. (It can also be downloaded from https://Modelica.org/documents.)

Some comments to the detailed description above:

- The content of the library key string is unspecified, but must match the key of the library.
- The license id currently supported is the Dymola license number or the computer host-id (as shown by Dymola using the command **Tools > License Setup**, the **Details** tab). A license id (e.g. 1234) can be specified as "`1234`" as well as "`lic:1234`", a host id (e.g. 0019d2c9bfe7) can be specified as "`0019d2c9bfe7`" as well as "`mac:0019d2c9bfe7`".
- In some cases there are multiple host id´s (docking stations etc.). In such case you should authorize all such host id's.

Note that a `hideFromBrowser` annotation is advised to prevent the authorization file from being shown in the Dymola package browser. See example below.

Important

It is important however to note that you cannot use the Dymola annotation `__Dymola_Protection()` together with the general Modelica annotation `Protection()` in the same file; the solution is to embed the wanted Dymola annotation components in the Modelica one, for example

```
annotation(Protection(License=…., Access=…., __Dymola_showFlat=true,
__Dymola_showVariable=true, …));
```

Example (before encryption)

The authorization file and the library are created independently in Dymola, at top level. The command **File > Save > Encrypted File…** is then used to create encrypted versions of the authorization file and the library. When used, they must be located in the same folder in order for the library to find the authorization file.

The library contains these annotations to specify the library key and the authorization file.

```
// File MyLibrary\package.mo
// (library file before encryption)
package MyLibrary
…
  annotation(Protection(License(
      libraryKey="15783-A39323-498222-444ckk4ll",licenseFile=
      "MyLibraryAuthorization_Dymola.moe")));
end MyLibrary;
```

The authorization file contains annotations that allow execution on three different licensed systems. (The first one uses Dymola license number, the second also has an expiration date and the third uses mac address/host id.):

```
// File MyLibraryAuthorization_Dymola.mo
// (authorization file before encryption)
package MyLibraryAuthorization_Dymola
  annotation(
    __Dymola_Protection(hideFromBrowser=true),
    Authorization(
      libraryKey="15783-A39323-498222-444ckk4ll",
      licensor="Organization A\nRoad, Country",
      license={License(licensee="Organization B, Mr. X",
        id={"lic:1269"}),
      License(licensee="Organization C, Mr. Y",
        id={"lic:511"}, expirationDate="2011-06-30"),
      License(licensee="Organization D, Mr. Z",
        id={"mac:0019d2c9bfe7","mac:0026c6b26950"})
  }));
end MyLibraryAuthorization_Dymola;
```

9.3.8 Scrambling in Dymola

Encryption of a package/model is a useful way of making a package useable without revealing information. However, in certain scenarios it is not the ideal choice when sending one (or a few) component models that shall only be used directly.

In such cases the most important information to conceal is data and internal structure, and there is no need to keep "replaceable" components or classes.

The ideal choice would in that case be to send something that:

- Does not contain internal structure and original data.
- Automatically hides all internal components.
- Can be used as any other model in Dymola (including differentiation for state-selection).
- Allows you to see exactly what is sent.

This is accomplished using **File > Save > Encrypted total model…** and can be done either on a model/block or for a package, where each public non-partial model/block is scrambled individually and then placed together in a package.

Each individual model is scrambled as explained in the next to remove unnecessary information and the resulting file is then encrypted as an additional safety precaution.

Note that the protection annotation for license check is preserved during scrambling, i.e. you can specify this in the model/package before scrambling. However, license checks from enclosing packages are not copied, i.e. if you want to protect a model you should add the annotation on the particular model.

Example of scrambling

We continue with the inertia example, but now rewrite the Inertia model by replacing the parameter 'J' by two variables 'r' and 'm' and computing the inertia based on these.

Create the model below as a top-level model (to be able to encrypt it) and extend Modelica.Mechanics.Rotational.Interfaces.PartialTwoFlanges.

```
model InertiaAlternative
  annotation (uses(Modelica(version="3.0")),
Documentation(info="<html> An inertia of a certain shape with
settable radius.
<html>"));
  extends Modelica.Mechanics.Rotational.Interfaces.
    PartialTwoFlanges;
  parameter Modelica.SIunits.Length r=1 "Radius";
protected
  constant Modelica.SIunits.Mass m=0.5 "Mass";
  Modelica.SIunits.Angle phi "Absolute rotation angle of
    component (= flange_a.phi = flange_b.phi)";
  Modelica.SIunits.AngularVelocity w "Angular velocity";
equation
  flange_a.phi=phi;
  flange_b.phi=phi;
  w = der(phi);
  m*r^2/12*der(w) = flange_a.tau + flange_b.tau;
end InertiaAlternative;
```

The mass and the shape should be hidden from the user of the model. By selecting **File > Save > Encrypted total model…** the model is first scrambled and then encrypted.

The procedure gives the messages:

```
Will encrypt to file C:/MyExperiment/InertiaAlternative.moe.
First scrambling to file
C:/MyExperiment/InertiaAlternative.tmp.moe
Scrambling InertiaAlternative.
The scrambling should preserve the following top-level
variables:
  connector flange_a
  connector flange_b
  parameter r
Scrambling complete, verifying it.
Encrypting.
Encryption complete, file can be found in
C:/MyExperiment/InertiaAlternative.moe.
```

The scrambling indicate which variables should be kept, and include a tag before the variable to explain why.

Users can examine the Inertia3.tmp.mo file to verify that the no vital information is present (if the file is to be loaded in Dymola, do not forget to first close the present file with the same name by right-clicking on it in the package browser and select **Unload**, otherwise there will be a name conflict):

```
model InertiaAlternative
encapsulated connector r0
Real phi(unit = "rad", displayUnit = "deg") "Absolute rotation
angle of flange";
flow Real tau(unit = "N.m") "Cut torque in the flange";
 annotation(Hide=true,
Icon(coordinateSystem(preserveAspectRatio=true, extent={{-100,-
100},{100,100}}, grid={2,2}),
             graphics={Ellipse(
               extent={{-100,100},{100,-100}},
               lineColor={0,0,0},
               fillColor={95,95,95},
               fillPattern=FillPattern.Solid)}));
  end r0;
r0 flange_a annotation (Placement(transformation(extent={{-
110,-10},{-90,10}}, rotation=0)));
encapsulated connector r1
Real phi(unit = "rad", displayUnit = "deg") "Absolute rotation
angle of flange";
flow Real tau(unit = "N.m") "Cut torque in the flange";
 annotation(Hide=true,
Icon(coordinateSystem(preserveAspectRatio=true, extent={{-100,-
100},{100,100}}, grid={2,2}),
             graphics={Ellipse(
               extent={{-98,100},{102,-100}},
               lineColor={0,0,0}, fillColor={255,255,255},
               fillPattern=FillPattern.Solid)}));
  end r1;
r1 flange_b annotation (Placement(transformation(extent={{90,-
10},{110,10}}, rotation=0)));
parameter Real r(unit = "m") = 1 "Radius";
```

```
protected
Real z2;
Real z1;
 annotation(Settings(NewStateSelection=true),
Documentation(info="<html>
An inertia of a certain shape with settable radius.
</html>"));
  protected equation
flange_a.phi = z2;
flange_b.phi = z2;
z1 = der(z2);
0.0416666666666667*r^2*der(z1) = flange_a.tau+flange_b.tau;
end InertiaAlternative;
```

As can be seen the mass and shape have been constant-evaluated making it impossible to determine their individual values. In addition the names of all internal variables are replaced by scrambled names (if the variable is preserved at all).

The encrypted file only contains this information, but is in addition encrypted. Encryption prevents disclosure of even the scrambled information and also makes the model read-only.

9.4 Model and library checking

9.4.1 Overview

This section is an overview of the functionality in the `Check` package.

Regression testing

Sometimes, model code modification can cause unintended changes in the model behavior. To catch this type of error regression testing can be used. By simulating and comparing the state variables of a model against a reference data file, changes in the behavior can be detected. With this new feature, a test suite can easily be constructed and the process of regression testing can be automated.

Class coverage

To determine how well the test suite for regression testing covers (uses) the classes in the library under test, class coverage analysis is used.

Condition coverage

For conditional expressions and Boolean variables, condition coverage analysis can be performed to investigate which branches of conditional equations in the test cases that are ever executed. This can be used to detect “untested code”.

Style checking

For model development, style checking is introduced to assure that models and libraries are well documented etc. Style checking is available directly in the `ModelManagement.Check` package, but is also available as the command **Check > Style**, available in the **Graphics** tab and the **Text** tab.

Translation statistics

The translation statistics option is an extension to the normal regression testing of state variables to make testing more powerful and to catch differences that could otherwise be hard to see. Using this option, the statistics of the translated model can be included in the regression testing to detect changes in, for example; the number and sizes of linear and nonlinear system of equations and the number of state variables.

9.4.2 Regression testing

It will be shown how to use **checkLibrary** to set up a test suite for regression testing of a library, and how to use the checking utilities included in the package.

Regression testing is the utility for library and model developers to assist with regression testing of libraries. Regression testing is performed to detect unintended changes in the behavior of models due to updates and rewriting of model code.

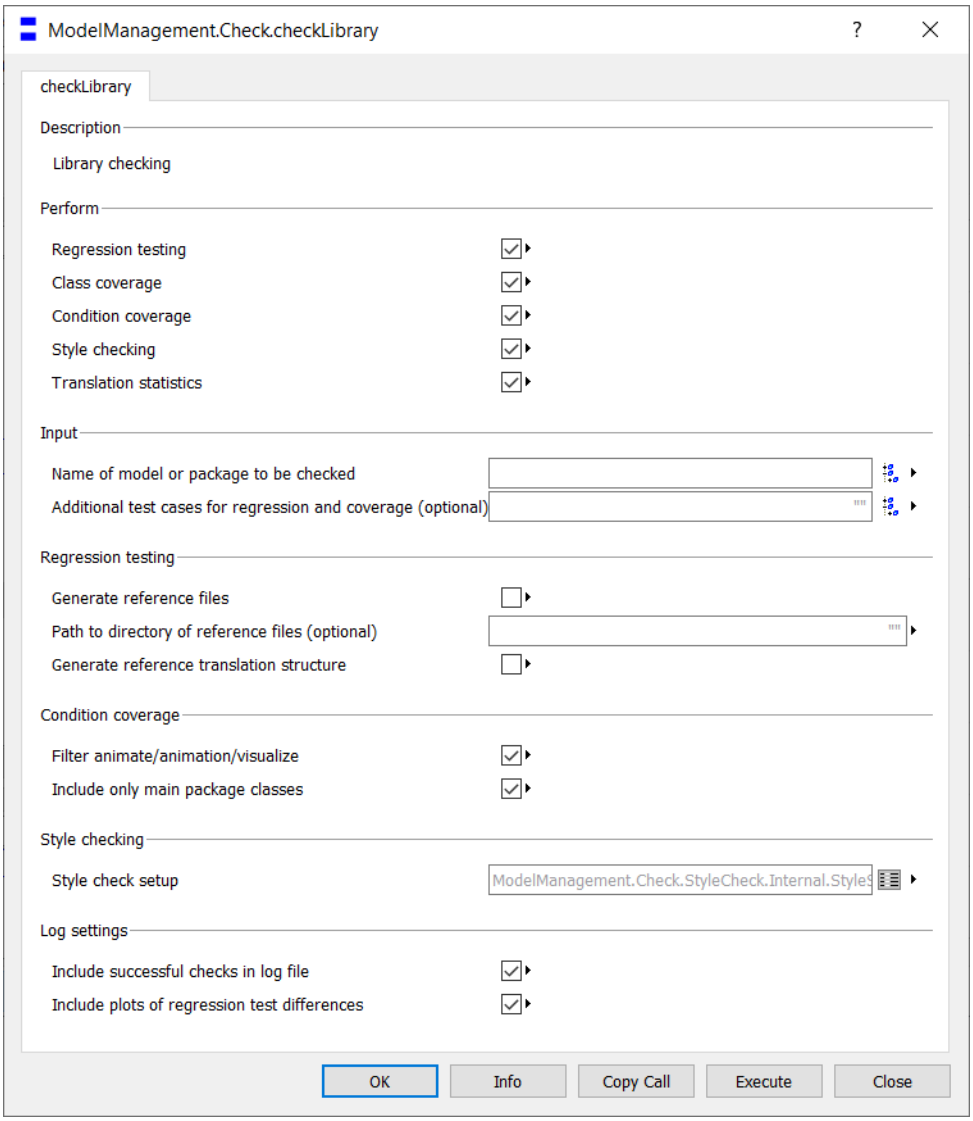
The package collects all test cases specified by the user, i.e. the models in the library that have the annotation `experiment(StopTime=…)` set and additional test cases from a test package (if available). After simulating a test case, the results are compared to the reference file. The package takes as comparison signals the possible continuous and discrete state variables of the model.

Setting up reference files

We will start by setting up reference files for the regression testing.

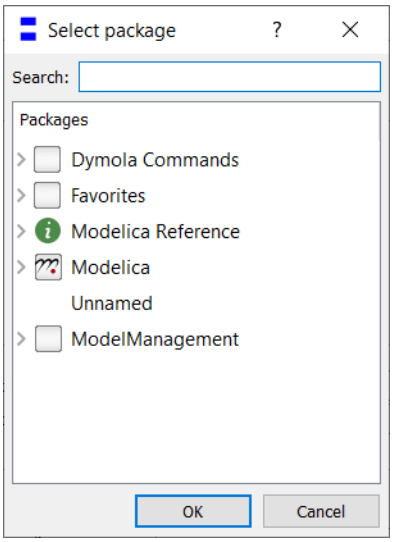
- Right-click on the function **checkLibrary** in the package Check and select **Call function...** in the context menu. Please keep the default settings with exception of the ones that are changed below.

Dialog of checkLibrary.



- Select the library/package or model you want to check by using the browser for **Name of model or package to be checked** in the menu above.

Package selector.



- Check the boxes for **Generate reference files** and **Generate reference translation structure**.
- Use the browser to select additional test cases by using the entry **Additional test cases for regression and coverage (optional)**, if there are any.
- Optionally, specify the path a directory where the reference files should be stored by entering the complete path under **Path to directory of reference files (optional)**. If not specified, the files will be stored in `workdir\ReferenceData\`*`Libraryname`*.
- Execute by clicking **OK** or **Execute**.

The package will now generate a set of reference files for the selected library and optional test suite.

In some cases one would like to expand the test suite by adding additional reference files without re-generating the entire test suite reference. This can easily be done by selecting that particular model as the package to generate reference files for, and then specify the old directory of reference files under **Regression testing**.

The new reference file will now end up in the same folder as the old files and the new test case can be included in the test package. The same procedure can be used to replace specific reference files if intended changes in the behavior of a model require a new reference file.

Performing regression testing

In order to perform regression testing, right-click on **checkLibrary** function to pop a dialog box with the default settings for checking and regression testing. By default, all checks are active.

As before, when generating reference files, choose your library, additional test cases (if any) and specify a path to your directory of reference files, if you are not using the default directory.

Execute the test by clicking **OK** or **Execute**.

The testing will now start and an html log file will be generated in your working directory. A message in the commands window will tell the name and path of the log file generated.

Output

As an example of the output generated from regression testing consider a package `MySimplePackage` containing a simple model `Test`,

```
package MySimplePackage
  model Test
  annotation (experiment(StopTime=1));
  Real x(start=1);
  parameter Real a = 10;
  equation
  der(x)= -a*x;
  end Test;
end MySimplePackage;
```

Note that the annotation experiment(StopTime=1) have been set to define this model as a test case.

Start by generating a reference file for the package as described previously. (Since this simple example covers only regression testing of a package with one model, please uncheck all items in the group **Perform** except **Regression testing** and **Translation statistics**.) Click **OK**. (By selecting **Execute** the dialog window will be kept displayed after performing the call. Note that in some cases the dialog window seems to disappear; but it is only put behind the Dymola main window, you can pause over the Dymola symbol in the taskbar to see that you have it still available by clicking on it there.)

A log named `MySimplePackage_ReferenceLog` will be generated in the working directory:

Log of reference file generation.

Referenece files

Reference files generated for **MySimplePackage**

Path to reference files: **C:/Users/uwm/Documents/Dymola/ReferenceData/MySimplePackage**

TASKS

- Reference file generation

Statistics

REFERENCE FILES

- MySimplePackage.Test: Reference file generated. Reference structure file generated.

Top of page

STATISTICS

Reference files generated

Top of page Statistics

Then perform a regression test by right-clicking the checkLibray function again. Select the package, uncheck everything in the **Perform** group except **Regression testing** and **Translation statistics** and click **OK**. A log named `MySimplePackage_HTML_log` will be generated in the working directory:

Log of successful test.

Library check log

TASKS

- Regression testing
- Model structure testing

Statistics

REGRESSION TEST RESULTS

- **MySimplePackage.Test**: Validation ok. Structural validation ok. Translation time validation ok.

Top of page

STATISTICS

2 tests performed on 1 test cases.
All tests ok, validation of MySimplePackage, **successful**.

Top of page Statistics

The regression tests (regression and model structure) are successful and this is expected since there should be no differences in the package since nothing has changed in the model code.

Changing the value of the parameter **a** to a=15 causes the regression testing to fail. The results can be seen below.

Log of failed test.

Library check log

TASKS

- Regression testing
- Model structure testing

Statistics

REGRESSION TEST RESULTS

- **MySimplePackage.Test**: Validation failed. Structural validation ok. Translation time validation ok.

Top of page

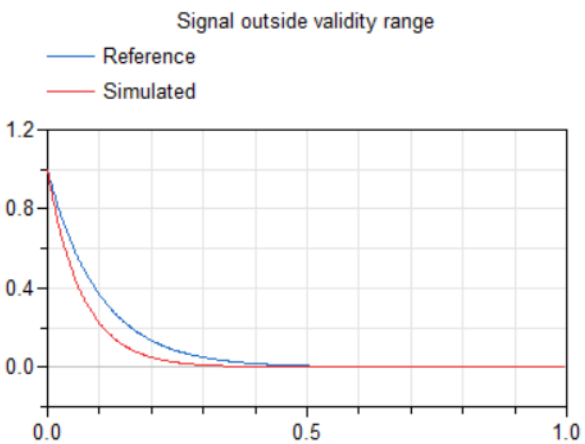
STATISTICS

2 tests performed on 1 test cases.
1 of 2 tests failed. Validation of MySimplePackage, **not successful**.

Top of page Statistics

Clicking the link for the validation failed shows the specific log for this test case. A plot is generated to show the difference in the state variable (compared to the reference).

Failed validation plot.



9.4.3 Class coverage

The class coverage analysis is performed by default to show how well the test cases cover the classes in a library. A class is considered covered if it is used in a test case, or if a class used in a test case extends from it. The analysis is performed for **models**, **blocks** and **connectors**, (**type**, **record**, **function** and **class** are considered covered by default).

The result of the class coverage analysis is reported in the log file under Class coverage.

There are two ways to present the results:

- List all classes and how many times they are used.
- Only list classes that are never used.

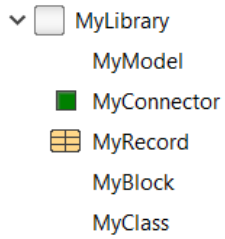
The output is by default the first option and it can be changed by unchecking the checkbox in the dialog box for **checkLibrary** under **Log settings**.

NOTE: The class coverage does not give any result for encrypted libraries.

Output

As an example, consider the following library, `MyLibrary`, containing 5 classes, depicted below. The library is available under `ModelManagement.Check.Examples`.

Library example.



The library contains 5 classes, 3 of which are considered in the class coverage analysis. The classes that are not included in the analysis are `MyRecord` and `MyClass` since only blocks, models and connectors are considered as explained above.

Since the package ModelManagement is encrypted, please duplicate the package MyLibrary to be able to run class coverage test on it. (Right-click on MyLibrary, select **New > Duplicate Class…**. Click **OK.** Finally save the library; now it can be tested.)

Running regression testing with the option class coverage on the library copy will generate the following section in the log `MyLibrary_HTML_log` (To test this without having to create a reference file, right-click the checkLibray function again, Select MyLibrary, uncheck everything in the **Perform** group except **Class coverage** and click **OK**.)

Class coverage log.

CLASS COVERAGE

1 of 3 classes are covered by the test suite. 2 classes are not covered.
Of the not covered classes

- 1 is a block
- 1 is a connector

Listing classes in `MyLibrary`

- model **MyModel**, used 1 times.
- connector **MyConnector**, not used.
- block **MyBlock**, not used.

The log indicates that the only class that was used in the test suite is `MyModel`. This is correct since `MyLibrary` only contains one test case (`MyModel` is the only model in `MyLibrary` with experiment.StopTime set). To get complete class coverage one could define a test case, or multiple ones, that uses the classes that are not covered. This can be done either by creating new models in the library or by creating a new TestLibrary containing only test cases. The TestLibrary can then be coupled to the regression as Additional test cases.

9.4.4 Condition coverage

Condition coverage analysis is performed to ensure that all conditional parts in model equations are tested. A global merge is performed for the entire test suite so that a condition is considered to be covered if it has been both **true** and **false** in a test case or if it has been **true** in one test case and **false** in another test case in the test suite.

Condition coverage analysis considers:

- Boolean variables
- if conditions in all equations and expressions

NOTE: For encrypted libraries the condition coverage analysis may neglect classes depending on the level of encryption.

Output

Consider a model, `MyModel`, containing a Boolean variable, **low**,

```
low = x < min;
```

Running regression testing with the option **Condition coverage** on this model (MyLibrary.MyModel) will generate the following section in the log file. (Please work with the copy of the library MyLibrary that was made in the previous section to avoid working with the encrypted MyModel. To test without reference files, only uncheck everything in the **Perform** group except **Condition coverage** when running the checkLibrary function.)

Condition coverage log.

CONDITION COVERAGE

- **MyLibrary.MyModel**
  - `low` is always **false**

This indicates that there is a parameter or variable, `low`, which always has the value **false**.

Acknowledgement

The Check package is based on earlier implementations used for several years for testing libraries and Dymola itself.

The condition coverage feature is inspired by Mike Tiller's work described in

Tiller, Michael M. and Burit Kittirungsi: "UnitTesting: A Library for Modelica Unit Testing", Proc. 5th International Modelica Conference 2006, pp. 695—704, Vienna, Austria. See http://www.modelica.org.

9.4.5 Style checking

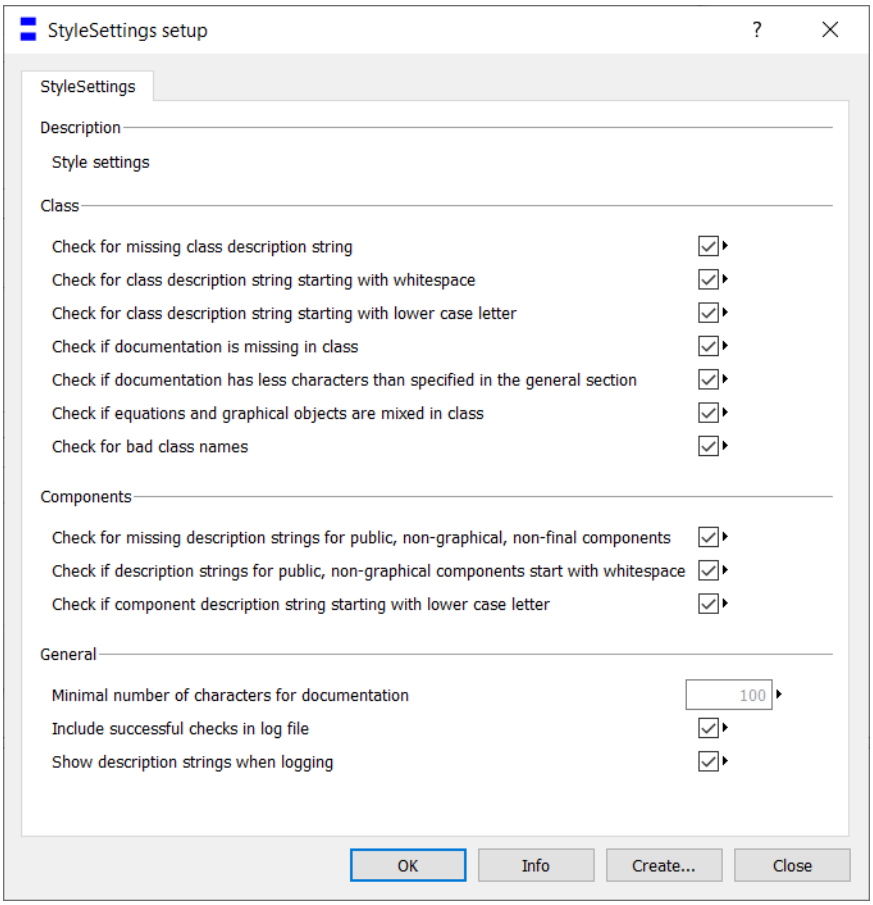
Style Checking is a utility for checking of libraries and classes according to the guidelines in the style guide. The intention is to assist a model developer with quality control of the library and to warn when issues regarding the specified quality constraints are not fulfilled. Note that style checking is also available as the command **Check > Style** in the **Grapics** tab and **Text** tab, see section "Style checking as the Check > Style command" on page 1228.

Settings

To change the default settings for library checking, right-click the checkLibrary function, and under the group **Style checking**, click the **Edit** icon after the **Style check setup** box (or click the little triangle).

It is then possible to edit the setup according to your own requirements. The checks are divided in to three categories; **Class checks**, **Component checks** and **General**. Below is a screenshot of the available settings in the **setup** menu.

Stylecheck setup dialog.



The **styleCheck** can be customized by selecting appropriate checks in the checkboxes.

The rules for the option **Check for bad class names** are:

- Class name shall start with capital letter except for functions that shall start with lower case letter.
- Class name shall not contain "_". Use names like "IdealDiode" instead of "Ideal_diode"

Documentation and description strings are not required for `type`.

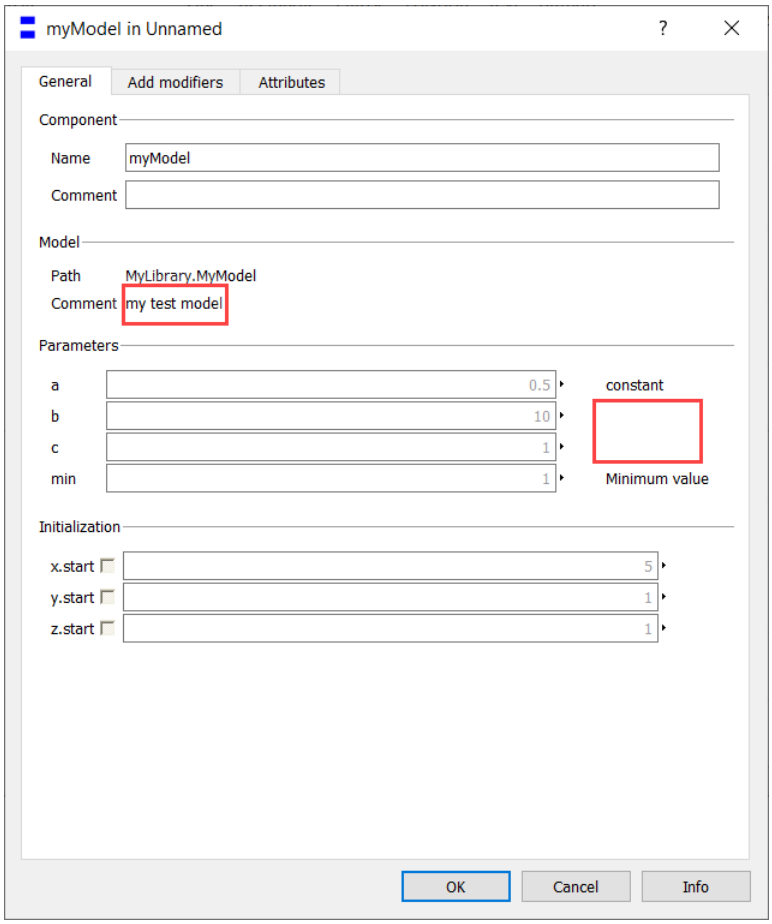
Output

Again, consider `MyLibrary`. The first class, `MyModel`, intentionally contains style errors.

Below is the Modelica text of that class to the left, and to the right, the parameter dialog. In the parameter part of the window one can see that the description strings are missing for parameter **b** and **c**, and that the description strings for the class and parameter **a** starts with a lower case character. These are style errors that will be detected by the style checking feature. The other classes in `MyLibrary` are implemented with correct style.

```
model MyModel "my test model"

  parameter Real a=0.5 "constant";
  parameter Real b=10;
  parameter Real c=1;
  parameter Real min = 1 "Minimum value";
  Real x(start=5);
  Real y(start=1);
  Real z(start=1);
  Boolean low;
equation
  low = x < min;
  der(x)=-a*x;
  der(y)=-b*y;
  der(z)=-c*z;
a
end MyModel;
```



Changing the default settings of the style check by specifying that 5 characters is enough for the documentation, the following style check log is generated when applying the Style Checking in the checkLibrary function to the model MyLibrary.MyModel (use the copy of the Library MyLibrary to check an encrypted library, see previous sections).

Style check log.

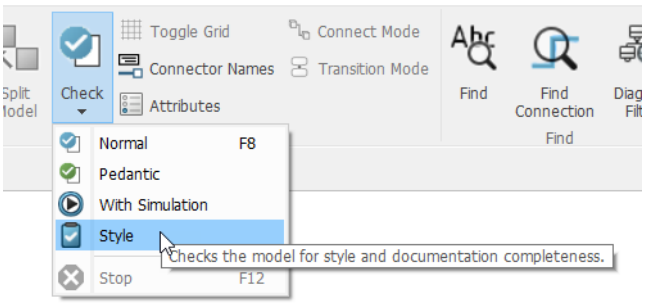
Library style check log

- **MyLibrary.MyModel**
  - Documentation missing
  - Class description string beginning with lower case character "my test model"
  - Component description string beginning with lower case character for
    - a "constant"
  - Description string missing for
    - b
    - c

The log reports 5 errors for `MyModel`. In addition to the errors described above it also reports that no HTML documentation exists for the class.

Style checking as the Check > Style command

The function `checkStyle()` is available in the ModelManagement.Check package. This is the default style checker, implemented by calling `checkLibrary()` using the default style check setup, see above. The style checker is typically run from the **Check** menu, available in the **Graphics** tab and the **Text** tab, (below the **Graphics** tab):



The user can instead use a custom style check function by setting the flag `Advanced.Check.StyleCheckFunction` to the full path of that Modelica function. The user-defined style checker must be a Modelica function, but it can of course invoke external tools. The style check function should take a single argument, a string with the path name of the model to check. The default value of the flag is:

```
Advanced.Check.StyleCheckFunction = "ModelManagement.Check.checkStyle"
```

If the style checker produces a file in the current directory called `<model name>_StyleCheckLog.html`, this file is automatically displayed by the **Graphics > Check > Style** or **Text > Check > Style** menu command.

9.5 Model comparison

9.5.1 Overview

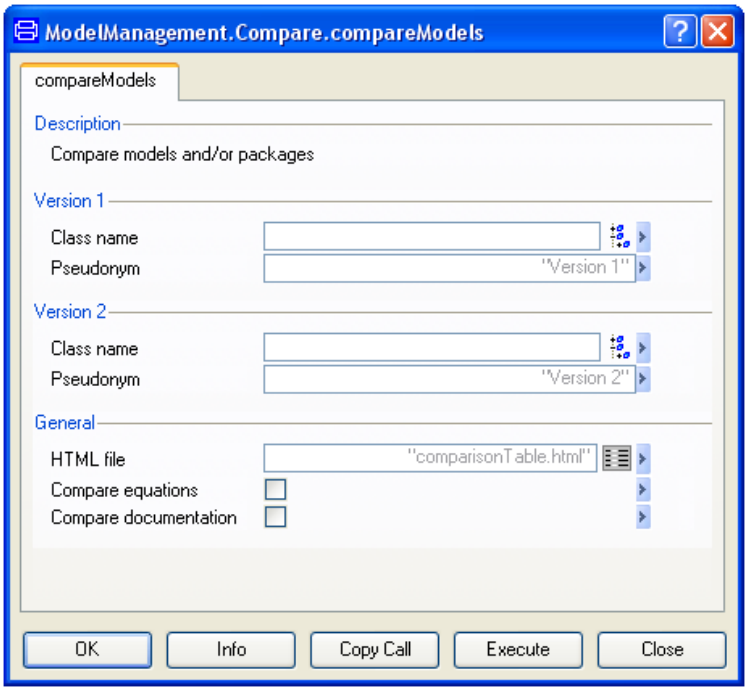
The aim of this package is to create a report with the differences between two classes. The package consists of a main function called `compareModels`.

To establish the comparison, the function `compareModels` needs as input parameters the name of the classes to be compared, together with corresponding pseudonyms (the names to be used in the report). The pseudonyms are by default Version 1 and Version 2. Another two input parameters are the check options Compare equations and Compare documentation. The differences between the two classes are presented in a HTML file whose name can be specified.

9.5.2 Getting started

This is a short getting started tutorial. It will demonstrate how to use `compareModels` to report differences in different versions of Modelica models or libraries.

- Make sure that the packages/models to be compared are available in the Package Browser. If not, load them by the **File > Open > Open…** command. To load two different versions of the same package, open the first one, and rename to e.g. name_a. Then load the second package and rename to e.g. name_b. Both packages should now be visible in the Package Browser.
  Note: It is not possible to rename read-only packages; instead you can duplicate and unload the original. The reason the second package is renamed is that there is an issue if the second name is a prefix of the first name.
- Right click the main function `compareModels` and choose **Call function...** to pop the dialog box depicted below.



- Now select your models/packages to be compared using the browsers under **Version 1** and **Version 2**.
- Enter the version names to display in the report by entering those names as pseudonyms under **Version 1** and **Version 2**.
- Use the browser to specify or select the output **HTML file** for the report.
- Optionally check the boxes for **Compare equations** and **Compare documentation** to include these features.
- Finally, press **Execute**

9.5.3 Comparison report

The comparison report consists of a number of possible tables for each class (in the case of comparing two packages, the classes with the same names are the ones to be compared as follows).

The first table appears when the attributes of the classes differ (for example, if one class is a model and the other one is an encapsulated model) or when components of the classes differ. The components are first compared by their names. For the components that share names, the comparison is based on all possible attributes (input, output, protected, graphical, modifiers, extent annotation, etc). The keywords are shown in the table with bold face. If there is a component which is present in only one of the classes it is expressed in the table as well as its position (extent annotation) in case of being graphical.

The second table presents the differences between the equations and it is created if the option Compare equations is checked.

The next table presents the differences between the documentation and it is created if the option Compare documentation is checked.

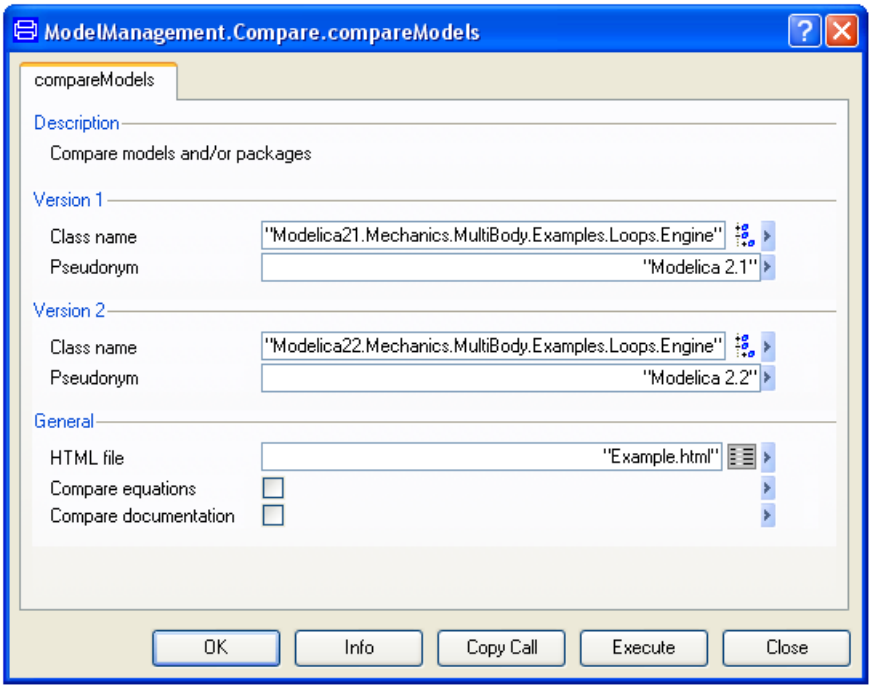
The comparisons between the equations of two classes and between their documentations are done using the same algorithm, based only on text comparison and the results of the comparison are also presented in the same fashion. In the tables appear mostly the differences between the texts, given a few common lines as references. The algorithm will recognize added/removed text or strings that have been changed. Moved text is not recognized as such.

In the case of comparing two packages, a last table may be created, containing the classes that only appear in one of the packages (see the last example).

To illustrate how the function `compareModels` works and the way in which the tables in the HTML file should be interpreted we present here some examples.

Class Attributes

We call the function **`compareModels`** with the setup as shown below.



The report is then in Example.html:

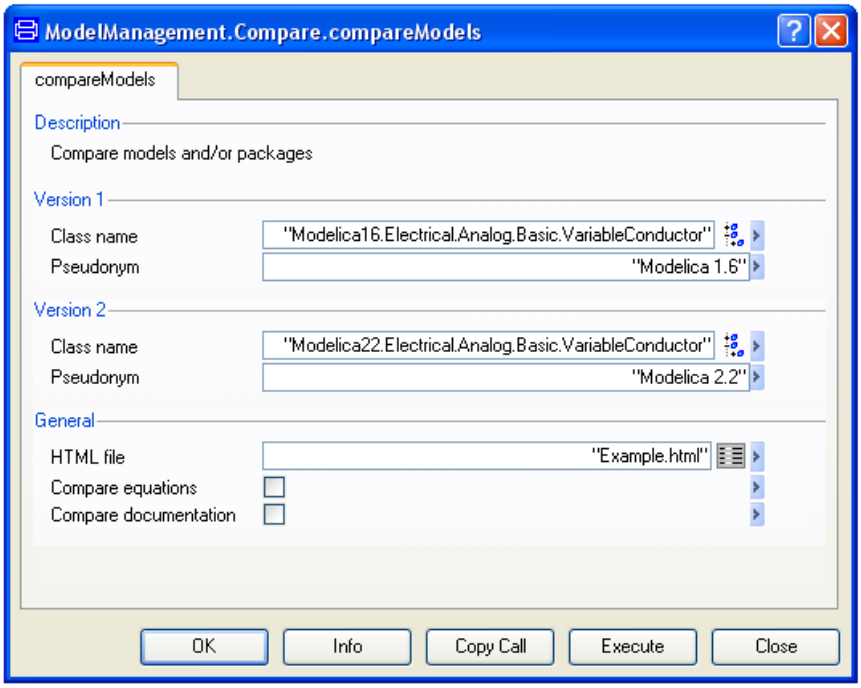
model Mechanics.MultiBody.Examples.Loops.Engine

Component	Modelica 2.1	Modelica 2.2
model	**encapsulated**	

This means that in Modelica 2.1 the model is encapsulated, but not in Modelica 2.2, and this is the only difference that they have (disregarding equations and documentation).

Components

We call the function **`compareModels`** as follows.



The report is then in Example.html:

model Electrical.Analog.Basic.VariableConductor

Component	Modelica 1.6	Modelica 2.2
G	**SIunits.Conductance**	**Blocks.Interfaces.RealInput**
	protected	
		graphical
		extent=[-20, 90; 20, 130]
G_Port	Present	
	extent=[-10, 90; 10, 110]	

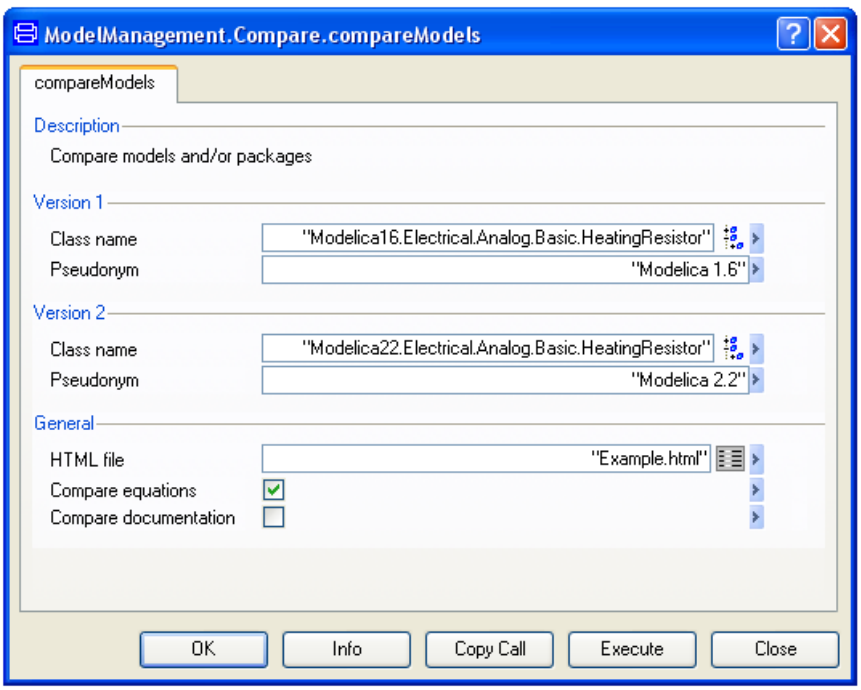
The component G is present in both models.

- In Modelica 1.6 it is defined `Modelica16.SIunits.Conductance` G among the protected members.
- In Modelica 2.2 it is defined as the public connector component `Modelica22.Blocks.Interfaces.RealInput` G. In addition in this version it is graphical and its position is indicated with extent.

The component G_Port is present only in Modelica 1.6. In Modelica 2.2 the simplification of the blocks library allowed G to be used both as input connector and directly in the equations.

Equations

We call the function **`compareModels`** to compare the HeatingResistor including its equations.



The report is then in Example.html:

model Electrical.Analog.Basic.HeatingResistor

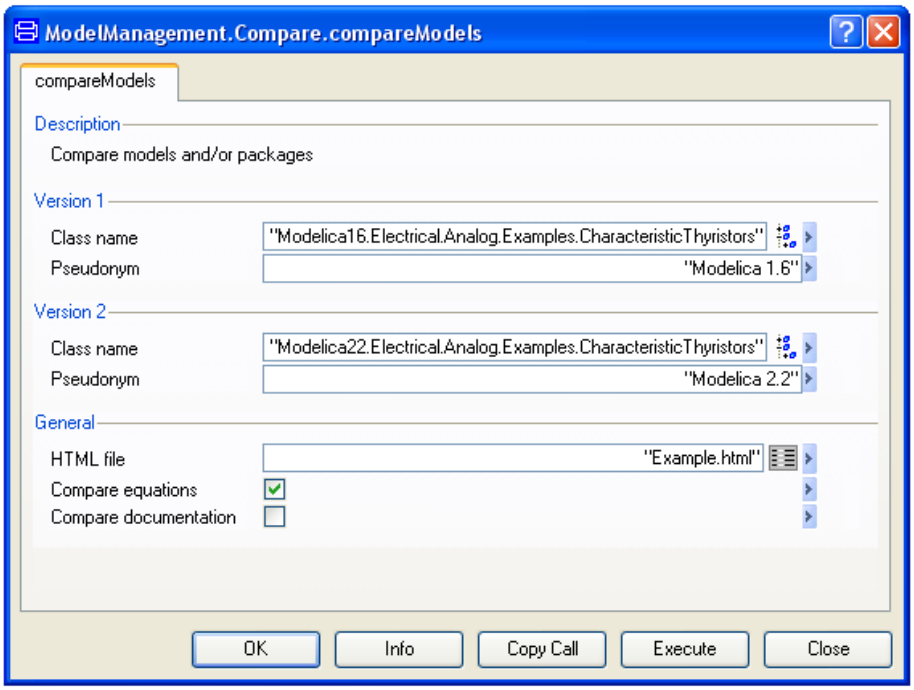
Component	Modelica 1.6	Modelica 2.2
R_ref		=1

Equations in Modelica 1.6	Equations in Modelica 2.2
... R = R_ref*(1 + alpha*(heatPort.T - T_ref));	
heatPort.Q_dot = -v*i;	heatPort.Q_flow = -v*i;
else ...	

If two equations have differences they are marked in boldface and with different colors. The one-column rows are common equations of the models. A default value has been added and the heat-flow variable was renamed to make it clear that it is not a time-derivative.

Connections

We call the function **`compareModels`** for CharacteristicThyristors.



The report is then in Example.html:

model Electrical.Analog.Examples.CharacteristicThyristors

Component	Modelica 1.6	Modelica 2.2
BooleanStep1	startValue={false}	startValue=false
	startTime={1.25}	startTime=1.25

Equations in Modelica 1.6	Equations in Modelica 2.2
... connect(SineVoltage1.p, IdealThyristor1.p);	
connect (BooleanStep1.outPort, IdealThyristor1.firePort);	connect(BooleanStep1.y, IdealThyristor1.fire);

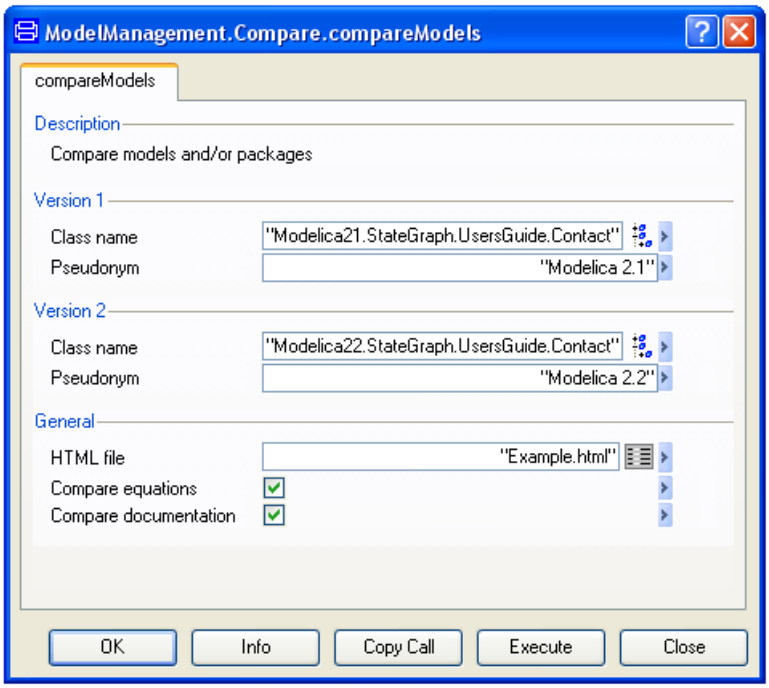
connect(IdealGTOThyristor1.n, R1.p);
...
connect(IdealGTOThyristor1.p, IdealThyristor1.p);	

connect (IdealGTOThyristor1.firePort, IdealThyristor1.firePort);	connect (IdealGTOThyristor1.fire, IdealThyristor1.fire);

When the connections between the components of the models are different they are presented as equations. The differences are due to the simplification of the blocks library in Modelica 2.

Documentation

We call the function **`compareModels`** with “Compare Documentation” enabled.



The report is then in Example.html:

class StateGraph.UsersGuide.Contact

Documentation in Modelica 2.1	Documentation in Modelica 2.2
... The development of this library was strongly motivated by the	
master thesis of Isolode Dressler	master thesis of Isolde Dressler
(see literature), ...	

The differences in the documentation are presented in analogous way to the equation differences.

9.6 Model structure

9.6.1 Introduction

This defines an Application Programmers Interface for traversing Modelica models, extracting information, and some modifications of the structure as well.

The API is defined in Modelica and to access it from non-Modelica code it relies on Dymola's interface to other languages.

The basics of the API are routines for accessing information for particular elements, giving the complete list of sub-elements. This allows extraction of the entire model information by recursively traversing the elements. The functions are stateless (i.e. there are no getNext-calls) corresponding to the functional style in Modelica.

There is no mapping from the structure to a complete structure in Modelica. Mapping the entire structure of a model to class-structure would not be possible in current Modelica because the class structure is inherently recursive. However, even if it were possible to replicate the entire class structure as a set of nested records it would not provide an efficient interface to the class structure for simple queries or modifications.

Important. In Dymola 2022, the API has been restructured, and a number of functions have been added; see "New from Dymola 2022: Improved model editing API" starting on page 1241. The text in the below sections are not yet updated to reflect this.

9.6.2 Traversing models before translation

In order to provide a useful interface to the classes and components three sets of routines are provided as follows in package ModelManagement.Structure.AST. The common basis is a set comprised of two functions and one record,

The three sets of routines are for classes, extends-clauses and components. In each set there is a routine for obtaining the elements (as an array), a record defining the "attributes" (protected, inner, full class name, …) and a routine for getting the attributes for a specific element.

These interfaces assume that one can use the name of elements in the queries, which is possible in the cases above (technically excluding the obscure case of repeated identical extends-statements which is legal Modelica, but without any reasonable use). Note that Dymola enforces this semantic restriction in Modelica already during parsing of classes, and thus it is safe to base the API-routines on this assumption.

The requirements also include access to the import-statements in the class. For import-clauses it is hard to define which name to use as a key (when considering both the qualified and the unqualified import-statements, thus a combined routine has been added that returns an array of records defining the import-statements.

This routine is also present for the other cases and provides an easy to use interface. Such functions are trivial to implement based on the primary routines, and we give a full example below (excluding its documentation):

```
function ComponentsInClassAttributes
   "Get components of a class"
   input String className;
   output ComponentAttributes res[:]=
    GetComponentAttributes(className,
       ComponentsInClass(className));
algorithm
end ComponentsInClassAttributes;
```

Here the names of the components are constructed by ComponentsInClass and this is then used in a vectorized call (as defined in Modelica) of GetComponentAttributes to get the attributes of all components.

Thus functions exist for all elements of table given on the below (where "elements in class" has a class/package as input and get attributes also exist in a form that returns an array containing the attributes of all elements).

Overview of basic interface to class structure

	Record of attributes	**Elements in class**	**Get attributes**
Classes	ClassAttributes	ClassesInPackage	GetClassAttributes
Extends	ExtendsAttributes	ExtendsInClass	GetExtendsAttributes
Components	ComponentAttributes	ComponentsInClass	GetComponentAttributes
Import	ImportAttributes	ImportsInClassAttributes	

The row headings are the element types and the column headings the different functions (and records)

To make it possible to traverse all classes it is also possible to list all top-level classes (optionally limited to the ones defined in a specific Modelica file).

Example

The functions in ModelManagement.Structure.AST are documented online with description, and example of use. There is furthermore a small set of examples, and one example is extracting statistics for packages, an example of use is ModelManagement.Structure. AST.Examples.countModelsInPackage("Modelica"); which find all restricted classes and can be used to provide e.g. the following list of accessible classes (excluding protected and partial ones):

Statistics for Modelica Standard Library

	Modelica 1.6	Modelica 2.1	Modelica 2.2
Model	222	429	494
Block	71	147	147
Function	41	199	472
Type	485	513	513
Package	50	130	1447

The growth of the standard library in 2.1 is in part due to the fact that ModelicaAdditions libraries were completed and after (in some cases major) revisions included in the Modelica

Standard Library. The increase in Modelica 2.2 is to a large extent due to the Modelica.Media library.

9.6.3 Interface to semantics not only to syntax

The API above defines basic routines that can be used directly. They also provide the basis for writing functions intended to answer higher-level questions, e.g. to search in a hierarchy for all components declared of a certain class.

Programming such queries require that the API answers questions related to the semantics of the declarations instead of questions based on their syntax (i.e. Dymola must not only parse the Modelica classes to answer the question, but also implement e.g. the semantics of look-up in Modelica).

To clarify this, consider the declaration of T2 in the coupled clutches demo:

```
parameter SI.Time T2;
```

To obtain information about this declaration we can use the following:

```
ModelManagement.Structure.AST.GetComponentAttributes(
"Modelica.Mechanics.Rotational.Examples.CoupledClutches", "T2")
```

this gives the result:

```
ModelManagement.Structure.AST.ComponentAttributes(
  name = "T2",
  fullTypeName="Modelica.SIunits.Time",
  isProtected = false,
  sizes = {},
  variability = "parameter",
  isInput = false,
  isOutput = false,
  isInner = false,
  isOuter = false,
  isReplaceable = false,
  isRedeclared = false,
  isGraphical = false)
```

By returning the full name of the type ("Modelica.SIunits.Time") and not the type-name part of the declaration ("SI.Time") it is straightforward to program this kind of queries and this also made it easier to program the calling interface in other languages.

Obviously advanced users would like to also have access to the exact declaration (including modifiers and annotations), and this is described in the next section.

Basing the API on the semantics is also important for API-routines that modify the classes, since e.g. copying (or moving) a class from one Modelica package to another might require changes to its declarations in order to ensure that the declarations refer to the same classes after the change. This is done automatically by Dymola's GUI and hidden from the user.

9.6.4 Extracting information and modifying structure before translation

The ModelManagement.Structure.AST contains routines for extracting information and modifying structure before translation. All of the routines are documented with information including description and examples. Thus, examine the information layer of the functions for further information.

The first section below contains additions and improvements in Dymola 2022. The rest of the section have not yet been updated to reflect these additions and improvements, so these headers state [not updated].

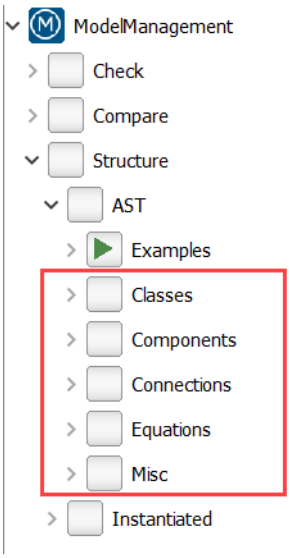
New from Dymola 2022: Improved model editing API

Introduction

The functions to create and edit Modelica models using function calls, located in the package `ModelManagement.Structure.AST` have been restructured and extended.

Improved AST package structure

The AST functions in the Model Management package have been grouped into five sub-packages for a better structure:



The five subpackages describe which part of a model that is edited, for example classes, components, connections, or other equations.

Automatic conversion

Automatic conversion is provided to upgrade existing code that uses the Model Management package to the new structure.

New AST functions

The following new functions have been added.

- New examples (in the **Examples** subpackage):

Name	Description
createNewModel	Creates a simple model using AST commands
makeReplaceable	Makes an existing component replaceable

- For editing classes (in the **Classes** subpackage):

Name	Description
CreateClass	Creates a new class with the given name
CreateExtends	Creates an extends in the designated class

- For editing components in a model (in the **Components** subpackage):

Name	Description
CreateComponent	Creates a new component in a class
SetComponentPlacement	Sets the graphical placement of a component in a model
SetComponentReplaceable	Set component replaceable with optional constraining type
SetComponentType	Sets the type of a component
SetAnnotation	Sets the annotation of a component

- For editing connections (in the **Connections** subpackage)

Name	Description
CreateConnection	Creates (or replaces) a connection without any annotation

DeleteConnnection	Deletes a connection between two connectors in the specified model
ConnectionsText	Returns the connections of the class as an array of text strings

- For editing equations (in the **Equations** subpackage)

Name	Description
CreateEquation	Creates one or more equations

Classes [not updated]

For classes it is possible to list classes in a package (ClassesInPackage), get attributes (GetClassAttributes), get a list with attributes (ClassesInPackageAttributes) and get the complete text of a class (GetClassText – with or without annotations).

One can thus e.g. extract all classes in package:

```
ModelManagement.Structure.AST.ClassesInPackage(
"ModelManagement.Structure.AST.Examples");

 = {"countModelsInPackage", "givePackagesInPackage",
    "attributeModelsInPackage"}
```

and extract the complete text (in this case without annotations) of an individual class:

```
ModelManagement.Structure.AST.GetClassText(
"ModelManagement.Structure.AST.GetClassText")

 = "function GetClassText
  input String fullName;
  input Boolean includeAnnotations=false;
  output String prettyPrinted;
external \"C\"
prettyPrinted=Dymola_AST_ClassText(fullName,includeAnnotations)
;
end GetClassText;
"
```

Components [not updated]

For components it is possible to list components in a class (ComponentsInClass), get the component attributes (GetComponentAttributes), get a list with component attributes (ComponentsInClassAttributes), get modifiers (ComponentModifiers) and the entire text of a component (GetComponentText – with or without annotations).

The modifiers are returned as an array of modifiers:

```
ModelManagement.Structure.AST.ComponentModifiers(
"Modelica.Mechanics.Rotational.Examples.CoupledClutches","J1");
```

Returns

```
= {"J=1", "phi(start=0)", "w(start=10)"}
```

Modifiers for the value are indicated with a leading '=':

```
ModelManagement.Structure.AST.ComponentModifiers(
"Modelica.Mechanics.Rotational.Examples.CoupledClutches","T2");
= {"=0.4"}
```

Equations and connections [not updated]

For models it is important to not only examine components, but also equations (EquationBlocks) – in particular connection-equations (Connections).

Both of these routines return a list of strings as follows:

```
ModelManagement.Structure.AST.EquationBlocks(
"Modelica.Mechanics.Rotational.Examples.CoupledClutches");
```

returns

```
 = {"  connect(sin1.y, torque.tau);", "
connect(torque.flange_b, J1.flange_a);",
   "  connect(J1.flange_b, clutch1.flange_a);", "
connect(clutch1.flange_b, J2.flange_a);",
   "  connect(J2.flange_b, clutch2.flange_a);", "
connect(clutch2.flange_b, J3.flange_a);",
   "  connect(J3.flange_b, clutch3.flange_a);", "
connect(clutch3.flange_b, J4.flange_a);",
   "  connect(step2.y, clutch3.f_normalized);", "
connect(step1.y, clutch2.f_normalized);",
   "  connect(sin2.y, clutch1.f_normalized);"}
```

The difference between EquationBlocks and Connections is that Connections only return the connection-equations. It is optional to include the annotations as well.

Annotations [not updated]

Extracting information is useful, e.g. to get default simulation stop-time, documentation, and placement of components.

To extract the default stop-time for the r3-robot:

```
ModelManagement.Structure.AST.GetAnnotation(
  "Modelica.Mechanics.MultiBody"+
".Examples.Systems.RobotR3.fullRobot","experiment.StopTime");
```

returns

```
= "=3"
```

Since many of the annotations are strings there is direct support for extracting the string itself:

```
ModelManagement.Structure.AST.GetAnnotationString(
"Modelica.Mechanics.MultiBody"+
".Examples.Systems.RobotR3.fullRobot","Documentation.info");
```

returns

```
 = "<HTML>
<p>
…
```

Extends and import [not updated]

Only the basic routines as presented in section "Traversing models before translation" are present.

Modifying models [not updated]

In order to modify models it is possible to copy or move a class.

These correspond to the package-browser context menu entries **New > Duplicate Class…** and **Rename…**.

As an example consider making a copy of the Inertia-model:

```
ModelManagement.Structure.AST.CopyClass("Modelica.Mechanics.Rot
ational.Inertia",
   "Inertia");

= true
```

Looking at the angular velocity(w) in the original component we note that it uses an import-statement from the package:

```
ModelManagement.Structure.AST.GetComponentText(
  "Modelica.Mechanics.Rotational.Inertia", "w");

 = "  SI.AngularVelocity w \"Absolute angular velocity of
component\";"
```

Since the new declaration is not in the same package the declaration is modified to preserve the reference:

```
ModelManagement.Structure.AST.GetComponentText(
   "Inertia", "w");

 = "  Modelica.SIunits.AngularVelocity w \"Absolute angular
velocity of component\";"
```

The class Inertia can then be moved to a new location I2 (if not already present):

```
if not ModelManagement.Structure.AST.ClassExists("I2") then
  ModelManagement.Structure.AST.MoveClass("Inertia","I2");
end if;
```

Internal references are also updated when moving (in the same way as copying), but references to the old class will not automatically refer to the new name.

File-system interface [not updated]

In order to modify and manage models it is necessary to read it from a file (ReadModelicaFile), erase it from memory (EraseClasses), save the model (SaveModel), or save a total model (SaveTotalModel). An example of saving and reading model is:

```
ModelManagement.Structure.AST.SaveModel("U2.mo","Unnamed");
ModelManagement.Structure.AST.EraseClasses({"Unnamed"});
ModelManagement.Structure.AST.ReadModelicaFile("U2.mo");
```

This saves Unnamed in a file, erases it from memory, and reads it back in (i.e. it is back as before).

EraseClasses takes vector of classes in order to be able remove interdependent classes at the same time without having any dangling references.

9.6.5 Traversing translated models

During translation (and check) the model is instantiated, i.e. the abstract syntax tree is transformed into a specific structure for the model. To answer questions related to this the package ModelManagement.Structure.Instantiated provides several API-routines. Note that the instantiation can be time-consuming in order to get the complete result, and thus these routines can be slower than the AST-routines.

These routines are used to implement the check in ModelManagement.Check, and can answer questions such as number of equations and unknowns, what are the possible states (discrete and continuous), and which models are used for components (useful for e.g. test-coverage). The content is:

Content of ModelManagement.-Structure.Instantiated.

Name	Description
NrEquations	Get the number of equations of a model
NrUnknowns	Get the number of unknowns of a model
ListPossibleContinuousStates	Get possible continuous states
ListPossibleDiscreteStates	Get possible discrete states
UsedModels	Get classes that are used as components in a model
CountUsedModels	Get the number of instances of components in a model

The number of equations and unknowns is counted as for check, and not as for translate. The numbers differ slightly, since during check an external environment for physical connectors is assumed (in order to allow check of models that cannot be translated on their own).

The continuous (and discrete) states are important to verify results. The term possible is used, since for dynamic state-selection all possible states are not active at all points in time.

The counting of used models is split into two routines: one that returns the used models and another that returns the count for specific models.

10 FMI, eFMI, AND SSP SUPPORT IN DYMOLA

10 FMI, eFMI, and SSP Support in Dymola

This chapter describes the support for the Functional Mockup Interface (FMI), FMI for embedded systems (eFMI), and System Structure and Parametrization (SSP):

- FMI Introduction (page 1250)
- Exporting FMUs from Dymola (page 1251)
- Importing FMUs to Dymola (page 1271)
- Validating FMUs from Dymola (page 1289)
- Profiling of FMUs (page 1292)
- FMI Kit for Simulink (page 1292)
- eFMI support in Dymola (page 1293)
- SSP Introduction (page 1294)
- SSP Import (page 1295)
- SSV Import (part of SSP import) (page 1297)
- SSP Export (page 1297)

10.1 FMI Support in Dymola

10.1.1 Introduction

FMI

The FMI ("Functional Mock-up Interface") standard allows any modeling tool to generate C code or binaries representing a dynamic system model that may then be seamlessly integrated in another modeling and simulation environment.

FMI started as a key development effort within the MODELISAR project, see

https://itea3.org/project/modelisar.html

The FMI standard is today maintained and developed as a long-term project within the Modelica Association.

Three official FMI specifications have been released. The 'FMI for Model Exchange' specification version 1.0 was released on January 28, 2010, and the 'FMI for Co-Simulation' specification version 1.0 was released on October 12, 2010. FMI 2.0 that merges the model exchange and co-simulation specifications into one document was published on July 25, 2014.

The model exchange specifications focus on the model ODE interface, whereas the co-simulation specifications deal with models with built-in solvers and coupling of simulation tools. A model package implementing the FMI standard is called a Functional Mockup Unit (FMU). For further details, visit:

http://www.fmi-standard.org/

The specification documents are also available in Dymola using the command **Help > Documentation**. The specifications are separated into an execution part (C header files) and a model description part (XML schema). A separate model description is used in order to keep the executable footprint small. Both FMI 1.0 specifications use essentially the same XML schema (a couple of capability flags are introduced for FMI for Co-Simulation).

In summary, an FMU (Functional Mock-up Unit) implementing an FMI specification consists of

- The XML model description.
- Implementation of the C function interface in binary and/or source code format.
- Resources such as input data.
- Image and documentation of the model.

FMI support in Dymola

The Dymola FMI support consists of the two built-in functions described below for FMU export and import, respectively. Commands are also available in the Dymola user interface to execute these functions.

The first three items in the list above are currently supported by Dymola. FMI (both Model Exchange and Co-Simulation) is supported for Windows and Linux.

Unless otherwise stated, features are available for both FMI version 1.0 and version 2.0.

For the latest information about limitations and supported features of FMI, please visit www.Dymola.com/FMI.

Online tunable parameters

Online tunable parameters are supported in FMI version 2.0 (tunable parameters were not allowed in FMI version 1.0). For declaring tunable parameters as fixed, you can use the setting **Declare tunable as fixed**. See section "Options group" starting on page 1257.

10.1.2 Exporting FMUs from Dymola

FMU export by the built-in function translateModelFMU

Exporting FMU models from Dymola is achieved by the function

```
translateModelFMU(modelToOpen, storeResult, modelName,
fmiVersion, fmiType, includeSource, includeImage,
includeVariables[:])
```

The input string `modelToOpen` defines the model to open in the same way as the traditional `translateModel` command in Dymola.

Note that, from Dymola 2020x, the function by default uses any experiment (simulation) setting of the model (that is, settings in the **General** tab of the simulation setup; for example, start time, stop time, and algorithm), to make it easier to script these settings. If this is not wanted, you can set the flag `Advanced.FMI.UseExperimentSettings=false`. (The flag is by default `true`.)

The Boolean input `storeResult` is used to specify if the FMU should generate a result file. If `storeResult` is true, the result is saved in *<model id>*`.mat` when the FMU is imported and simulated, where *<model id>* is given at FMU initialization. (If empty, "`dsres`" is used instead.) This is useful when importing FMUs with parameter `allVariables = false`, since it provides a way to still obtain the result for all variables. The result generation interval when storing simulation results can be controlled by the flag `Advanced.FMI.FMUStoreResultInterval`. The default value of the flag is `0.001`. Simultaneous use of result storing and source code inclusion (see below) is not supported.

The input string `modelName` is used to select the FMU model identifier. If the string is empty, the model identifier will be the name of the model, adapted to the syntax of the model identifier (for example, dots will be exchanged with underscores). The name must only contain letters, digits and underscores. It must not begin with a digit.

The input string `fmiVersion` controls the FMI version ("`1`" or "`2`") of the FMU. The default is "`1`".

The input string `fmiType` defines whether the model should be exported as

- Model exchange (`fmiType="me"`)
- Co-simulation using Cvode (`fmiType="cs"`),

- Both model exchange, and Co-simulation using Cvode (`fmiType="all"`)
- Co-simulation using Dymola solvers (`fmiType="csSolver"`).

The default setting is `fmiType="all"`. This parameter primarily affects modelDescription.xml. For the three first choices binary and source code always contains both model exchange and Co-simulation. For the last choice, the binary code only contains Co-simulation; the solver and tolerance that are selected in Dymola in the general tab in the simulation setup are also used by the exported FMU. Note that co-simulation using Dymola solvers requires the Binary Model Export license. Please see also "Notes on Co-Simulation" on page 1267 concerning Co-simulation.

The Boolean input `includeSource` is used to specify if source code should be included in the FMU. The default setting is that it is not included (`includeSource=false`). Simultaneous use of result storing (see above) and source code inclusion is not supported. Notes:

- Source code generation is not supported for Co-simulations using Dymola solvers.
- General source code documentation is available in the Documentation folder inside the generated FMU folder.
- You can use sparse Jacobians when exporting FMUs with source code included. See "Using sparse Jacobians when exporting FMUs with source code included" on page 1263.
- From Dymola 2020, when exporting source code FMUs using external c-functions the source code of those functions are included, specifically:
    - The contents of includeDirectory will be included under sources/include
    - The contents of __Dymola_SourceDirectory (default "…/Resources/Source") will be included under sources/source.

The Integer input `includeImage` is used to specify if an image should be included in the export. The default is `0`, meaning no image is exported. `1` means including the icon of the model in the export, `2` means including the image of the diagram layer of the model in the export. The image is stored in the FMU resources with the name `model.png` which actually means it can be exchanged with any image `model.png`.

The string input parameter `includeVariables[:]` is by default an empty array, but variables can be added in this array. If the default value (an empty array) is kept, all variables that passes selected model description filtering are kept. To control this filtering in scripting, you can use the three flags described in "Model description filters group – filtering variables" on page 1257.

If variables are added, only these variables are included in the generated FMU. If values that are not included in the model are added, you will get warnings when exporting the model. You can add variables to export in the dialog for exporting an FMU by the selection **Select variables**, see section "Variables to export group – option to filter on individual signals" on page 1261.

The function outputs a string `FMUName` containing the FMU model identifier on success, otherwise an empty string.

As an example, translating the Modelica CoupledClutches demo model to an FMU with result file generation, is accomplished by the function call

```
translateModelFMU("Modelica.Mechanics.Rotational.Examples.
CoupledClutches", true);
```

After successful translation, the generated FMU (with file extension .fmu) will be located in the current directory. The user can select if 32-bit and/or 64-bit FMU binaries should be generated – see the FMI tab description below.

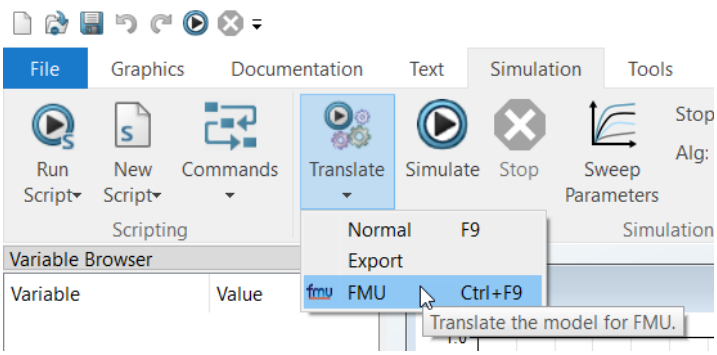
The generated FMU contains information about if it has been generated without export options. In the corresponding XML file of such an FMU, the following is seen:

```
generationTool="Dymola Version 2015 (64-bit), 2014-02-21
(requires license to execute)"
```

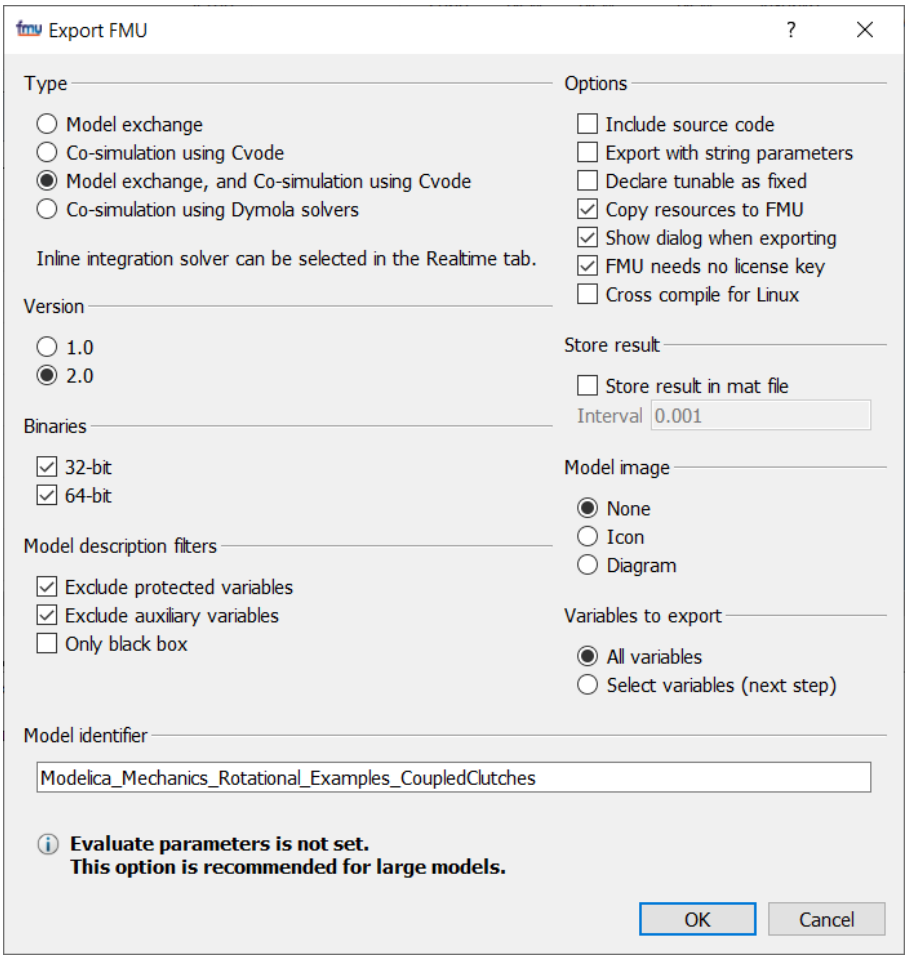
FMUs exported from Dymola support intermediate results for event update (fmiEventUpdate) for Model Exchange for FMI version 1.0.

FMU export by command

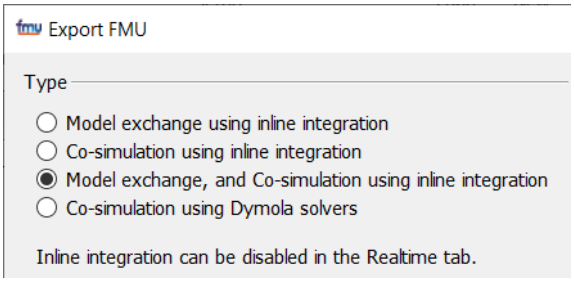
An alternative to executing the `translateModelFMU` function is to use the command **Simulation > Translate > FMU** as illustrated below:



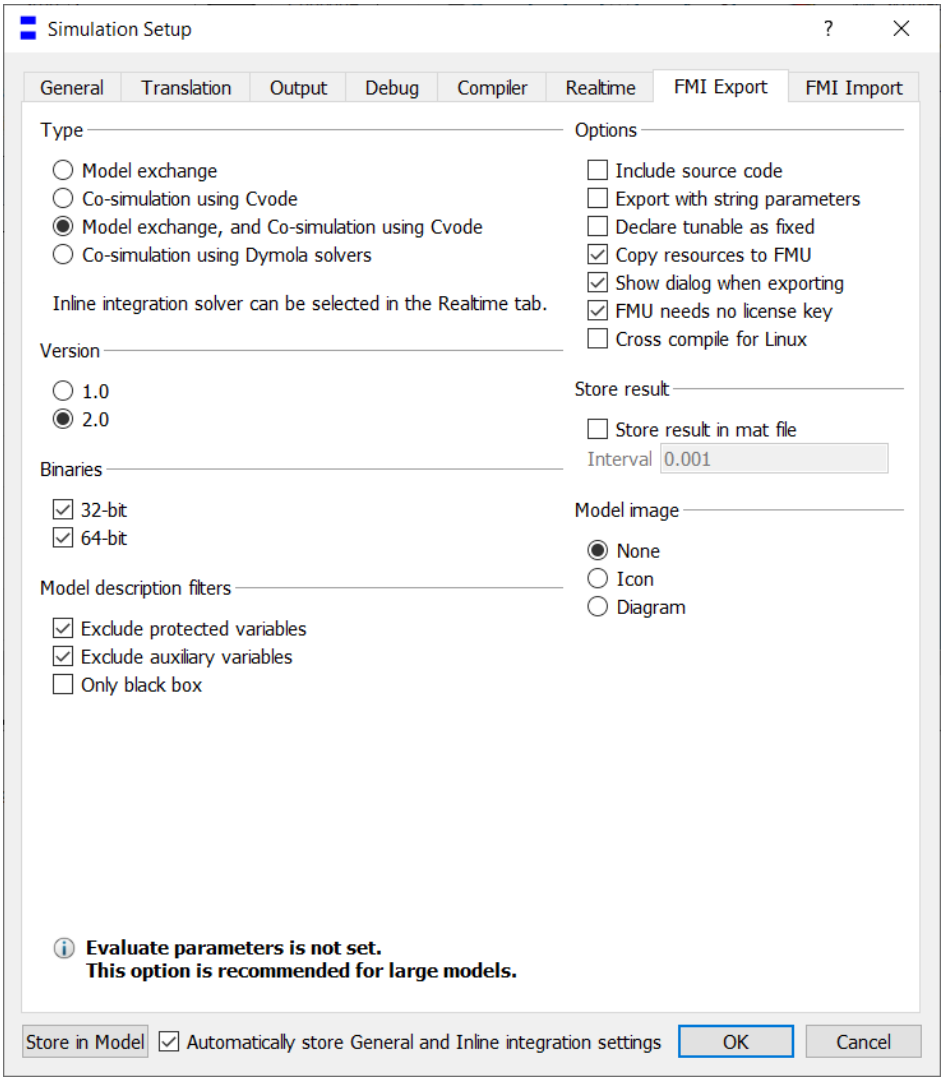
The settings that will be used when using any of the above commands is specified in a dialog that appears when the command has been given:



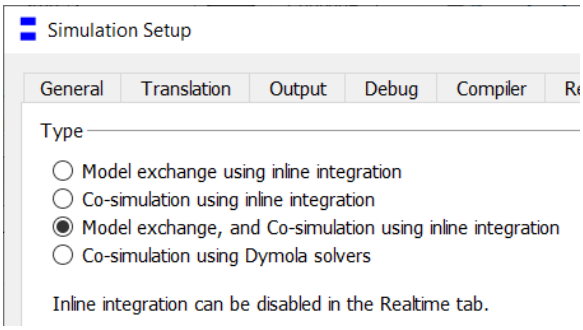
The Type group in the dialog above looks different if any inline integration solver has been selected in the **Realtime** tab of the simulation setup (reached by the command **Simulation > Setup**, the **Realtime** tab):



This dialog corresponds to the **FMI Export** tab of the simulation setup (reached by the command **Simulation > Setup**, the **FMI Export** tab), except for the groups **Variables to export** and **Model identifier**:



As in the dialog for exporting an FMU, the Type group looks different if any inline integration solver has been selected in the **Realtime** tab of the simulation setup:



Changing settings when exporting will affect also this menu. Changed settings are remembered in the session, but not between sessions.

Note the information message that appears in the lower part of the menu if the setting **Evaluate parameters to reduce models (improves simulation speed)** is not enabled in the **Translation** tab of the simulation setup. (Reached by the command **Simulation > Setup…**, the **Translation** tab.)

Type group

When any inline solver is not selected in the **Realtime** tab of the simulation setup, the FMI type can be selected as **Model exchange**, **Co-simulation using Cvode**, **Model exchange, and Co-simulation using Cvode** or **Co-simulation using Dymola solvers**; this setting corresponds to the parameter `fmiType` in `translateModelFMU` (see description above of this setting for more information). There is also a corresponding flag `Advanced.FMI.FMUFMIType` with the same alternatives, except the default of the flag being an empty string.

When any inline solver is selected in the **Realtime** tab of the simulation setup, the corresponding selections are **Model exchange using inline integration**, **Co-simulation using inline integration**, **Model exchange, and Co-simulation using inline integration**, and finally **Co-simulation using Dymola solvers**. They also correspond to the the parameter `fmiType` in `translateModelFMU` and the flag flag `Advanced.FMI.FMUFMIType`, the reason for having specific selections for inline integration in the GUI is to make it clearer what is selected, in particular concerning the use of the Cvode solver; that solver not used for inline FMUs. For more information about Cvode, see section “Notes on Co-Simulation” on page 1267.

Version group

The FMI version can be selected as "`1`" or "`2`", the default being "`1`". The setting corresponds to the parameter `fmiVersion` in `translateModelFMU`. There is also a corresponding flag `Advanced.FMI.FMUFMIVersion` with the same alternatives, except the default of the flag being an empty string.

Binaries group

The user can select whether 32- and/or 64-bit FMU binaries should be generated. This option is not available in `translateModelFMU`.

Model description filters group – filtering variables

You can control the filtering of the `modelDescription.xml` file with these settings:

- **Protected variables** (by default activated) filters away protected Modelica variables. This setting corresponds to the flag `Advanced.FMI.xmlIgnoreProtected = true;`.
- **Auxiliary variables** (by default not activated) works differently in FMI version 2.0 and FMI version 1.0:
    - For FMI version 2.0 activating this setting means filtering away all variables of causality local, except states and derivatives of states.
    - For FMI version 1.0 activating this setting means all variables of causality internal except the ones with variability parameters are filtered away.

  This setting corresponds to the flag `Advanced.FMI.xmlIgnoreLocal = false;`.
- **Black-box** (by default not activated) works differently in FMI version 2.0 and FMI version 1.0:
    - For FMI version 2.0 activating this setting means filtering away all variables except the following:
        - Variables of causality inputs and outputs
        - Variables needed for the model structure. The names are however hidden (concealed).
    - For FMI version 1.0 activating this setting means filtering away all variables except variables of causality inputs and outputs.

  This setting corresponds to the flag `Advanced.FMI.BlackBoxModelDescription = false;`.

  Black-box export can be used to export sensitive models without exposing the names of parameters and internal variables.

  Note that if you activate **Black-box**, the settings **Protected variables** and **Auxiliary variables** are dimmed; they are not relevant is this case.

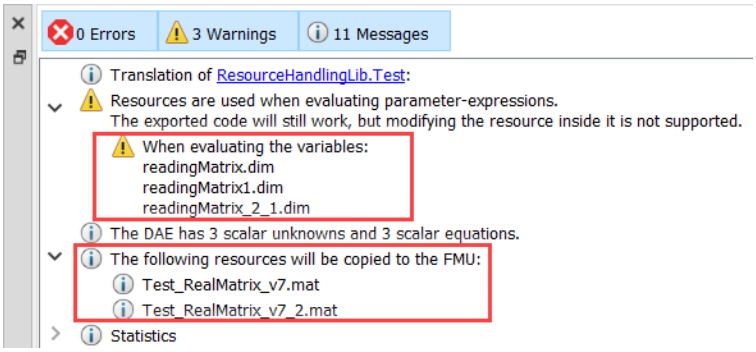
Options group

Seven options are available. Note that the two first ones cannot be ticked simultaneously.

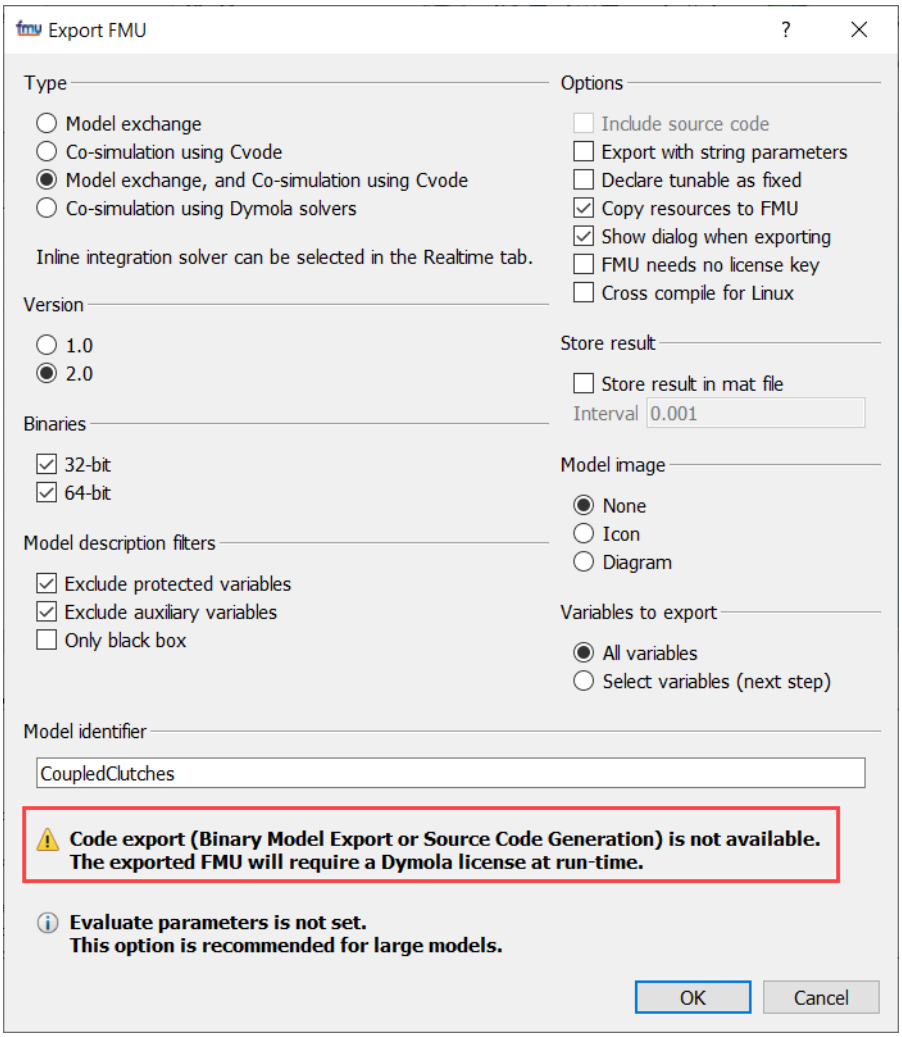
- **Include source code** – corresponds to the parameter `includeSource` in `translateModelFMU`. If ticked (`includeSource=true`) source code is included, if unticked the source code is not included (default). For more details, see the description of the parameter in the section about `translateModelFMU` above. Note that there is a

corresponding flag `Advanced.FMI.FMUIncludeSource` with the same alternatives as the parameter `includeSource`, the default is `true`.

- **Export with string parameters** – enables using string parameters when exporting FMUs. All types of FMUs are supported. Notes:
    - The default value is `false`, since you would normally not want the regular simulation to use string parameters as this makes the code slightly less efficient.
    - The default setting corresponds to the flag `Advanced.AllowStringParametersForFMU = false`.
    - String variables are currently not supported.
    - For examples, see "String parameter support - examples" starting on page 1264.
- **Declare tunable as fixed** - for FMI 2.0, you can activate this setting to declare all tunable parameters as fixed instead of tunable in modelDescription.xml of the exported FMU. Normally, Dymola deduces if a parameter is tunable or fixed, and declares it accordingly. By declaring them as fixed, it signals to the importing tool that it is illegal to change them after initialization. Activating the setting corresponds to setting the flag `Advanced.FMI2.NoTunableParameters = true`.
- **Copy resources to FMU** – external resources using the functions `ModelicaServices.ExternalReferences.loadResource` or `Modelica.Utilities.Files.loadResource` are by default copied to the FMU. (Note that the URI called by the function must be a string literal or a string parameter.) The resulting FMU will be larger due to this. If this is not wanted, de-selecting the setting will not copy the resources to FMU, but the resource-paths using Windows-shares will be changed to UNC-paths when possible. This makes the FMU usable within a company – without increasing its size. An example of using the resource copying is given below, the extended example in the "String parameter support - examples" section. (There is a corresponding flag `Advanced.FMI.CopyExternalResources` with the default value `true`.) Note that even if you select to copy resources to the FMU when exporting it, you cannot change structural parameters depending on these resources after the FMU is created. The reason is that Dymola must evaluate the structural parameters in the beginning of the translation of the model to the FMU, by reading the resources. Structural parameters involve for example sizes of vectors and matrices. The only way to change such sizes is to recreate the FMU after changing the resources. The translation log contains a warning and the names of the structural parameters that have been evolved in such a way. An example, the first red frame below (note the second framed message listing all files that will be copied to the FMU):



- **Show dialog when exporting** – this option is by default ticked. If unticked, the Export FMU dialog is not displayed when exporting FMUs.
- **FMU needs no license key** – this option corresponds to the option **Enable checking out code export options** in the license setup dialog, reached by the command **Tools > License Setup**, the **Details** tab. These options are working on the same feature, if you activate it in one dialog; it is activated also in the other dialog. By default, this option is activated. If you export an FMU without any code export license, you will get a warning:



The warning is given under two conditions:

- The code export is disabled by selection.
- If the code export is not disabled by selection, Dymola tries to check out an export option. If none is available (no such license, or license already used up by other users), the warning is shown.

If you export an FMU without any code export license, the usefulness of the exported FMU is greatly reduced. To export an FMU without any code license should be easy to do if you have a small number of export licenses, because in that case people are usually required to disable code export while doing model development.

- Cross compile for Linux – activates cross-compilation for Linux on Windows. For more information, see section “Activating FMU export for multiple platforms” on page 1271.

Store result group

- **Store result in mat file** – corresponds to the parameter `storeResult` in `translateModelFMU`. If enabled (`storeResult=true`) a result file is generated and stored as a .mat file <*model id*>`.mat`, if not enabled, no result file is generated. For more details, see the description of the parameter in the section about `translateModelFMU` above. There is a corresponding flag `Advanced.FMI.FMUStoreResult` with the default value `false`.
- **Interval** – If the above setting is enabled, the result generation interval when storing simulation results can be controlled here. The default value is `0.001`. There is a corresponding flag `Advanced.FMI.FMUStoreResultInterval` with the same default.

Model image group

You can select to include a model image in the FMU export. The following alternatives are possible:

- **None** (default) No image is included in the FMU export.
- **Icon** The icon of the model is included in the FMU export.
- **Diagram** The image of the diagram layer of the model is included in the FMU export.

If an FMU with an image included in the export is imported, instances of this imported FMU will display this image. See "Using the exported model image to display the imported FMU" on page 1281.

The alternatives above corresponds to parameter `includeImage` in `translateModelFMU`, with the values `0`, `1`, and `2`, respectively. There is a corresponding flag `Advanced.FMI.ImageSetting` that with the same values.

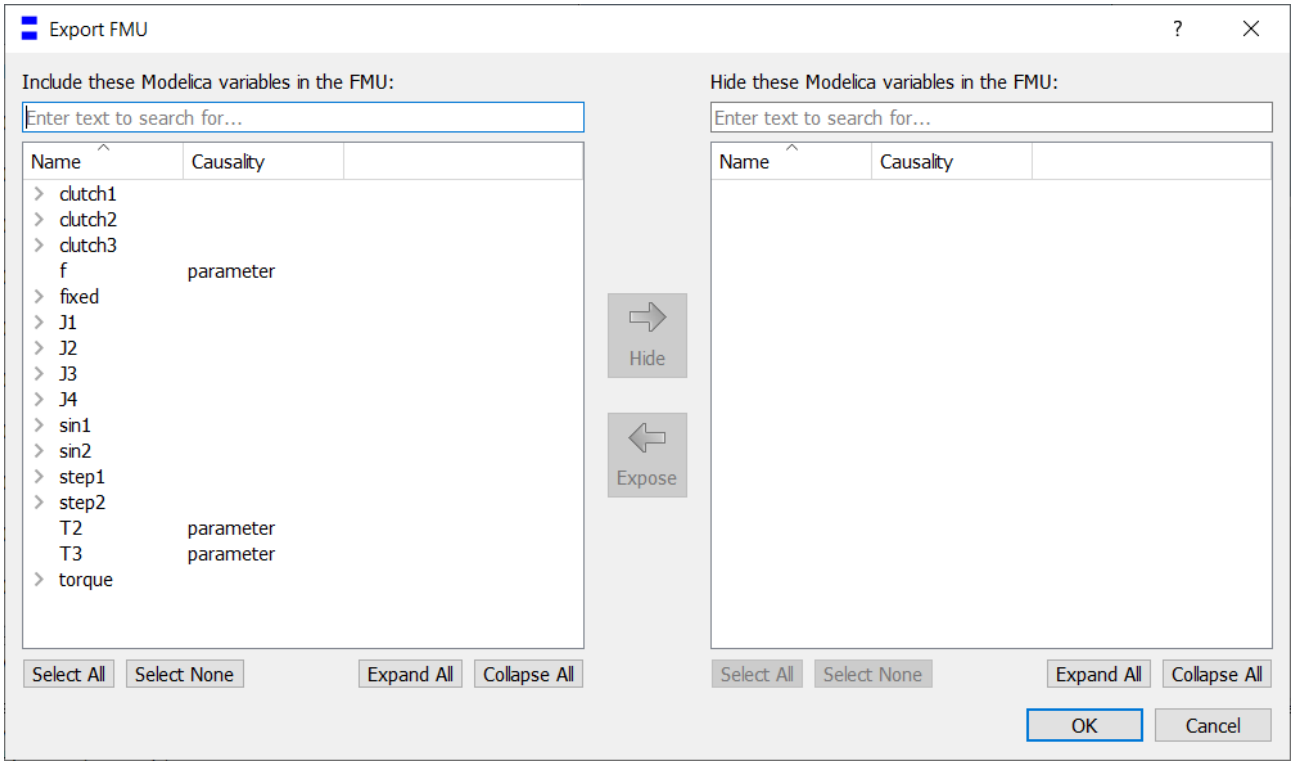
The image is stored in the FMU resources with the name `model.png`. (Note that this also means that you can use any image `model.png` to replace it.)

Variables to export group – option to filter on individual signals

You can select any of the alternatives:

- **All variables** – this is the default option, it means that the filtering is according to what you have selected in the **Model description filters** group above – see that group.
- **Select variables (next step)** – this is an option to filter variables individually.

If you select **Select variables (next step)**, and click **OK**, you will get another dialog (the below is an example exporting the demo Coupled Clutches as an FMU):



You select the variables you want to move to the other pane, and then click the arrow **Hide** or **Expose** depending on in which pane you want to have the variables, if they should be exposed or hidden.

For each pane, you have a search field on top, and four buttons at the bottom: **Select All**, **Select None**, **Expand All**, and **Collapse All**. Note that it might be handy to, in the above figure, to click **Select All** and then click the **Hide** arrow, if you want to select which variables to expose rather than the ones to hide.

When the menu appears, it is by default adapted to what is selected in the Model description filters group in the previous menu.

Note! Inputs cannot be deselected (hidden). They cannot be selected or moved in the menu, they are grayed out in the menu above, if present.

This filtering can also be used in scripting, the selected variables will be added in the array of the `includeVariables[:]` parameter of the built-in function `translateModelFMU`. Note that in this case, the flags corresponding to the **Model description filters** group above are ignored.

Model identifier group

This group only appears in the dialog when exporting an FMU, not in the general **FMI Export** tab.

Model identifier User-defined model identifier of the FMU. The default value is the name of the model to export, including the path. (Two images above, it is the Coupled Clutches demo.) The setting corresponds to the parameter `modelName` in the built-in function `translateModelFMU`. The setting also decides the exported FMU file name.

Specifying the maximum simulation run time per simulation for an exported FMU

You can specify the maximum simulation run time per simulation for an FMU by the simulation setup, the **General** tab, the **Per simulation** option. (The simulation setup is reached by the command **Simulation > Setup**.)

By default, the maximum simulation run time is not specified, meaning there is no simulation stop time.

The alternative **Per simulation** means that each simulation gets its own timer, with the specified time. Each simulation starts its own timer when it starts the simulation and stops itself if this timer runs out. Note that wall-clock time is used.

This option may be useful if a few of the simulations in a batch take very long time and eventually fails. Then these timers can be automatically stopped early, without affecting the other simulations. That is, each simulation has its specified max simulation run time, and that time is not affected by the simulation time of any other previous simulation, as is the case if a single timer is used. (Note: The single timer option (**For batch**) is not applicable for exported FMUs.)

The option **Per simulation** is embedded in the simulator, including an exported FMU.

Exporting a model in DAE mode

Exporting a model in DAE mode as an FMU is supported if the selected solver supports DAE mode. If this is not the case, the export is not allowed – an error message will be displayed.

Handling of large matrix dependencies when exporting an FMU

To limit the size of the generated FMU by avoiding a very large number of initial unknowns for long parameter vectors, a flag monitors the maximum size of number of elements in interactive parameters. If the number of elements exceeds the value of the flag, a warning is given, and the parameter is removed as an interactive parameter, and will not have any declared start value in the modelStructure.xml file.

The name of the flag is `Advanced.FMI.MaximumSizeOfInteractiveParameters`, and the default value is `100`. You can change the value of the flag if you need to include interactive parameters with more than 100 elements as interactive parameters.

Using sparse Jacobians when exporting FMUs with source code included

FMU source code export includes code for building the Cvode solver. Cvode can be enhanced with sparse solver capabilities. If the flag `Advanced.SparseActivate` is set to `true` before the FMU export, additional files are copied to the "sources" subdirectory of the generated FMU directory.

The details on how to enable the code for sparse solvers are found in the documentation provided in the FMU directory, in the index.html file, located in the generated FMU. (To display this directory, use any zip tool to reveal the structure of the generated FMU.)

Including settings in the exported FMU

Note the possibility to include settings in the exported FMU by ticking **Settings included in translated model**, reachable by the command **Simulation > Setup…**, the **Debug** tab. (If such settings are included in a Dymola-generated FMU, they can be logged by activating `fmi_loggingOn` in the FMI tab of the parameter dialog of the imported and instantiated FMU.)

Option to include "save total" of the model in the generated FMU

It is possible to include a "save total" of the model in the generated FMU by setting the flag

```
Advanced.FMI.IncludeSaveTotal=true
```

The default value of the flag is `false`.

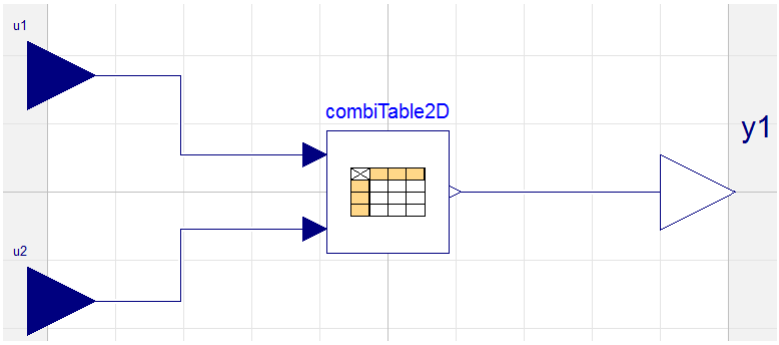
String parameter support - examples

String parameters are supported in FMUs if the option **Export with string parameters** is selected (see the setting above).

Basic example

String parameter support can be illustrated by a simple example of changing tables for an FMU; consider creating a simple model for linearization.

Create a model; drag an instance of `Modelica.Blocks.Tables.CombiTable2D` into the model. Connect the two inputs and the output and create the corresponding connectors. The result is:

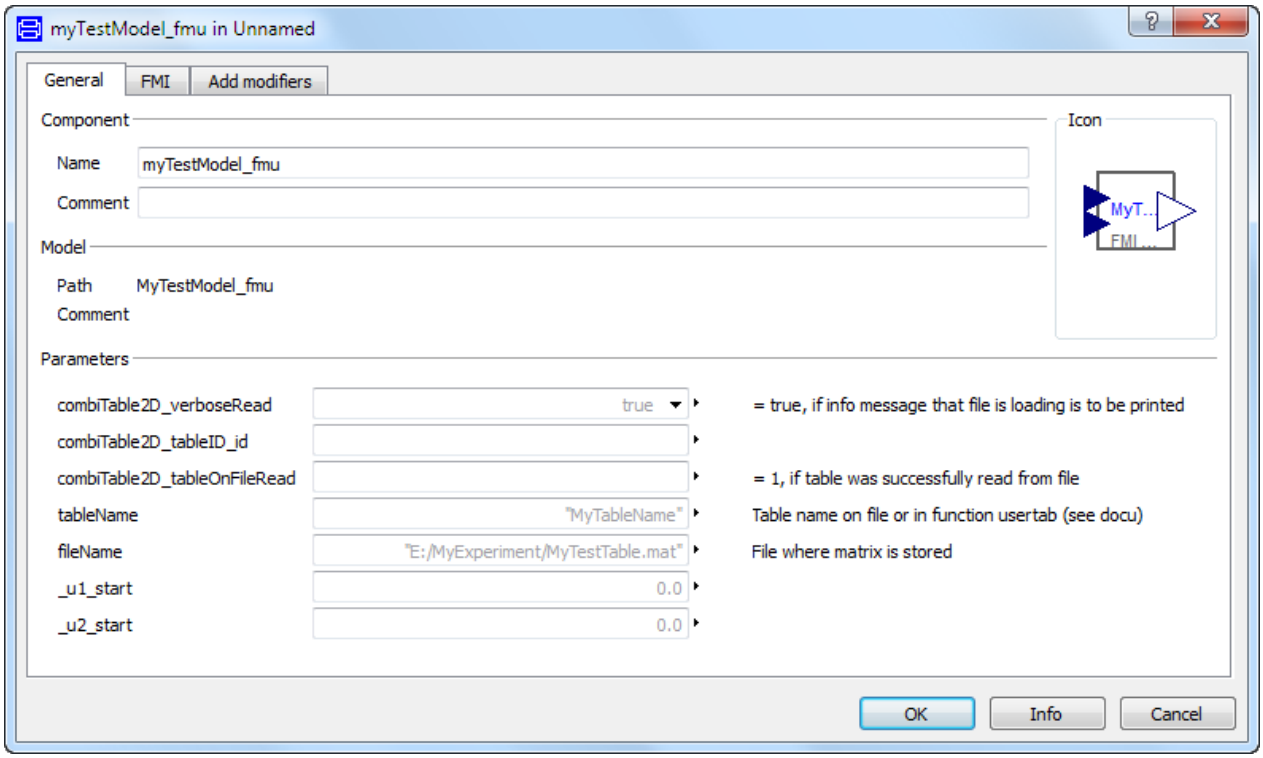


In the parameter dialog of combiTable2D, select **tableOnFile** to true, and propagate **tableName** and **fileName**. Give relevant default values for them. As an example, looking at the resulting Modelica code when having specified a table name and file name as default value, we find:

```
model MyTestModel

  parameter String tableName="MyTableName"
    "Table name on file or in function usertab (see docu)";
  parameter String fileName="E:/MyExperiment/MyTestTable.mat"
    "File where matrix is stored";
equation
  a
end MyTestModel;
```

Saving the model, and then generating an FMU from it (do not forget to set the flag above), we can import this FMU and look at the resulting parameter dialog of an instance of that FMU:



This FMU supports changing the table name and file name as string parameters.

Extended example (resource handling)

If the FMU should contain the table as a resource, the following can be done:

Rename the parameter **fileName** to **includeFileInFMU** (really not needed, but for clarity). Use, in the variable definition dialog of **includeFileInFMU**, in the default value input field, the

context command **Insert Function Call...** to access `Modelica.Utilities.Files.loadResource`, and specify the file name. (Note that the URI called by the function must be a string literal or a string parameter.) The resulting code is (given a new model MyTestModel2 is created):

```
model MyTestModel2

  parameter String tableName="MyTableName"
    "Table name on file or in function usertab (see docu)";

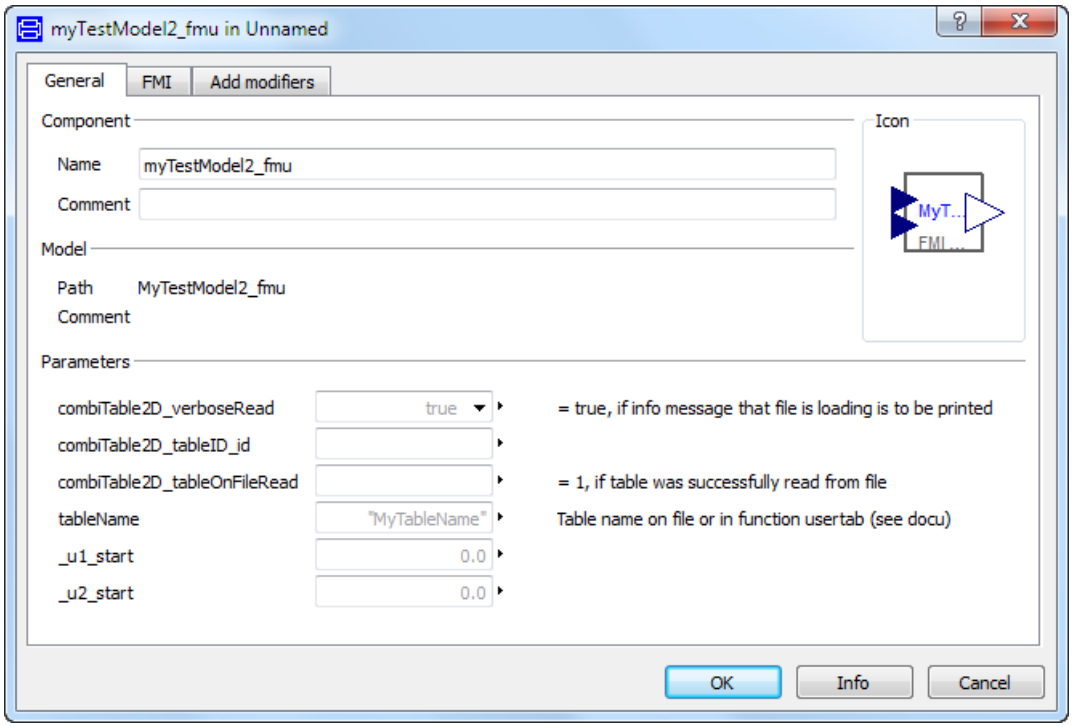
  parameter String includeFileInFMU=Modelica.Utilities.Files.loadResource("E:/MyExperiment/MyTestTable.mat")
    "File where matrix is stored";
equation

  a
end MyTestModel2;
```

Save the model. Before generating the FMU, check:

- that `Advanced.AllowStringParameters=true`.
- that **Copy resources to FMU** is ticked in the **FMI** tab of the simulation setup.

We can import the generated FMU and look at the resulting parameter dialog of an instance of that FMU:



The **includeFileInFMU** parameter is not displayed, it is evaluated, and the corresponding file has been copied to the Resources directory of the FMU.

Handling multiple FMUs

An extra source code file `all.c` is provided; it includes all other C files. This file is needed to compile all FMUs source code as one unit, which in turn is required because the demand that all internal functions and symbols needs to be static to be able to combine several source code FMUs.

The only disadvantage compiling this file instead of the separate C files is that any modification in any source code file requires re-compilation of everything.

To avoid multiple instances of this file, you can select to create it with a unique id (modelidentifier) appended to the name. As an example, if your model identifier is `MyModel`, the extra source code file name will be `all_MyModel.c`. You activate this feature by setting the flag

```
Advanced.FMI.FMUSourceCodeUniqueNaming = true
```

The flag is by default `false`. The flag is stored between sessions.

Multiple instantiation of the same FMU

FMUs generated by Dymola 2016 and later support multiple instantiation. This means that the same FMU can be used several times in the same model.

The generated XML file indicates that the model can be instantiated multiple times.

Restrictions:

- Multiple instances are currently only supported for Co-simulation with Cvode, see next section.
- The support for multiple instances has a runtime cost, you can for this reason disable the support for multiple instances by setting the flag `Advanced.AllowMultipleInstances=false`. (This flag is by default true.)
- The old table handling, corresponding to tables in previous versions for Modelica Standard Library (3.2 or older) is not supported. If you have user models with such old table handling, those must be updated to use this feature.

Notes on Co-Simulation

Note that all Dymola solvers are supported for FMU Co-simulation export (if the Binary Model Export license is available); however, the CVODE solver can be selected as a particular solver by any export type selection containing **Co-simulation using Cvode**. The support for features is currently larger when selecting CVODE as a particular solver this way than when selecting **Co-simulation using Dymola solvers**:

- Including source code is currently only supported when selecting **Co-simulation using Cvode**.
- Multiple instances are currently only supported when selecting **Co-simulation using Cvode**.

CVODE solver

The SUNDIALS suite of numerical solvers (version 2.6.2) can be used in the co-simulation FMUs. The SUNDIALS CVODE solver with Backward Differentiation Formula (BDF) and Newton iteration can be used as solver in the exported co-simulation FMUs. For further details, visit

https://computation.llnl.gov/casc/sundials/main.html

Fixed-step embedded (inline) solvers for FMU Co-Simulation export

The Dymola inline integration solvers are supported also for FMU Co-Simulation export.

For source code export, you may set the flag

```
#define ONLY_INCLUDE_INLINE_INTEGRATION
```

in the header file `conf.h`.

This will exclude Cvode from the compilation, as it is not used for inline FMUs.

If you export your inlined model as model-exchange FMU, the states will be discretized and not visible in the `modelDescription.xml`, in addition you will be required to apply a fixed step solver with the same step size as the inline solver is using.

Support for optional FMI Export options

Support for optional FMI Export options in FMI 2.0

The following tables list Dymola support for optional export options in FMI 2.0. Since both “True” and “False” can be a limitation, the cells are color-coded: green means “underlying feature supported in Dymola”, yellow means “underlying feature not supported in Dymola”. Furthermore, capital letters are used for “underlying feature supported”.

The order of the features is the order they appear in the specification. See next page; the tables are on the same page for comparison reasons.

Optional FMI 2.0 features	Model Exchange	Model Exchange with inline integration	Co-simulation using Cvode	Co-simulation with inline integration	Co-simulation using Dymola solvers
needsExecutionTool	FALSE	FALSE	FALSE	FALSE	FALSE
completedIntegratorStepNotNeeded	false	false	NA	NA	NA
canBeInstantiatedOnlyOncePerProcess	FALSE	FALSE	FALSE	FALSE	true
canNotUseMemoryManagementFunctions	FALSE	FALSE	FALSE	FALSE	true
canGetAndSetFMUState	TRUE	TRUE	TRUE	TRUE	Partly[18]
canSerializeFMUState	TRUE	TRUE	TRUE	TRUE	Partly[19]
providesDirectionalDerivative	TRUE	TRUE	TRUE	TRUE	TRUE
canHandleVariableCommunicationStepSize	NA	NA	TRUE	TRUE	TRUE
canInterpolateInputs	NA	NA	TRUE	false	TRUE
maxOutputDerivativeOrder	NA	NA	1	0	1
canRunAsynchronuously	NA	NA	false	false	false

Support for optional FMI Export options in FMI 1.0

Optional FMI 1.0 Co-simulation features	Co-simulation using Cvode	Co-simulation with inline integration	Co-simulation using Dymola solvers
canHandleVariableCommunicationStepSize	YES	YES	YES
canHandleEvents	YES	YES	YES
canRejectSteps	false	false	false
canInterpolateInputs	YES	false	YES
maxOutputDerivativeOrder	1	0	1
canRunAsynchronuously	false	false	false
canSignalEvents	false	false	false
canBeInstantiatedOnlyOncePerProcess	FALSE	FALSE	true
canNotUseMemoryManagementFunctions	FALSE	FALSE	true

[18] Supported except for the Dymola solver Cvode.
[19] Supported except for the Dymola solver Cvode.

Propagating annotations from Modelica variables to the FMI model description

Dymola supports propagating annotations from Modelica variables to the fmi2Annotation node "Annotations" in the corresponding scalar variables in an FMI 2.0 `modelDescription.xml` document.

To activate this feature, set the flag

```
Advanced.FMI2.OutputVariableAnnotationsInXML = true;
```

The flag is by default false.

FMU export from Linux

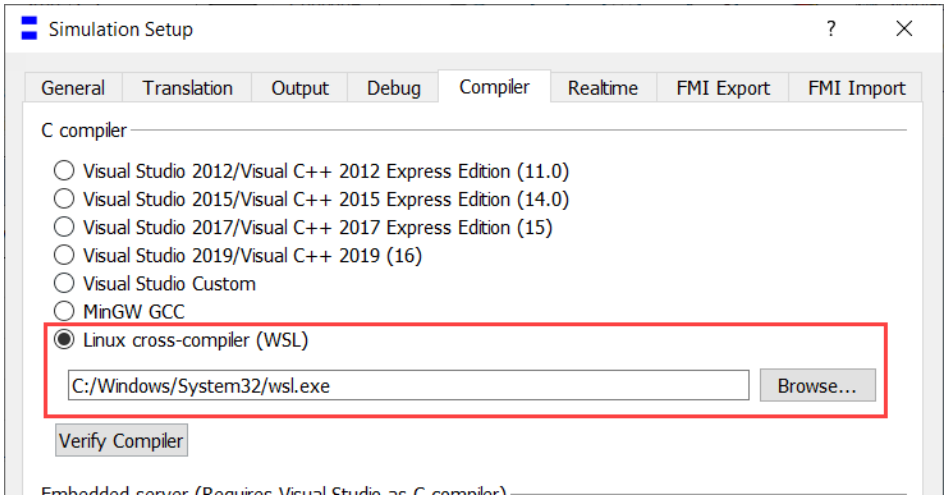
The FMU export from Linux requires the Linux utility "zip". If not already installed, please install using your packaging manager (e.g. apt-get) or see e.g. http://www.info-zip.org.

FMU export for multiple platforms

Prerequisites

Dymola on Windows supports cross-compilation for Linux via use of Windows Subsystem for Linux (WSL). The default WSL-setup is 64-bit only and Dymola adopts to this limitation.

To use this feature, you must enable WSL and install needed software components, including a suitable Linux distribution. You can then activate **Linux cross-compiler (WSL)** in the simulation setup, reached by the command **Simulation > Setup**, the **Compiler** tab, and test it using **Verify Compiler**:



For more information, and an example, how to install the above, see the index entry "*FMU export for multiple platforms*".

Activating FMU export for multiple platforms

Dymola partially supports cross-compilation when exporting FMUs on Windows. Is partial in the sense that your main compiler environment is limited to Visual Studio or MinGW and the cross-compiler environment is limited to WSL. So first, you need to select either Visual Studio or MinGW in the simulation setup in the figure above. Then, to activate cross-compilation, activate, in the **FMI Export** tab of the simulation setup, the setting **Cross compile for Linux**. Now when exporting and FMU, you will get 64-bit Linux binaries in addition to Windows binaries.

Important! In this case, you should *not* select **Linux cross-compiler (WSL)**; the setting **Cross compile for Linux** sets the suitable cross-compilation.

The setting **Cross compile for Linux** is also available in the dialog that appears when you export an FMU on Windows.

Activating the setting corresponds to setting the flag `Advanced.FMI.CrossExport=true`. (The default value of the flag is `false`.)

Note that the value of the setting/flag is not saved between sessions.

Limitations

- The value `meUndefinedValueReference` is never returned when value references are requested. Therefore, some value references returned may not be present in the model description file.
- The result file generation is currently only fully supported for the traditional solvers (Lsodar, Dassl, Euler, Rkfix2, Rkfix3, and Rkfix4) when importing the FMU in Dymola. For the other solvers, the number of result points stored will typically be too low. However, the values are accurate for the time-points at which they are computed.
- String variables cannot be used in models that are exported as FMUs. String parameters are however supported.

10.1.3 Importing FMUs in Dymola

FMU import by the built-in function importFMU

The Dymola FMU import consists of (1) unzipping the .fmu archive, (2) transforming the XML model description into Modelica, and (3) opening the resulting Modelica model in Dymola.

Importing FMU models to Dymola is achieved by the function

```
importFMU(fileName, includeAllVariables, integrate,
promptReplacement, packageName, includeVariables[:])
```

The input string `fileName` is the FMU file (*with* the .fmu extension).

The parameter `includeAllVariables` is related to the parameter `includeVariables[:]` (see this parameter below). Only if the array of `includeVariables[:]` is an empty array, `includeAllVariables` is used. If this is the case, by setting the parameter `includeAllVariables` to false, only inputs, outputs and parameters from the model

description are included when generating the Modelica model. Such black-box import can be used as separate compilation of models to substantially reduce translation times. For large model this is recommended since the generated .mo file otherwise becomes huge and will take long time for Dymola to parse and instantiate.

The parameter `integrate` controls if integration is done centralized or in the FMU, i.e. `integrate=true` means import the Model Exchange part of the FMU and `integrate=false` means use the Co-Simulation part of the FMU. By default, this parameter is true. This setting is only relevant if the FMU to import supports both types. Otherwise, this setting is silently ignored. If the Co-Simulation part is used, the macro step-size can be set as the parameter `fmi_CommunicationStepSize` in the FMI tab of the parameter dialog of the imported FMU. See also section "Settings of the imported FMU" on page 1279.

The parameter `promptReplacement` can be set to true to generate prompting before replacement of any existing Modelica model being the result of a previous import. Having no prompting is useful when repeatedly importing FMUs using scripting. By default, this parameter is false.

The string parameter `packageName` can be set to the package to where the FMU should be imported. The package must be open in Dymola when importing.

The array parameter `includeVariables[:]` is by default an empty array, but variables can be added in this array. If variables are added in this array, these variables will be included in the generated FMU. In that case, the value of the parameter `includeAllVariables` is ignored. You can add variables to import in the dialog for importing an FMU by the selection **These variables**, see section "Filtering signals when importing an FMU" on page 1277.

The function `importFMU` outputs `true` if successful, `false` otherwise.

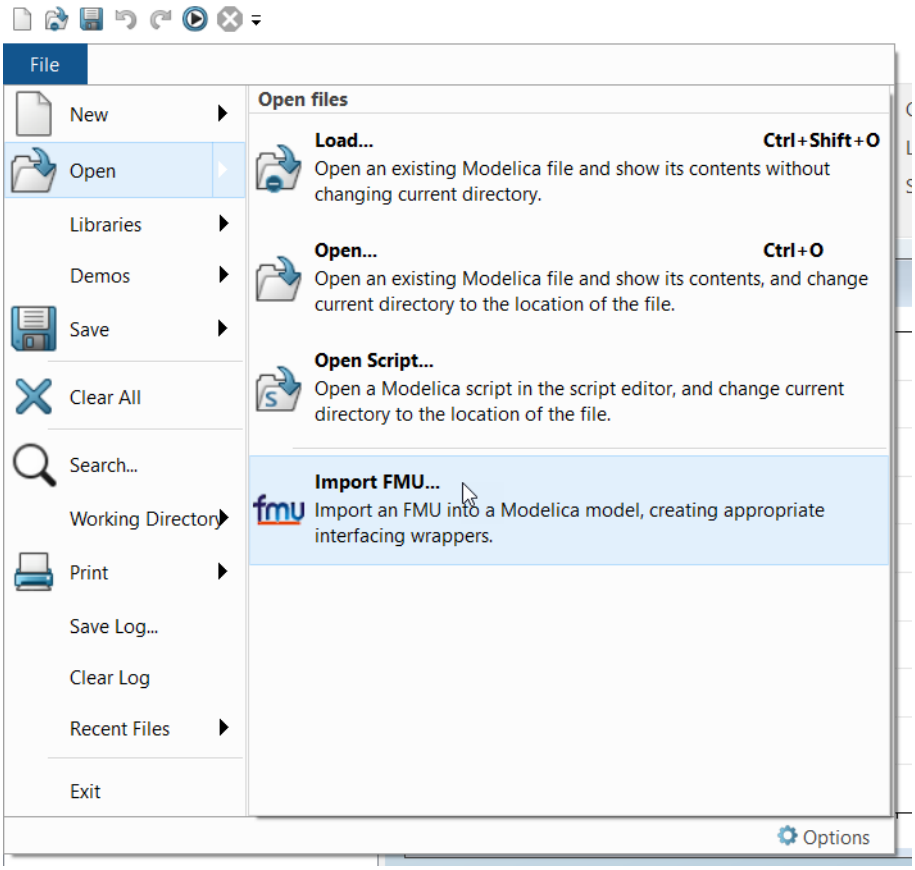
The generated Modelica file will get the name `model_fmu.mo` or `model_fmu_black_box.mo`, depending on the value of `includeAllVariables`.

ASCII characters of values larger than 32 are supported in the xml file of the imported FMU. Also UTF characters are supported, but not recommended.

Note: The binary library files from any previous import are replaced when calling `importFMU` and thus translations of previously imported models are not guaranteed to work any longer (in the unlikely event of a name clash).

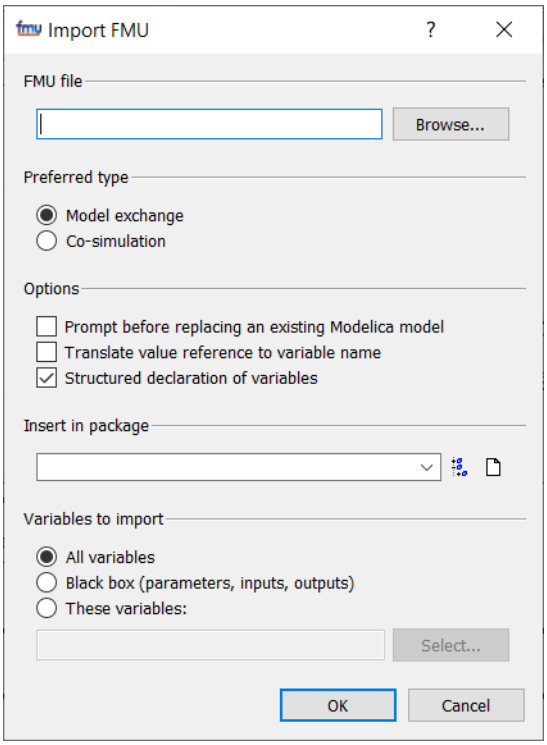
FMU import by command

An alternative to executing the `importFMU` function is to use the command **File > Open > Import FMU...**.



Note that this command also will be automatically applied on an .fmu file by dragging it into the Dymola main window.

What settings will be used when using any of the above commands is specified in a dialog that appears when applying any of the commands:

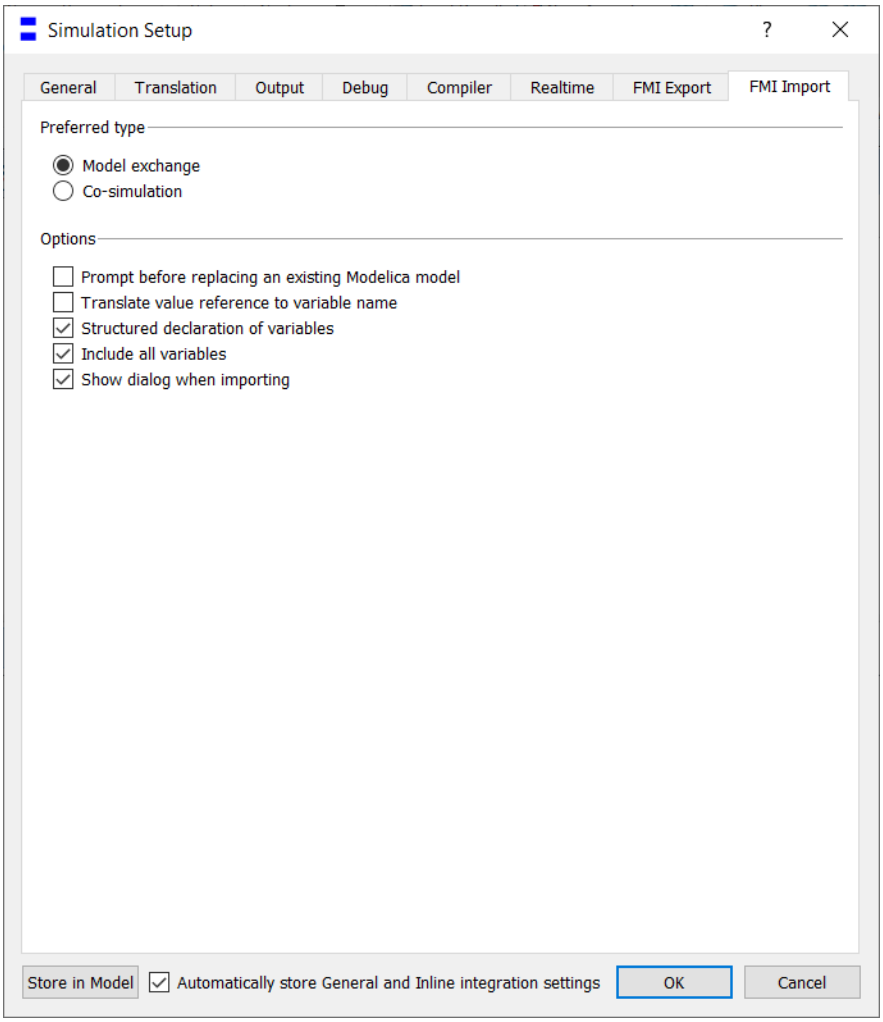


Specifying the FMU file to import

The first part of the dialog above enables you to browse for the FMU file to import. When the FMU is dragged into Dymola, the path is prefilled.

Selecting preferred FMU type and options

The settings in the **Preferred type** group and the **Options** group corresponds, with two exceptions, to the settings in the **FMI Import** tab of the simulation setup, reached by the command **Simulation > Setup**, the **FMI Import** tab:



Changing settings when importing will affect also this menu. Changed settings are remembered in the session, but not between sessions.

Preferred type can be selected as **Model exchange** or **Co-simulation**. This setting is only relevant if the FMU to import supports both types. Otherwise, this setting is silently ignored. This setting corresponds to the parameter `integrate` in `importFMU` (see above for description). There is a corresponding flag `Advanced.FMI.Integrate` that has the same alternatives as the `integrate` parameter.

In the **FMI Import** tab, five options are available; the first three are also present in the dialog for importing an FMU:

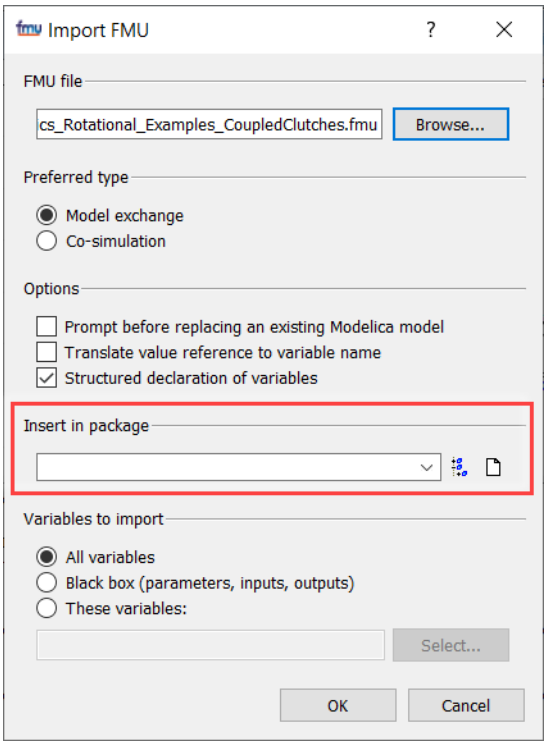
- **Prompt before replacing an existing Modelica model** – corresponds to the function parameter `promptReplacement` (see above). There is a corresponding flag

`Advanced.FMI.PromptReplacement` that has the same alternatives as `promptReplacement`.

- **Translate value reference to variable name** – this option is not present in `importFMU`. If ticked, the imported FMU will contain a translation from value references to variable names. This is useful for debugging, however will decrease the performance. There is a corresponding flag `Advanced.FMI.TranslatedLoggedValueReferences` with the default value `false`.
- **Structured declaration of variables** – this option is not present in `importFMU`. If ticked, (the default value) the variables of the imported FMU will be presented in a hierarchical structure, that is, as records. This is useful when e.g. wanting to change variable values. To be able to use this option, the attribute `variableNamingConvention` in the model description file of the FMU to be imported must be set to `variableNamingConvention="structured"`. There is a corresponding flag `Advanced.FMI.structuralDeclaration` with the default value `true`.
- **Include all variables** – corresponds to the function parameter `includeAllVariables` (see above). There is a corresponding flag `Advanced.FMI.IncludeAllVariables` that has the same alternatives as `includeAllVariables`. ***Important!*** This setting is ignored if you select specific variables to include in the FMU using the alternative **These variables**, see section “Filtering signals when importing an FMU” on page 1277
- **Show dialog when importing** – this option is by default ticked. If unticked, the Import FMU dialog is not displayed when importing FMUs. This setting is only present in the **FMI Import** tab.

Selecting the package where the imported FMU should be inserted

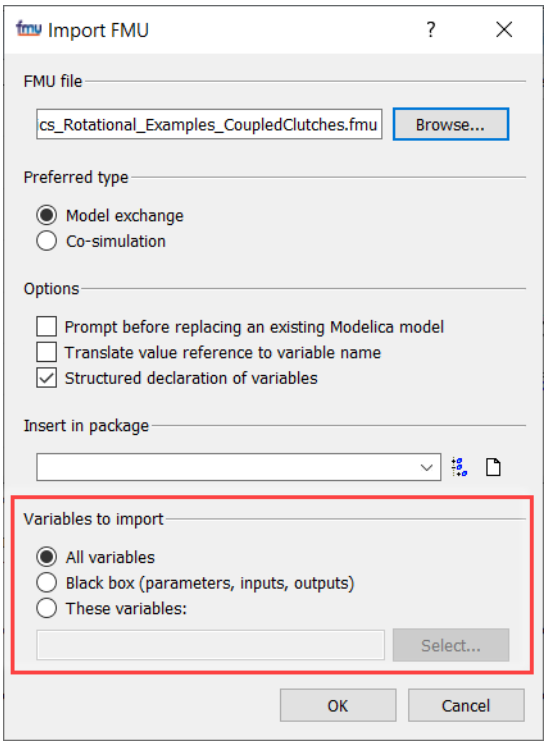
You can specify where the imported FMU should be inserted:



You can use the arrow to select from current packages, you can also browse the trees in the package browser, and you can also create a new package.

Filtering signals when importing an FMU

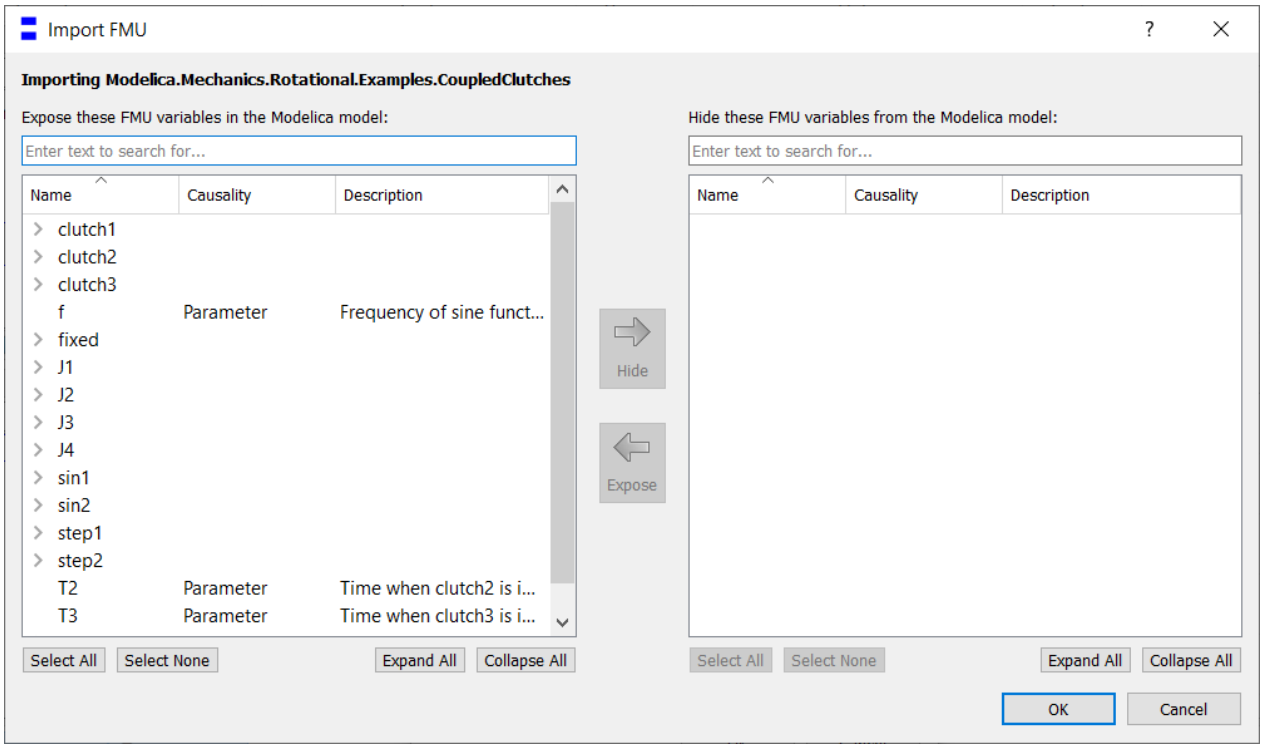
You can filter signals when importing an FMU. An example of dialog, with default settings, to import an FMU created from the demo Coupled Clutches:



There are three alternatives:

- **All variables**: This corresponds to the option **Include all variables** in the **FMI Import** tab of the simulation setup being activated (see above). This setting corresponds to the parameter `includeAllVariables=true` in the built-in function `importFMU`. This is the default setting.
- **Black box (parameters, inputs, outputs)**: This corresponds to the option **Include all variables** in the **FMI Import** tab being deactivated. This setting corresponds to the parameter `includeAllVariables=false` in the built-in function `importFMU`.
- **These variables:** Selecting this alternative enables you to select in detail what variables to expose or hide from the FMU. Selecting this alternative, you can then click **Select…** to have a dialog for what variables to select, and you will have the resulting variables in the variable field.

If you select **These variables:** and then click **Select…**, an example of the dialog that appears may be (in the case the demo Coupled Clutches is to be imported as an FMU):



You select the variables you want to move to the other pane, and then click the arrow **Hide** or **Expose** depending on in which pane you want to have the variables, if they should be exposed or hidden.

For each pane, you have a search field on top, and four buttons at the bottom: **Select All**, **Select None**, **Expand All**, and **Collapse All**. Note that it might be handy to, in the above figure, to click **Select All** and then click the **Hide** arrow, if you want to select which variables to expose rather than the ones to hide.

The filtering can also be used in scripting, the selected variables will be added in the array of the `includeVariables[:]` parameter of the built-in function `importFMU`. Note that in this case, the parameter `includeAllVariables` in `importFMU` is ignored.

Settings of the imported FMU

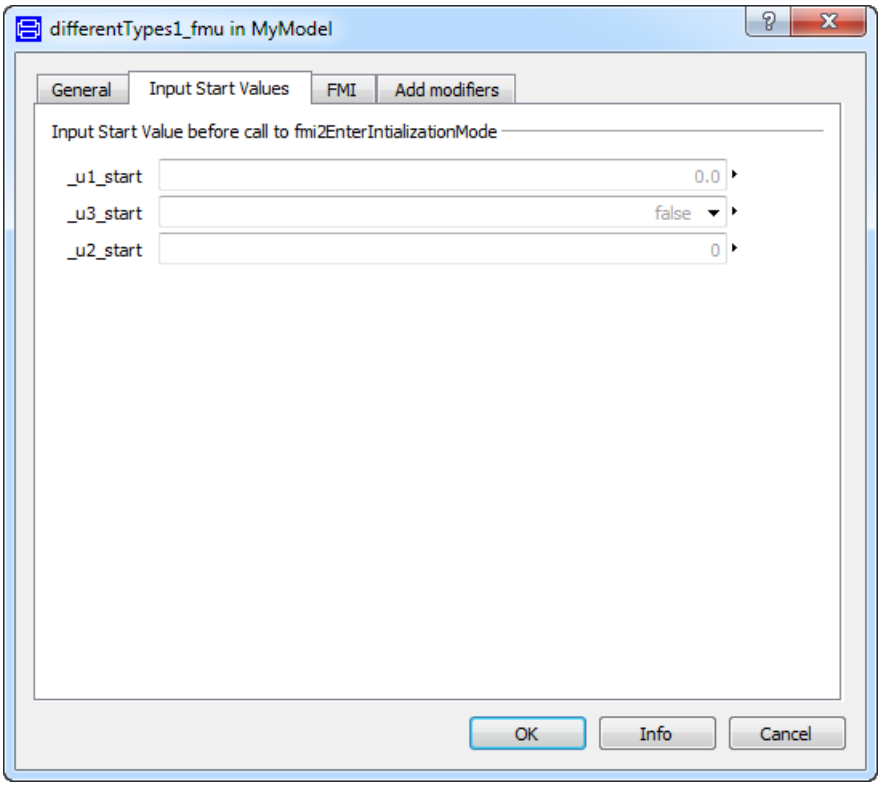
The parameter dialog of the imported and instantiated FMU contains an **Input Start Values** tab and an **FMI** tab.

Input Start Values tab

For FMI Model Exchange in FMI version 2.0, input start values can be set before initialization. This should however only be necessary if your FMU is constructed in such a way that the default start values for an input is illegal in the FMU, e.g. division with an input variable having a default value of zero. For such an input variable, you can set the input start value to

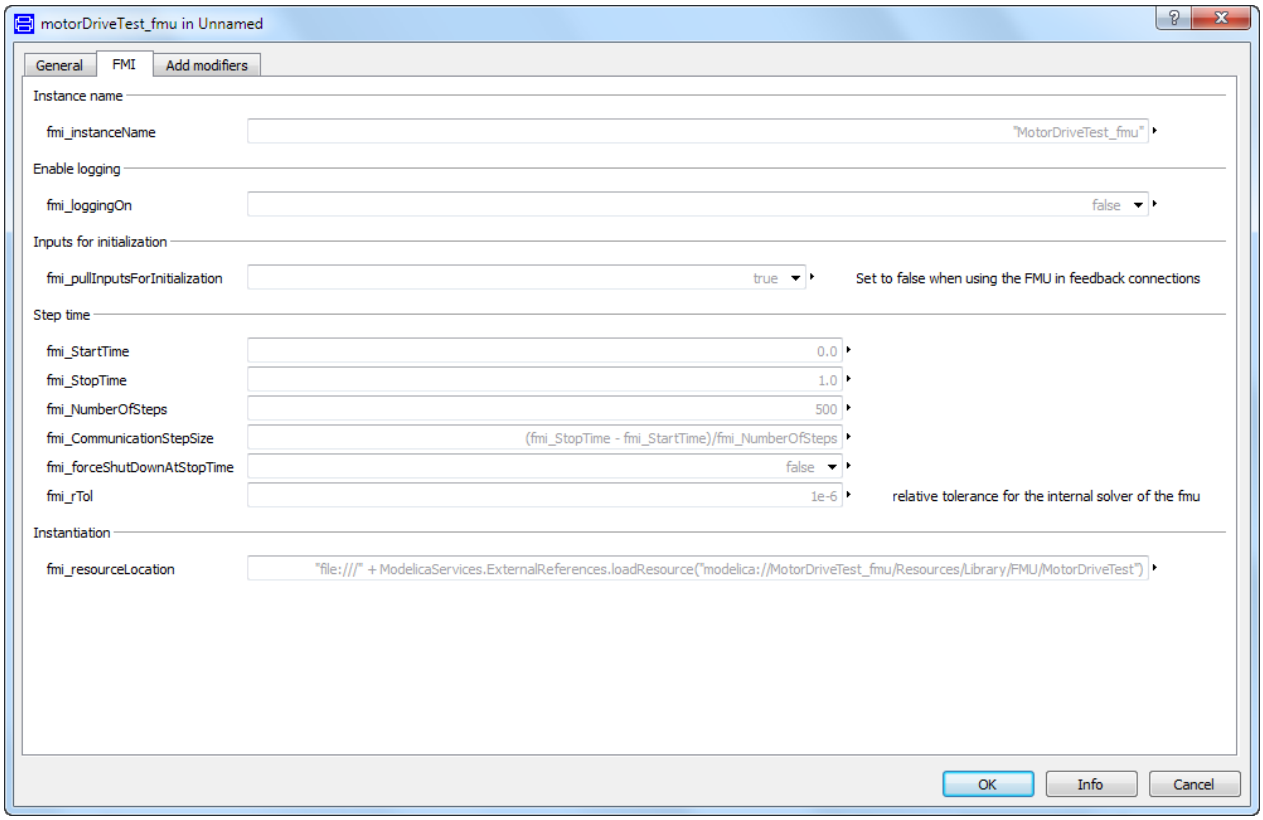
some value not being zero; sources of the FMU will then be handled properly in the initializationMode.

Such start values are collected in the **Input Start Values** tab:



For FMUs of FMI version 1.0, you should avoid a design where input values affect initialization, since the FMI 1.0 interface lacks proper support to iterate during initialization.

FMI tab

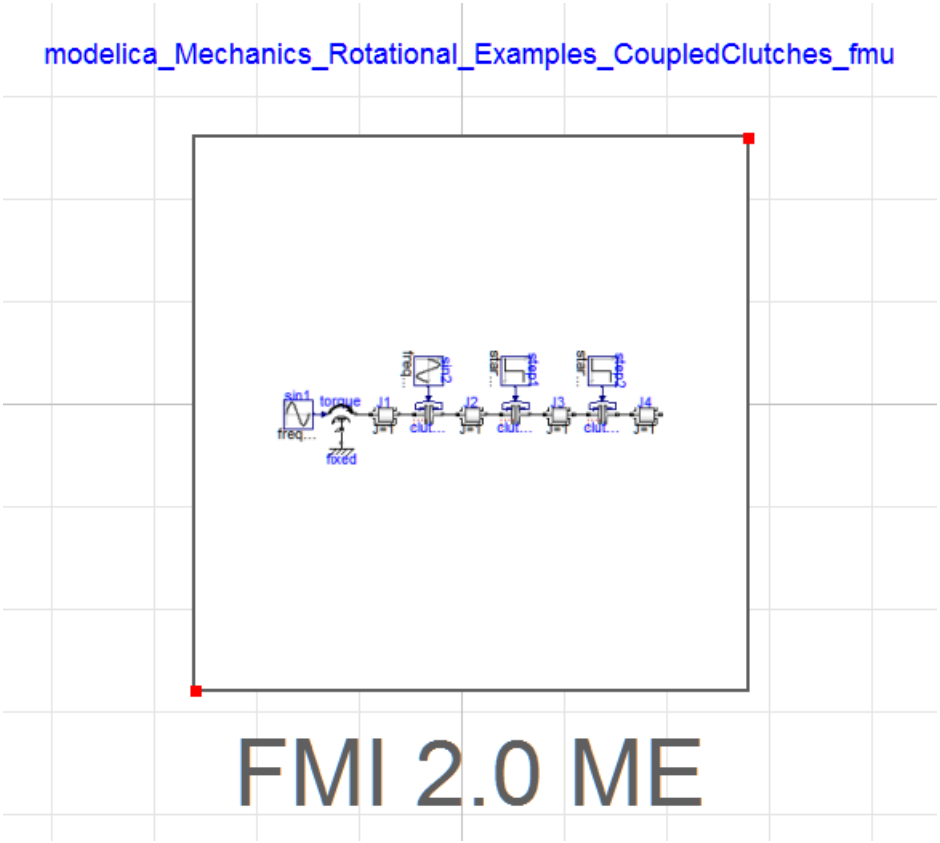


The available settings depend on the FMU type.

fmi_resourceLocation might be needed when importing FMUs from other vendors, to specify the location of external resources. (For FMI version 1.0 Co-simulation the name is **fmi_fmuLocation**.) By default, the parameter displays the location where the FMU is unpacked, which is usually the location of external resources (dlls, tables, etc,) as well.

Using the exported model image to display the imported FMU

If the exported FMU includes an exported model image `model.png`, this image is used to represent the FMU when importing it. Below, the demo Coupled Clutches has been exported with the diagram layer as the model image. The result importing it and instantiating it is:

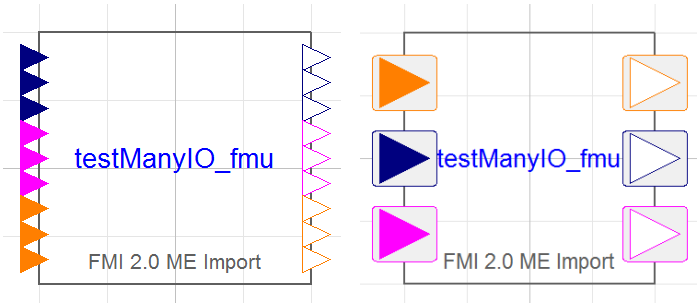


If no model image is available in the imported FMU, a standard icon is used instead:



Importing FMUs with many inputs/outputs

When importing FMUs with many inputs/outputs, the input and output connectors of the imported FMU are automatically stacked at the same location, one location for each type (Integer, Real, and Boolean) of input and output connectors (the image to the right below).



The limit of the number of connectors when stacking should be applied is defined by the flag

```
Advanced.FMI.OverlappingIOThreshold
```

The default value of the flag is 10 (so for creating the figure above, the value was set to 4).

Dragging a connection from/to a stacked connector displays a dialog to conveniently select what connectors to connect.

Import of FMUs of FMI version 1.0 and version 2.0 to the same model

Import of both FMUs of FMI version 1.0 and version 2.0 to the same model, is supported.

Input handling for co-simulation FMU import

Input time point

You can now choose the time point used as input when calling `doStep` with a co-simulation FMU from time `t0->t1`.

Your choices are input at time `t0` input at time `t1` or `pre` on input at time `t1` (default and behavior of Dymola 2017).

Input time can be chosen with the parameter `fmi_Input` time, and if `StepEnd` is chosen `fmi_UsePreOnInputSignals` can be used to disable pre operator on the input signal.

Be aware that using `StepStart` at for `fmi_inputTime` will introduce delays in output if you have direct dependencies on the input.

Using `StepEnd` and disabling `pre` can introduce algebraic loops when connecting with feedback which cannot be solved by co-simulation FMUs.

Using pre on inputs at `StepEnd` will break these loops if you create them with connections by introducing an infinite small delay.

Input derivatives

Dymola supports interpolation of input and to set the input derivatives of real inputs if the FMU has the capability flag `canInterpolateInputs`.

The interpolation is a first order interpolation. This can be activated by setting the flag `Advanced.FMI.SetInputDerivatives = true` before importing an FMU that supports this feature.

Since we only support interpolation and not extrapolation, similar restrictions exist as when using `StepEnd` as input time and disabling pre, i.e. all FMUs in a feedback loop cannot have this feature at the same time.

Improved fmi2 initialization for co-simulation

FMI 2.0 co-simulation supports initialization with algebraic loops (but not solving algebraic loops at simulation time).

Improvement has been made to make this more robust at Dymola import, this has also removed the need for the parameter `fmi_pullInputsForInitialization` and thus it been removed, (it is still needed for some cases in FMI 1.0 co-simulation as you cannot solve initialization of multiple FMUs there).

Default value of the integration tolerance for FMI 2.0 Co-simulation FMU

In previous Dymola versions (up to and including 2022x), the default value of the integration tolerance of an imported FMU could be specified by the parameter `fmi_rTol` which was sent into function `fmi2SetupExperiment`. The default value used vas 1e-6.

From Dymola 2023, you can select what default value to use by the new parameter `fmi_setTol`. The alternatives are:

- `fmi_setTol` is `false` (default value) - The value of the integration tolerance in the exported FMU is used as default integration tolerance. (The integrator tolerance in an exported FMU from Dymola is usually taken from the simulation setup.)
- `fmi_setTol` is set to `true` - The parameter `fmi_rTol` is used to set the default value of the integration tolerance of the imported. The default value of `fmi_rTol` is the value given in the attribute `tolerance` of `defaultExperiment` in the `modeldescription.xml`, if this attribute exists. If it does not exist, the value 1e-4 is used.

Enhanced functionality when importing Co-simulation FMUs supporting fmi2GetDirectionalDerivative

If you import a Co-simulation FMU and it supports the function `fmi2GetDirectionalDerivative`, Dymola will generate parameter arrays listing the input, output, exposed state, and exposed state derivative value reference and names.

In addition there is a function for the FMU, `linearizeFMU` that will take the valueReference as an argument and return the A,B,C,D matrices for the state space representation of the linearized system by calling `fmi2GetDirectionalDerivative` on the FMU.

If the flag `Advanced.FMI.LinearizeCSFMU` is set to `true` when you import the FMU, you will get a parameter `fmi_linearizeTime` that will allow you to select a time when the linearize function will be called. The function assigns the A,B,C,D matrices to the variables `fmi_AMatrix`, `fmi_BMatrix`, `fmi_CMatrix`, and `fmi_DMatrix`, respectively, at that given time.

Translation of underscore

The default (in Dymola 2017 and later) is to translate underscore "_" without any changes when importing an FMU. If you want underscore "_" to be translated to "_0" when you import an FMU, you can set the flag

```
Advanced.FMI.UseTrueVariableNames = false;
```

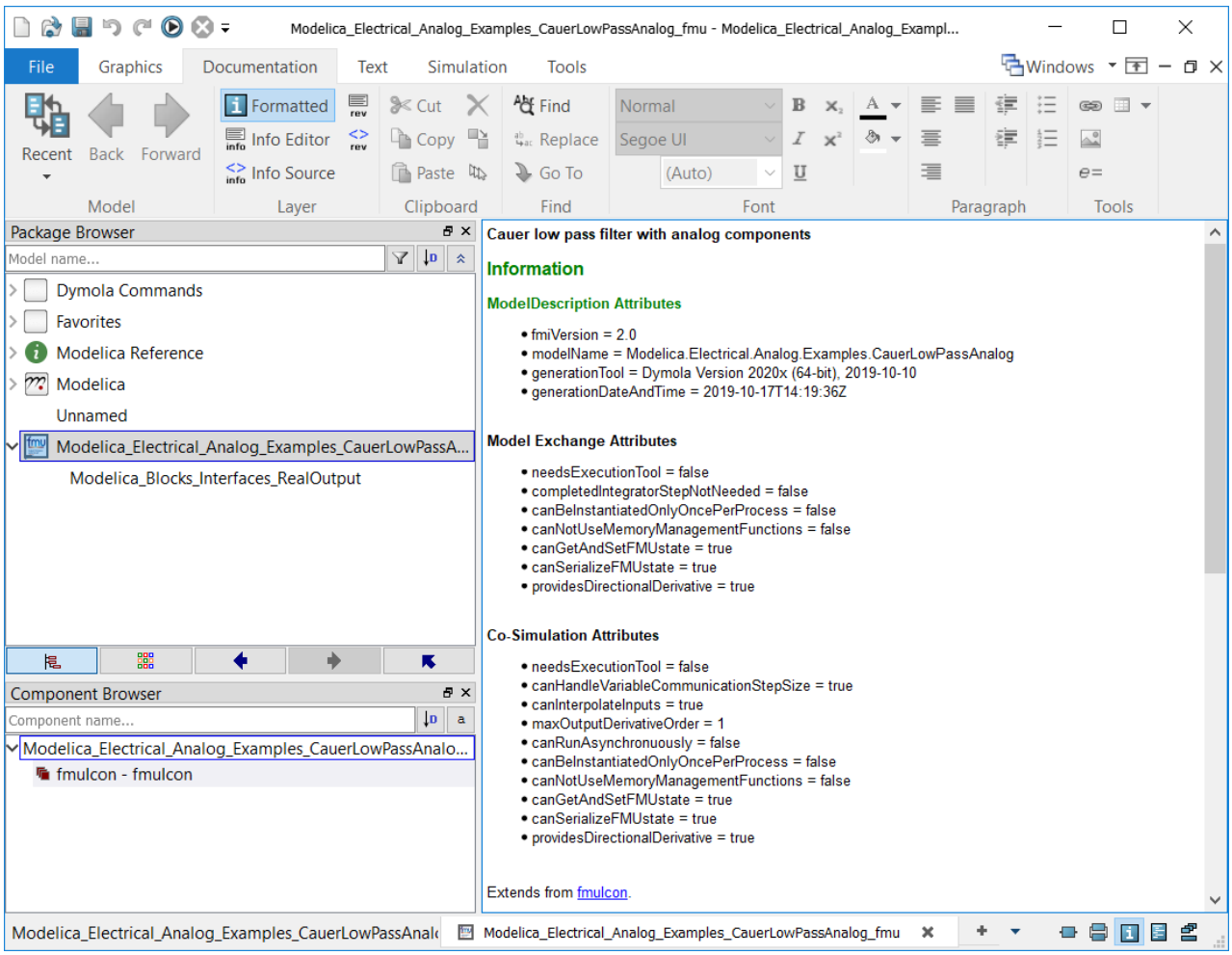
Previously the default value was to always translate underscore "_" to "_0" because of possible conflicting names (the period "." in Modelica paths is always translated to underscore when importing an FMU). Now, when structured variables are used by default when importing an FMU, the likeliness for conflicting names is very small, hence the changed default behavior, and the flag to revert to the old behavior.

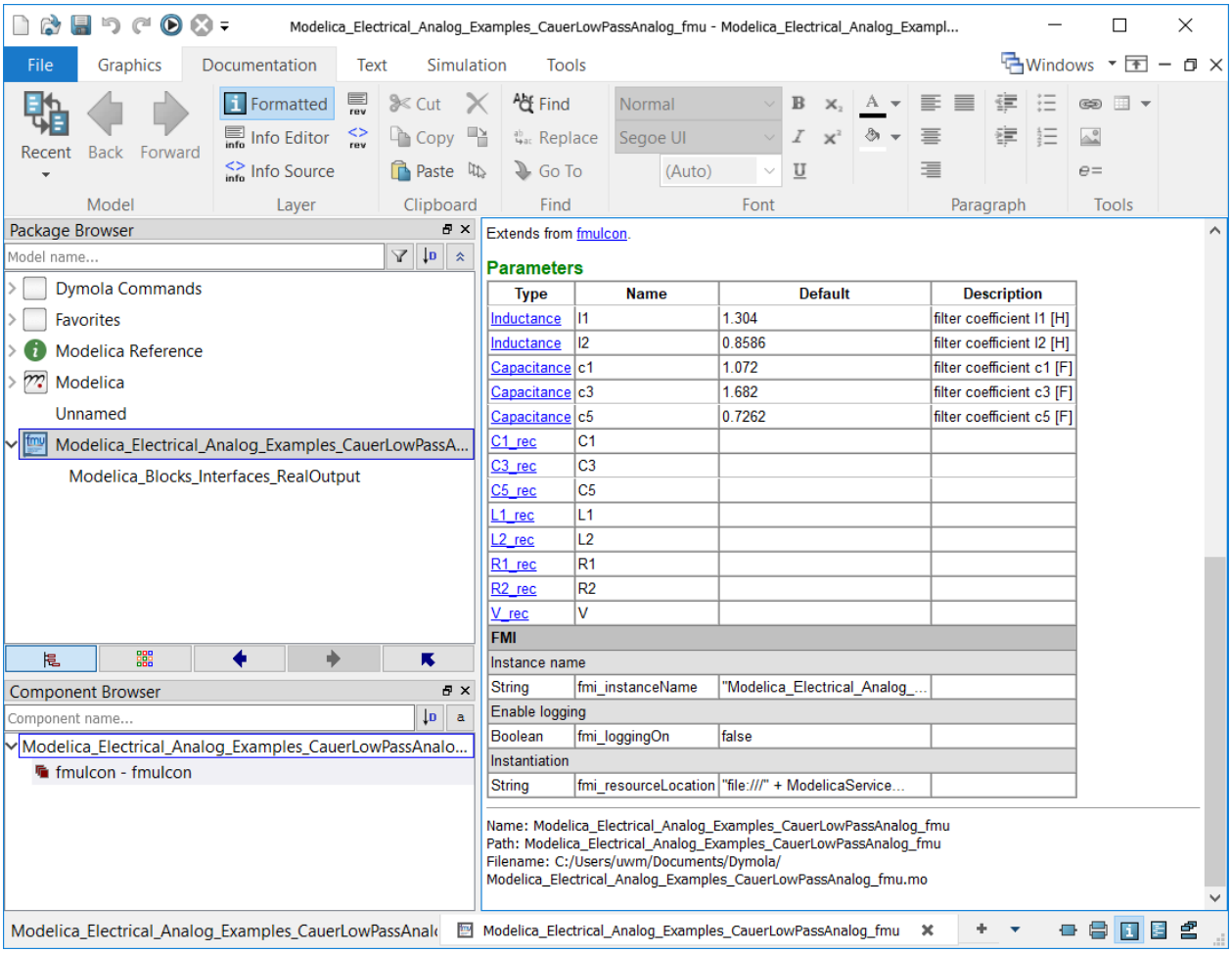
Note! If you have a model that contains an FMU as a connected component, you might get errors if you want to reimport the FMU to the model, due to the changed translation of

underscore. You need in such a case to either redo your connections or set the above flag to false to have the old naming convention when reimporting the FMU:

Display of information for an imported FMU

Information from the modelDescription.xml file of an imported FMU is displayed in the information layer of the imported FMU (two images presented below to be able to display the whole information layer):





Unit handling

FMI version 2.0 supports unit handling where an FMU exporter can define any unit for inputs and outputs as long as conversion to base units according to the FMI standard is available. This allows for proper unit checking for inputs and outputs between FMUs.

Dymola supports this; units can be automatically converted to base units for inputs and outputs of imported FMUs. Such unit handling for parameters in FMUs is also supported.

The flag `Advanced.FMI.ConvertUnits` is an integer flag that specifies when nonstandard Modelica units should be converted to their declared base units. The possible value of the flag are:

- 0 – Don´t convert any units.
- 1 – (Default) Convert nonstandard units with factor 1 and offset 0.

- 2 – Convert all nonstandard units. Units with factor !=1 and offset !=0 will have a different value in the Modelica wrapper.

To log all units with skipped unit conversion, you can set the flag `Advanced.FMI.LogAllIgnoredUnitConversions = true`. By default, the flag is `false`.

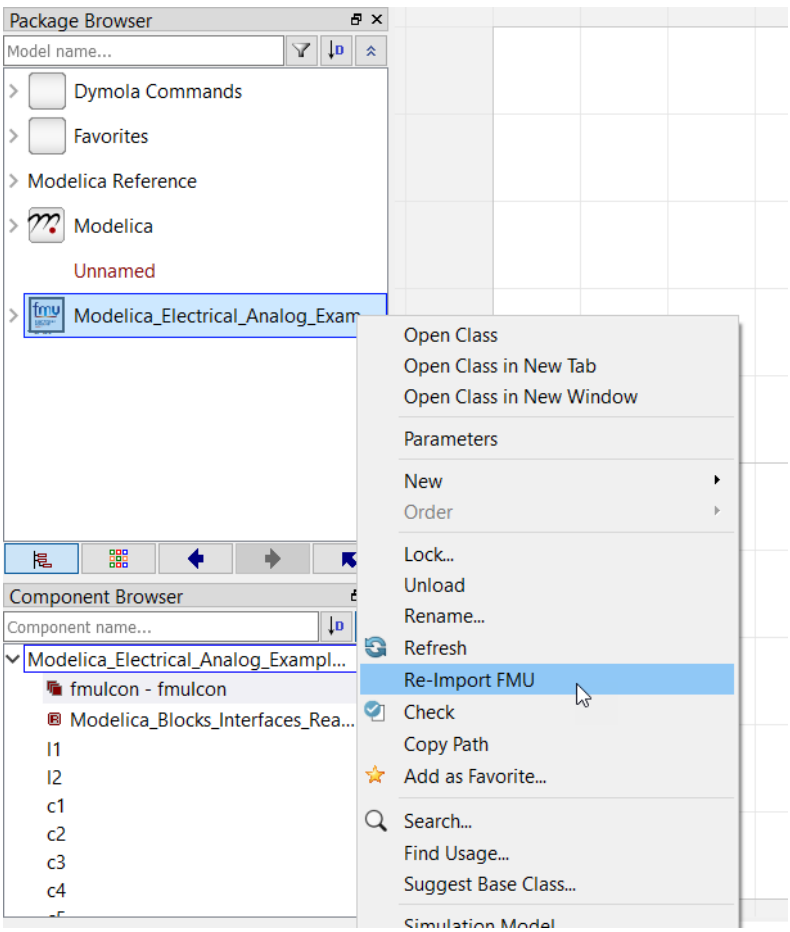
(If you want to skip the unit declarations completely, you can set the flag

```
Advanced.FMI.DoNotDeclareUnits = true;
```

Setting this flag means units are not declared. The flag is by default `false`.)

Refresh of FMUs

FMU components can be refreshed by using the FMU context command **Re-import FMU**:

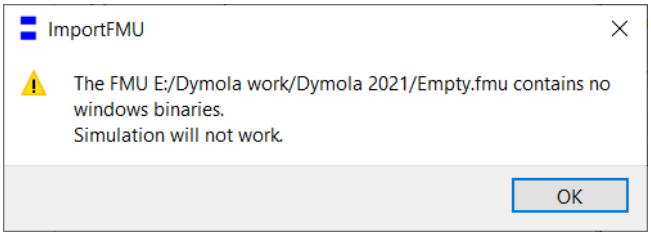


There is no scripting support for this functionality.

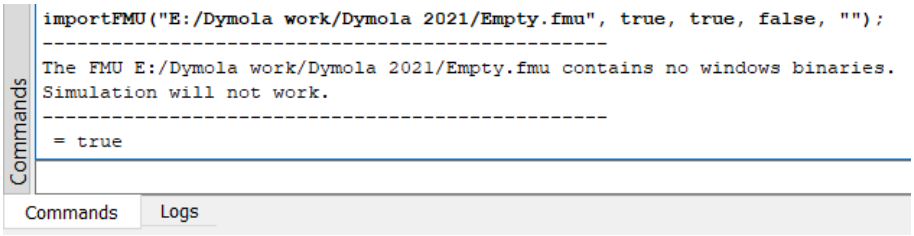
Check if Windows binaries are available when importing an FMU in Windows

When you import an FMU in Windows that contains no Windows binaries, you will get a warning.

The warning when importing by the command **File > Open > Import FMU...** or dragging the FMU into the Dymola window:



The warning in the Command log when importing the FMU by scripting:



FMU import on Linux

The FMU import on Linux requires the Linux utility “unzip”. If not already installed, please install using your packaging manager (e.g. apt-get) or see e.g. http://www.info-zip.org.

Limitations

- For FMI version 1.0, the attribute nominal for scalar variables is not supported when importing FMUs with Model Exchange. (For FMI version 2.0, this is supported.)

10.1.4 Validating FMUs from Dymola

Once the dynamic behavior of a model is verified and it is ready to be exported as FMU, one would like to verify that this behavior could be repeated on the targeted simulation environment. For model exchange, which is dependent on the solver of the target, this is naturally less straightforward than for co-simulation, where the solver is built into the FMU. We focus this discussion on the co-simulation case, although all is possible for model exchange as well.

Normally, the FMU contains inputs that need to be connected to signal generators (sources) before this validation can be commenced. Since this is model and test dependent and hard to automate, we will assume the model inputs have been connected to necessary sources

beforehand. The result is a test model with no disconnected inputs. After the validation, these sources are of course removed before the final FMU is created.

Since Dymola supports FMU import, it becomes natural to re-import the FMU in Dymola and compare its simulation with the original model. We demonstrate this for the demo model CoupledClutches. For brevity, we use a scripting perspective. First, export as FMU with, say, both model exchange and co-simulation support:

```
translateModelFMU(
  "Modelica.Mechanics.Rotational.Examples.CoupledClutches",
  false, "", "1", "all");
```

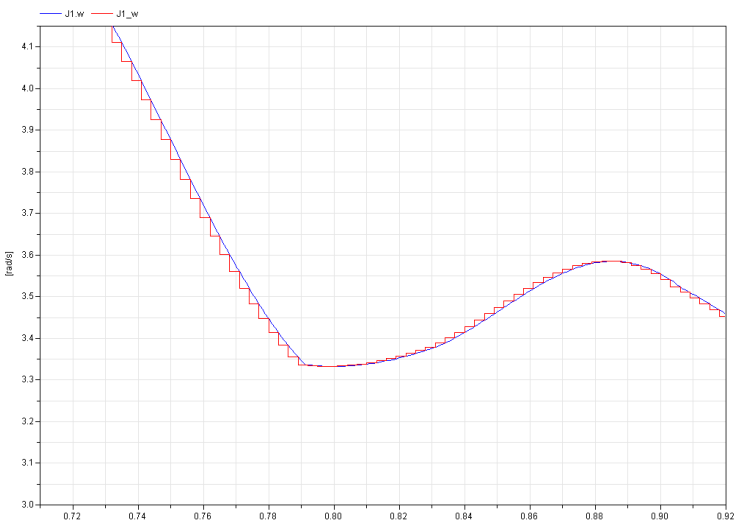
Re-import, in a non-interactive mode, the FMU for co-simulation:

```
importFMU(
  "Modelica_Mechanics_Rotational_Examples_CoupledClutches.fmu",
  true, false, false);
```

Simulate the model being the result of the import:

```
simulateModel(
  "Modelica_Mechanics_Rotational_Examples_CoupledClutches_fmu",
  stopTime=1.5, method="dassl");
```

Finally, the resulting trajectories can be plotted and compared visually with the original (non-FMU) simulation. Note that, since the imported model is flattened, the trajectory names are somewhat different; e.g., J1.w becomes J1_w:



The blue trajectory is from the reference simulation and the red is from the co-simulation. Note that the latter is rendered as constant between the sample points.

While this validation is ok for sample testing of a single model, this clearly becomes infeasible for systematic validation of several trajectories.

The remedy is a new function `validateModelAsFMU`, which automates the following steps:

- Generation of reference trajectories.
- Exporting of the FMU.
- Importing of the FMU.
- Mapping of trajectories names to those of the original model.
- Numeric comparison of trajectories.
- Graphical HTML presentation of deviating trajectories in fashion similar to the plot above.

Main features include:

- Using a default set of trajectories to compare or specifying it explicitly. The default it the set of all state candidates.
- Choosing tolerance for the comparison.
- Optional generation of reference trajectories, which is typically only needed once.
- Optional FMU export, which might not be needed each time.
- Test of co-simulation or model exchange.
- Test of FMI version 1.0 or 2.0.

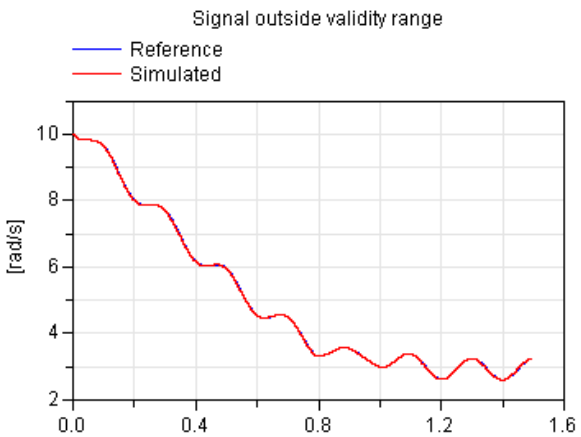
It is available in Modelica\Library under the Dymola installation.

Below call validates CoupledClutches as a co-simulation FMU for FMI 1.0:

```
validateModelAsFMU(
  "Modelica.Mechanics.Rotational.Examples.CoupledClutches");
```

An excerpt from the log file is given below:

```
Variable: J1.w has scalar criteria 0.00248651 larger than tolerance 0.001
```



In this case, we may argue that the comparison tolerance should be increased to avoid the report of this trajectory.

10.1.5 Profiling of FMUs

The feature of profiling of function calls can be used to investigate FMUs. For more information of profiling of function calls, see the chapter “Simulating a Model”, section “Improving simulation efficiency”, subsection “Profiling”.

10.1.6 FMI Kit for Simulink

FMI Kit for Simulink supports export of FMUs from Matlab/Simulink as well as import of FMUs into Matlab/Simulink.

The FMI Kit for Simulink is from Dymola 2020x not included in the Dymola distribution, you can instead access it freely from GitHub (including all source code):

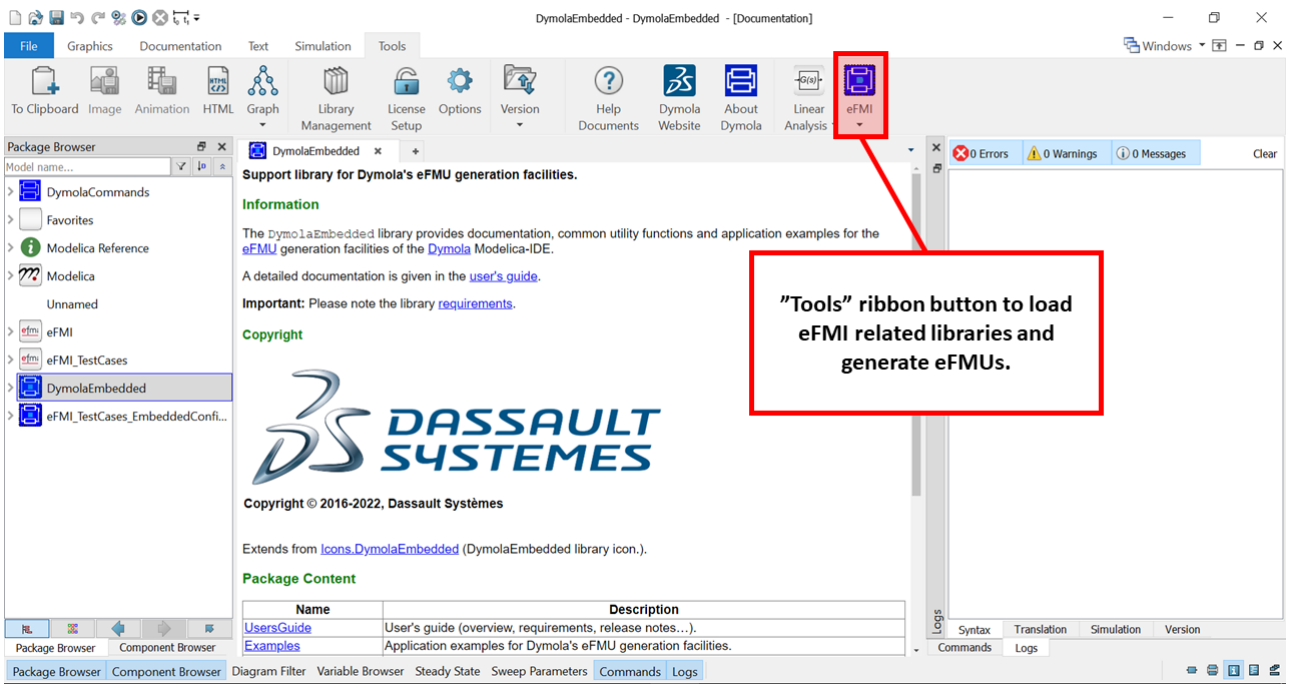
https://github.com/CATIA-Systems/FMIKit-Simulink

The support for the FMI Kit for Simulink is handled through the issue system at GitHub.

10.2 eFMI Support in Dymola

Dymola supports eFMI according to the public available eFMI Version 1.0.0-alpha.4 specification. Dymola´s eFMI facilities comprise generation of eFMUs with Algorithm Code, CATIA ESP based Production and Binary Coded containers, and the usage of generated eFMUs for Co-simulation from within Modelica models.

The **eFMI** button in Dymola´s **Tools** ribbon can be used to load Dymola´s eFMI facilities (by the menu entry **Load Libraries…**)



For eFMU generation, a Microsoft Windows Dymola installation must be used. A Dymola Source Code Export license is required.

Further requirements, limitations, and documentation – in particular how to configure the eFMU generation – can be found in the "Users guide" of the DymolaEmbedded library. Just load the library via the **Load Libraries…** menu entry of the **eFMI** button in the **Tools** ribbon.

10.3 SSP Support in Dymola

10.3.1 Introduction

Overview

SSP is a tool-independent format for the description, packaging and exchange of system structures and their parameterization. The standard is comprised of a set of XML-based formats to describe a network of component models with their signal flow and parametrization, as well as a ZIP-based packaging format for efficient distribution of entire systems, including any referenced models and other resources. This description is tool neutral and is intended to be used primarily as an exchange format of simulation system descriptions between different tools.

SSP can be seen as an extension to the FMI (Functional Mockup Interface) standard [FMI 2.0]. FMI describes a tool independent standard to exchange single simulation models. Using SSP complete systems consisting of multiple interconnected simulation models can be defined with the desired signal flow and also with the wanted parameterization of each single model as well as the parameters for the complete system. This system topology can include hierarchies of sub-systems for proper structuring of the overall system. Along the signal flow, unit definitions can be used for automatic unit conversion or unit checking. Parameters can be derived from other parameters to be able to introduce dependencies of parameters between different components or to assure consistent use of same values for multiple components where needed. While SSP is closely aligned to FMI, it can also be used with components of other formats than FMI.

Properties and Guiding Ideas

This section introduces the basic properties and guiding principles that underlies the design of the SSP standard:

Tool independence

The core intent of SSP is to enable the exchange of partial and complete simulation systems between tools. In consequence, any tool-specific data is not stored in the standard schema and will be lost during transfer between implementations. However, specific data can be stored in customized annotations for reuse in the same tool. Additional common data for use across tools can also be standardized as layered standards.

Simplicity

SSP is focused on the possibility to exchange complete or partial topologies and parameters between different tools as simply as possible, while retaining essential information. This optionally also includes basic graphical information, to ensure basic recognizability, while eschewing the complexities of full and exact graphical model exchange. This approach also differentiates SSP from systems engineering standards like SysML: By focusing on simplicity and simulation-related features, SSP avoids the complexities that have plagued the tool-independent exchange of SysML models, enabling a wider adoption across the tool landscape.

Maximum Reusability

SSP contributes to maximizing reusability of models and parameters across tools and use cases. SSP can be used to specify required interfaces of a single component without the need to build a running model. Such an interface specification can be used as a template for the implementation of a model. In consequence, compatibility between different implementations can be guaranteed, ensuring that these implementations will fit into the overall system structure. The feature of automated unit conversion further reduces the need for adaptations between uses. Mechanisms like signal dictionaries support simplified signal matching even across hierarchies, without the need to adjust all intervening interfaces. Hierarchical structures enable encapsulation and combination of small components into subsystems that can be reused in larger subsystems.

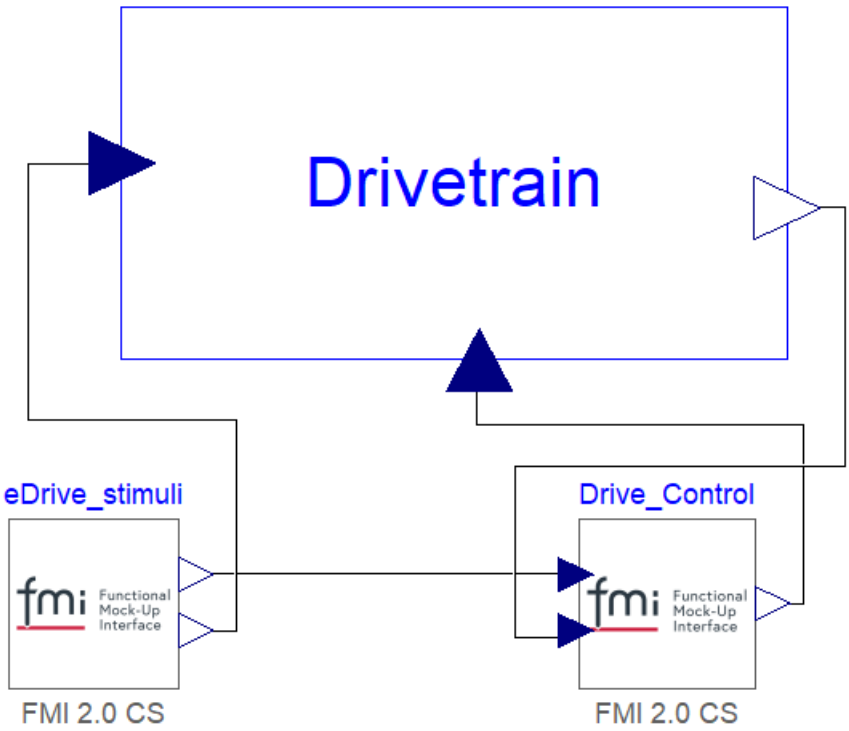
10.3.2 SSP Import

Importing an SSP package by a command

The command **File > Open > Import SSP…** provides initial support for importing basic SSP files comprising:

- Functional Mockup Units.
- Hierarchically nested SSPs.
- The connection structure between FMUs and nested SSPs.
- Parameter bindings as defined in SSP.

Dymola creates a Modelica package with models for the imported System Structure Description and its FMUs, using graphical information if available in the SSP. For each SSP parameter set, an additional model is created from the system model that sets parameters through modifiers in the extends statement.



Note that if you import an SSP package by scripting instead of by a command, you also have the option of selecting which FMU variables in the imported package that are to be exposed in the imported SSP package. See below.

Importing an SSP package by scripting

You can import an SSP package by the built-in function `importSSP`. The input arguments are:

`fileName` is the name of the imported SSP package file.

`filterVariables` is a Boolean argument, by default `false`. If you set it to `true`, you can filter which FMU variables in the SSP package that are to be exposed in the imported package. The dialog box is very similar to the one you can use when importing an FMU. See section “Filtering signals when importing an FMU” on page 1277.

`packageName` is the name of the Modelica package where the imported SSP model is to be located. By default, this argument is an empty string "", meaning that the SSP package is imported to a new Modelica library, where it is the root package. If you specify a Modelica package name, the SSP package is imported into that Modelica package.

There is one output argument `result`. It is a text string that, in case of success, contains the full path of the created model. If the function call was not successful, the text string is empty.

10.3.3 SSV Import

The System Structure Values (SSV) file format is really a part of SSP, but it has gained interest as a freestanding entity. You can store model parameterizations in SSV files, and apply these parameterizations on Modelica models and FMUs.

The idea is:

Start out with a Modelica model as your base model.

Open an SSV file to read parameter data.

Create a new Modelica model from the base model, applying the parameters (with values) as modifiers in an `extends` clause.

Importing SSV files by a command

The command **File > Open > Import SSP...** can also be used to import SSV files. When applying this command to an `.ssv` file (instead of an `.ssp` file), the command will call the `importSSV` built-in function described below. The current root level will be used as the base model, and, by default, the newly created model is placed at the top level of the package hierarchy.

Importing SSV files by scripting

You can import an SSV file by the built-in function `importSSV`. The input arguments are:

- `filename` – a text string that is the name of the SSV file to import.
- `modelName` – a text string that is the name of the existing (base) model that parameters are applied to.
- `packageName` – a text string that is the name of the Modelica package where the new model is to be inserted. (The name of the model is taken from the parameter set in the SSV file.) By default, this is an empty string, meaning that the created model is placed at the top level of the package hierarchy.

There is one output argument `result`. It is a text string that, in case of success, contains the full path of the created model. If the function call was not successful, the text string is empty.

10.3.4 SSP Export

Dymola can export Modelica models according to the System Structure and Parameterization (SSP) file format. Dymola performs the following steps when exporting to SSP:

- The top-level model structure is stored as a System Structure Description (SSD). That structure includes top-level components and any connections between these.
- Components are stored in two formats, depending on implementation:
    - If the component is an imported FMU, the FMU itself is copied to the resources directory of the SSP. The SSD will contain a reference to that FMU copy.

    - Otherwise, it is assumed that the component is represented by a Modelica model. In that case, the full path of the component´s model is stored in the SSD. The Modelica model itself is not stored in the SSP file.

- Model documentation, user metadata annotation, and the start and stop times, if specified, are stored in the SSD.

SSP export is available by the command **File > Save > Export SSP…** or as the built-in function `exportSSP`.

11 SIMULATION ENVIRONMENTS

11 Simulation Environments

11.1 Introduction

This chapter describes how to interface models created in Dymola to other simulation environments, or applications. The following external interfaces are detailed:

- Dymola interface to Matlab and Simulink (page 1302).
- Real-time simulation on dSPACE and Matlab Simulink Real-Time hardware (page 1313).
- DDE communication (page 1328).
    - DDE interface for Dymola.
    - DDE server support in the Dymosim simulator.
- OPC server support in the Dymosim simulator (page 1334).
- Building of stand-alone executables from source code or binary export models exported from Dymola (page 1339).

Note that information about external functions in other languages (C, Java, C++, and FORTRAN 77) is available in the chapter "Advanced Modelica Support".

11.2 Dymola – Matlab interface

The connection between Dymola and Matlab consists of an interface to Simulink (*DymolaBlock*) and a set of Matlab m-file utilities. The paths below are for Windows, for Linux the path to the Mfiles will be different.

In order to use the Matlab interface and all available m-files, please include

- `Program Files\Dymola 2023\Mfiles`
- `Program Files\Dymola 2023\Mfiles\dymtools`
- `Program Files\Dymola 2023\Mfiles\traj`

in the Matlab path. The first path *must* be present every time you want to use blocks constructed with Dymola; the others are optional, if all m-files should be available. A good idea is to do it once and then store the paths in Matlab.

Make sure that you have a Visual Studio C++ compiler installed on your computer. Make sure that the Matlab mex utility has been configured to use that compiler (type `mex -setup` in Matlab to configure). Finally, test by trying to compile and link an example mex file, e.g., `matlab\extern\examples\mex\yprime.c`

The Dymola – Simulink interface of Dymola 2023 supports Matlab releases from R2017a (version 9.2) up to R2021b (version 9.11). Only Visual Studio C++ compilers are supported to generate the DymolaBlock S-function on Windows. For Linux the gcc compiler is supported. The LCC compiler is not supported, neither on Windows nor on Linux.

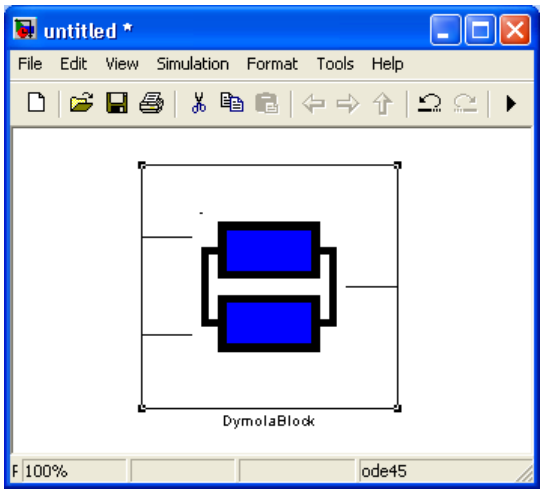
11.2.1 Using the Dymola-Simulink interface

Note that you need a Dymola-Simulink license to use the Dymola-Simulink interface.

The following describes the Simulink interface for Windows, for Linux please see section “Using the Simulink interface on Linux” on page 1309.

The Dymola interface to Simulink is located in the Simulink library browser as Dymola Block/DymolaBlock (if it does not appear you have probably not included the directory `Program Files\Dymola 2023\Mfiles` in your Matlab-path; please see above). You click once to open the library and you then drag the DymolaBlock to other models.

Dymola block before compiling.



The DymolaBlock block, with the Dymola logo, represents the Modelica model. It can be connected to other Simulink blocks, and also to other DymolaBlocks. Input and output ports are added after compiling the model. The DymolaBlock is a shield around an S-function MEX block, i.e., the interface to the C-code generated by Dymola for the Modelica model.

The DymolaBlock GUI

By double-clicking the DymolaBlock you open a GUI used to configure the block (see figure on the next page). Configuration involves specifying the Modelica model, selecting compiler options, compiling the Modelica model into a Simulink MEX file, setting and resetting parameters and start values of the block, etc.

In the DymolaBlock GUI, you can set the name of the model and optionally the file name. A simple way of setting the model name is to start Dymola, open the correct model, and then press **Select from Dymola>**. This automatically gives the model name. Note that the model name may contain a dotted name, e.g., `Modelica.Blocks.Sources.ExpSine`. The file name is optional, but is used both to find the model and to ensure that the Simulink model is up-to-date (for example, if the Modelica model is more recent than the MEX file you are prompted to re-compile the DymolaBlock).

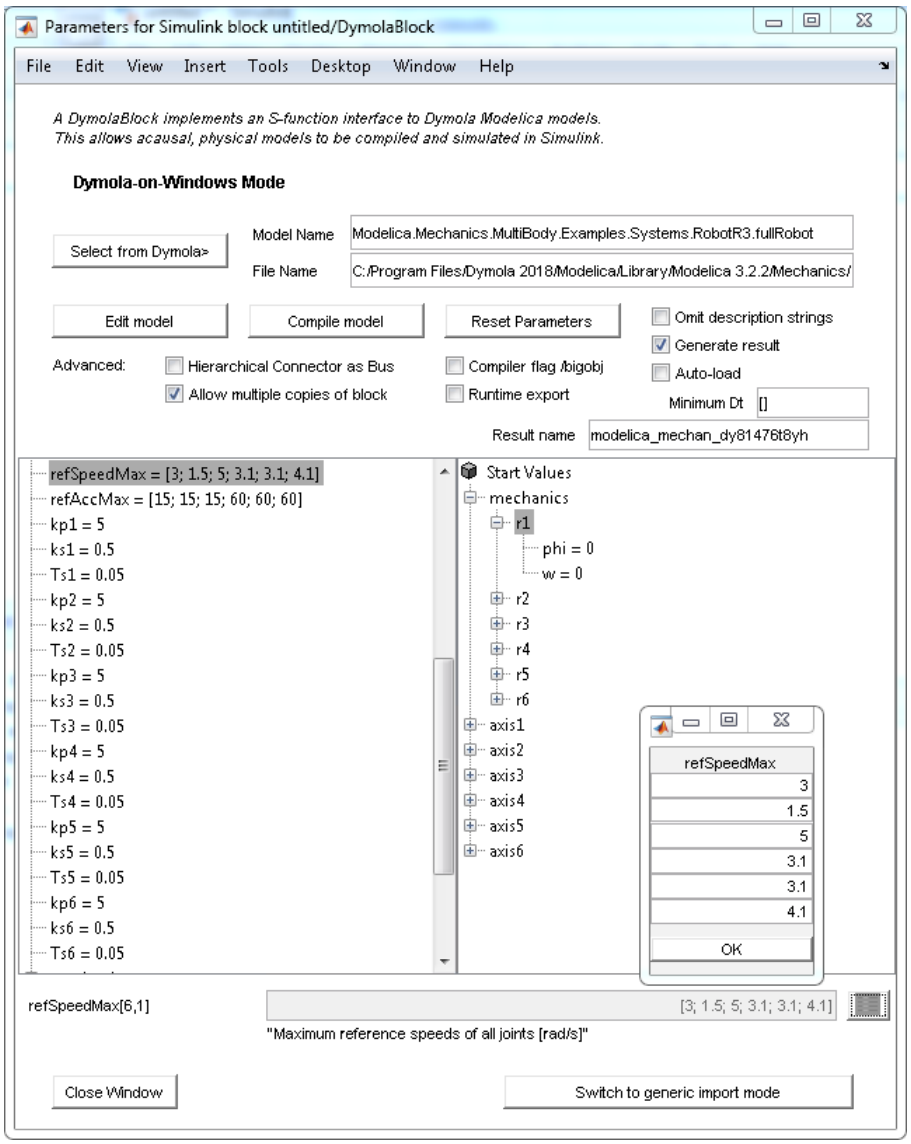
When associating a Modelica model with a DymolaBlock for the first time or when you need to edit the Modelica model, you may click **Edit model** (after selecting the correct current directory at the Matlab prompt). This launches Dymola, changes directory to the correct directory in Dymola and opens a window with the model. Edit the model in Dymola and verify it by making a test simulation in Dymola. After that, click on **Compile model**.

Note that Dymola is not started with the option /DSLS when launched automatically from the DymolaBlock. If DSLS licensing is used, you need to have Dymola started manually.

Two tree views are used to display parameters and start values of states (depending on the model). After compilation, the tree views are populated and the first level is expanded.

Selecting a variable will display its name, size (for matrix variables), and description string. The variable may be edited by entering a new value in the text field. Matrix variables may also be edited using a matrix editor, which is launched by the button to the right of the text field. While editing using the matrix editor, the rest of the DymolaBlock GUI is disabled.

Model dialog for a DymolaBlock in Simulink.



The parameter settings are kept even if you press **Compile model** (assuming they match). To reset parameters and start values to the default in Dymola press **Reset Parameters**.

Advanced configuration options

Hierarchical connectors as buses

Sometimes it is desirable to use buses for signals in Simulink and it is possible to map hierarchical connectors in Modelica to buses in Simulink. This is accomplished by checking **Hierarchical Connector as Bus** and re-compiling. In order to get an understanding of the structure, an easy way is to construct hierarchical output connectors in Modelica and examine the corresponding bus signals in Simulink. The rules for the buses are:

- All hierarchical input/outputs components are transformed into buses, except the signal-element of block-connectors.
- Connector members of simple types (Real/Integer/Boolean) are mapped to real signals in Simulink with the same name as in Dymola. Non-scalar elements are mapped into non-scalar signals in Simulink.
- Connector members that are scalar records/connectors are mapped to buses in Simulink with the same name as in Dymola.
- For connector members that are arrays of records/connectors, each element is mapped to buses in Simulink with the same name as in Dymola (except that Simulink requires that any "," for matrix elements is replaced by ";").

The mapping is fully hierarchical (i.e. connectors can contain sub-connectors that contain sub-connectors etc), and applied to all input/outputs.

Using multiple DymolaBlocks

A Simulink model may contain several DymolaBlocks as long as they all use different underlying MEX files. Using the same MEX file for several blocks in the same model will give incorrect results or runtime errors. For this reason, copying of the DymolaBlock takes special care to create a unique identification `<tag>` for the block such that all blocks in the model use different MEX files even if they are multiple copies of the same Dymola model.

Unchecking **Allow multiple copies of block** will disable the generation of unique tags when DymolaBlocks are copied. Unchecking it **should only** be done if you have dSPACE multi-processor hardware, since the build command of dSPACE internally makes copies of the model that should share the original MEX file name.

Compiling large models, /bigobj

When compiling large models, you may need to check the box **Compiler flag /bigobj**. This will supplement `COMPFLAGS` with the `/bigobj` option to increase the number of addressable sections of the produced object file. It should be noted that object files produced with `/bigobj` can only be consumed by linkers shipped with Visual C++ 2005 or later.

Export for use with Dymola runtime concept

Checking the box **Runtime export** enables the compiled model to be distributed and executed on a different computer by using a runtime concept. This box should never be checked in any other case.

Omitting description strings

For models with big input files, you can select to omit description strings for parameters and start values by selecting **Omit description strings**. This will avoid problems with Matlab memory limitations (especially for 32-bit installation).

Result files

Sometimes it is useful to generate result files in the Dymola standard format when running in Matlab/Simulink, e.g., to:

- Animate 3D-objects
- Investigate intermediate variables in the model
- Generate an input file for further simulation studies in Dymola (this requires an extra step: to export the generated result file (using the context menu in the variable browser in Dymola) as 'dsu.txt').

All of this is accomplished by selecting **Generate result**. If you also want the result file to be automatically loaded in Dymola select **Auto-load**. If the result files become very large it might be useful to reduce the size by introducing a **Minimum Dt**, which is a minimal output interval length, e.g., 0.1 (similar, but not identical to "Interval length" in Dymola). Any modifications of these choices require a new compilation of the DymolaBlock.

The generated result file will get the name specified in the **Result name** text field. By default, the text field is empty and if not specified the result name will be populated with the same name as the generated S-function MEX file after compilation. This is an automatically generated name (on the form `<modelname>_dy<tag>`) that ensures uniqueness between DymolaBlocks and result files also when generated from the same original Modelica model.

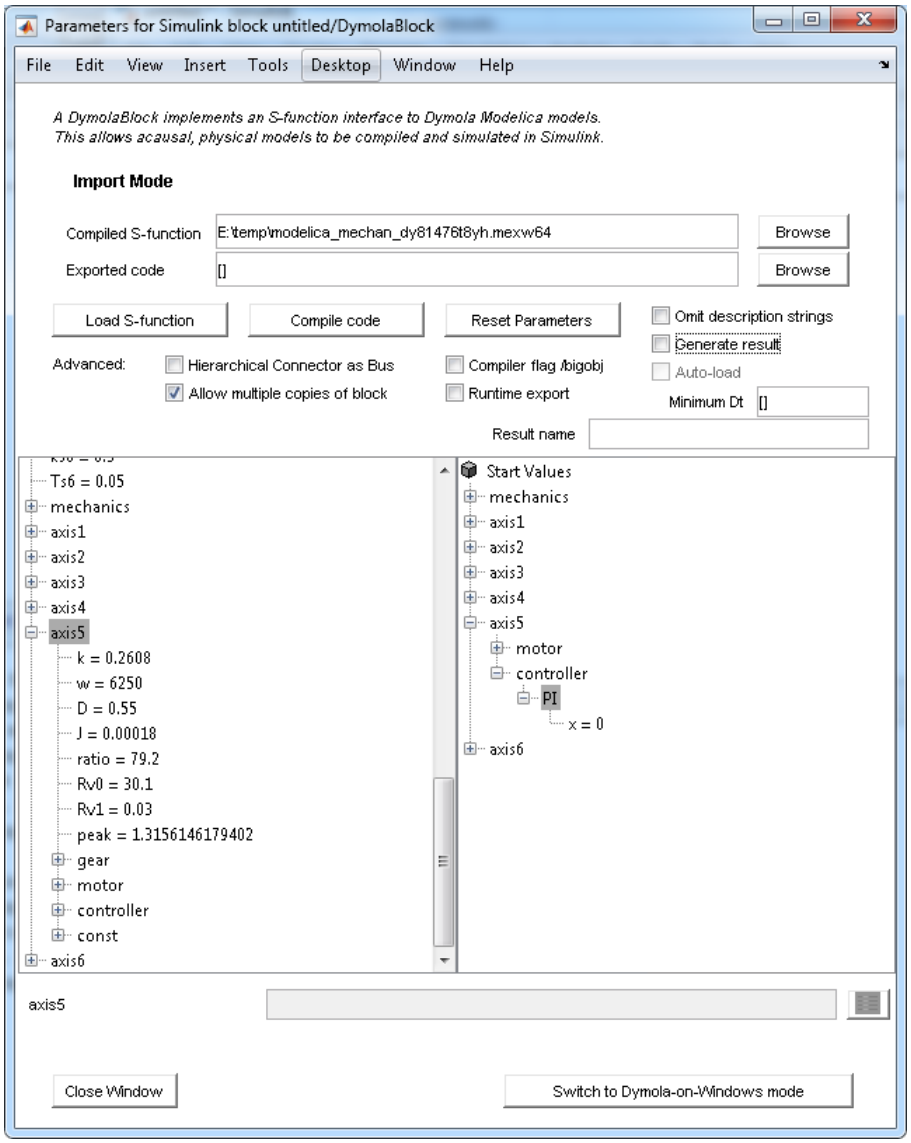
Import mode

The functionality of the **Select from Dymola>**, **Edit model**, and **Compile model** as described above relies on communication between Matlab and Dymola using a DDE connection. This mode is only supported on Windows and is therefore called Dymola-on-Windows mode.

The DymolaBlock also supports a generic import mode, to be used on non-Windows computers or on computers without a full Dymola installation. This mode does not require a DDE connection to Dymola and is intended for import of compiled DymolaBlock S-functions or code that have been exported from Dymola for use in Simulink. One dedicated use of this new functionality is to import S-functions or code that has been generated on a different computer using the binary model export feature.

The button at the bottom right corner of the DymolaBlock GUI is used to toggle between the Dymola-on-Windows (using DDE) mode and the Import mode.

GUI of DymolaBlock in import mode.



In import mode, it is possible to either import pre-compiled DymolaBlock S-functions (.mexw32, .mexw64, .mexglx or .mexa64 files) or C code that has been exported to Simulink from Dymola.

For an example on how to export code to Simulink, see section “Using the Simulink interface on Linux” on page 1309; the export is applicable to Windows as well.

Compiled S-function import

To import a pre-compiled DymolaBlock S-function, press the **Browse** to the right of the **Compiled S-function** input field. This opens a file selector that allows you to browse for the desired S-function to load. The selected S-function is then loaded to the DymolaBlock by pressing the **Load S-function** button. This will also create the tree views of parameters and start values and create the input and output ports of the block.

This import functionality allows a DymolaBlock to be compiled at one computer with a full Dymola installation and then sent in binary form to another computer where it can be loaded and simulated.

Importing compiled S-functions requires having a copy of the `Program Files\Dymola 2023\Mfiles` folder available. The location of this folder should be added to the Matlab-path as described above.

The following files created during compilation of a DymolaBlock need to be distributed to allow this type of import

1. Compiled S-function
2. dsin.txt or model-specific input file (`<modelname>_dy<tag>.txt`)

Note that the names of S-functions generated by the DymolaBlock are machine-generated, on the form `<modelname>_dy<tag>.<mexext>`, where `<tag>` is a unique tag for the DymolaBlock.

C code import

To import code that has been exported for Simulink, press the **Browse** to the right of the **Exported code** input field. This opens a directory selector that allows you to browse for the folder containing the exported code. The code is compiled into a DymolaBlock S-function by pressing the **Compile code** button. This will also create the tree views of parameters and start values and create the input and output ports of the block.

This import functionality allows a Modelica model to be exported at one computer and then the code generated during compilation can be distributed to another computer where it can be compiled, simulated, and also downloaded to real-time platforms.

Use of the C code import requires a full Dymola installation. The location of the Mfiles folder should be added to the Matlab-path as described above.

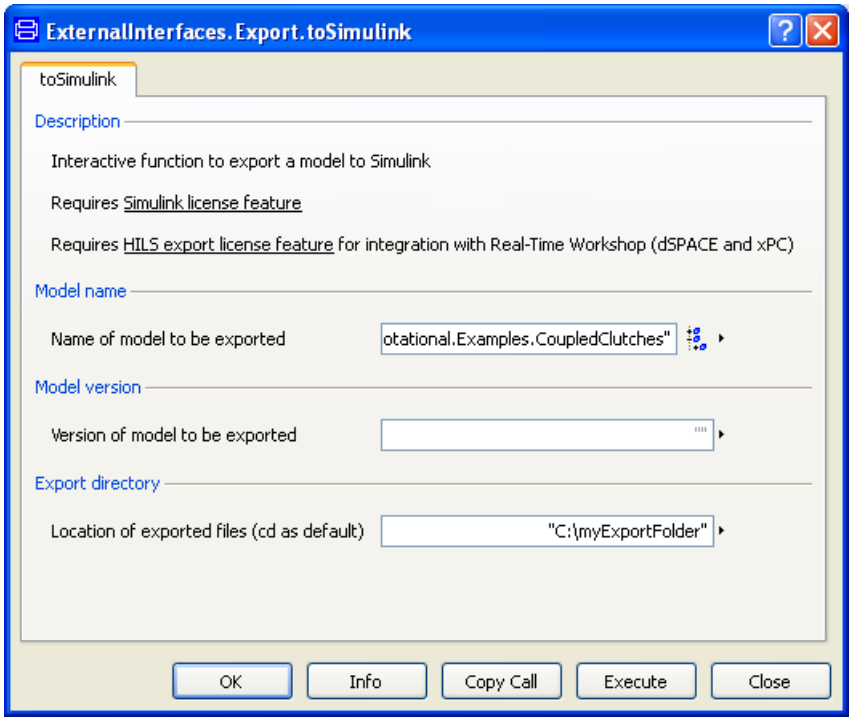
The following files created during compilation of a DymolaBlock need to be distributed to allow this type of import

1. dsmodel.c
2. dsin.txt
3. dymModelInfo.m

Using the Simulink interface on Linux

The Simulink interface on Linux uses a two-step export/import procedure. The import step relies on the C code import of the DymolaBlock GUI as described in the previous section. To export the needed files from Dymola, open the package `ExternalInterfaces` (located in the Modelica/Library subdirectory of the Dymola installation) and execute the function `ExternalInterfaces.Export.toSimulink`:

Function dialog for export of model code to Simulink.

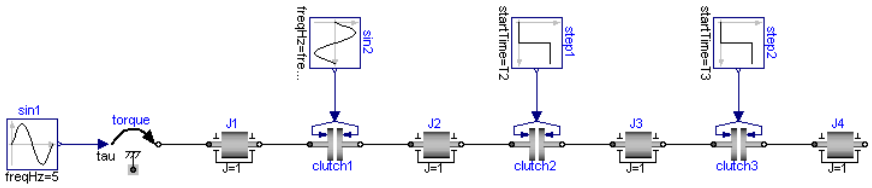


Select the name of the model to be exported and the export directory, the model version field can be left unchanged. The C code and .m file required by the import will then be generated in the selected export directory. Then browse to this directory in the **Exported code** input field of the DymolaBlock GUI and press the **Compile code** button.

Example model from Dymola – Coupled clutches

As an example of using the Dymola-Simulink interface, we will use the Dymola demo model *Coupled Clutches* shown in the figure below. The coupled clutches demo is opened in Dymola using the command **File > Demos**.

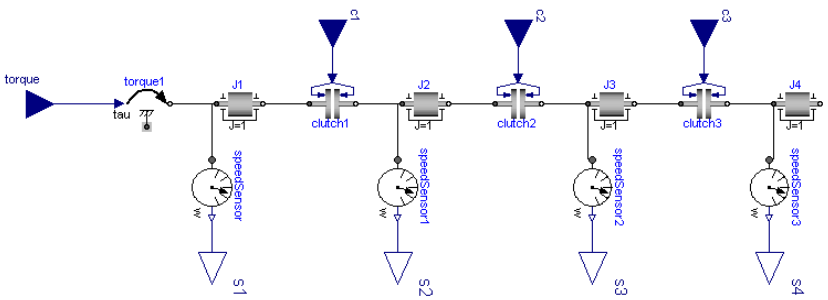
Coupled clutches example in Dymola.



In order to make the Dymola model useful as a block in Simulink, you first need to define external inputs and outputs to and from the Dymola block. You change this in your model in Dymola, and it may either be accomplished by declaring variables as input or output in the top-level model or graphically by adding input (filled) and output (non-filled) connectors, e.g., from `Modelica.Blocks.Interfaces`.

The latter option is used for the coupled clutches example and shown in the figure below. The sine wave and step inputs are replaced by input connectors, and speed sensors are connected to the inertias and used as outputs.

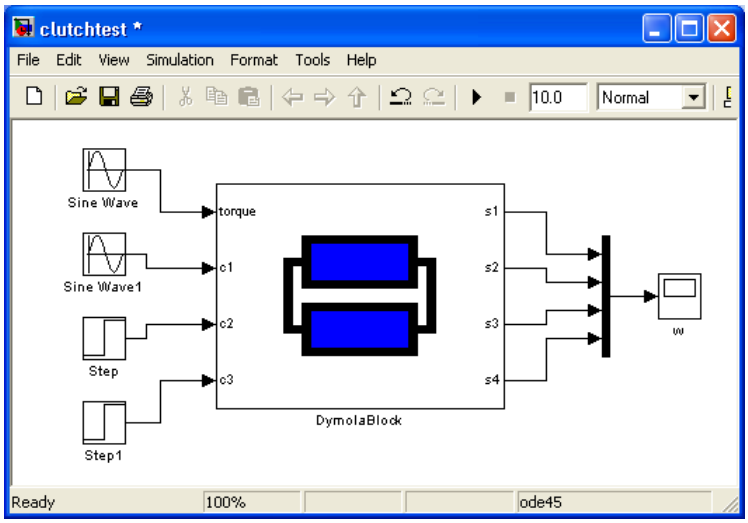
Model with external inputs and outputs for use in Simulink.



The next step is to create a new Simulink model and drag a DymolaBlock from the Simulink library browser to the model. To compile the model, double-click the DymolaBlock and press the **Select from Dymola>** button followed by **Compile model**. The compilation process creates a C-style S-function with a machine-generated name, in this case `coupledclutches_dy<tag>.c`, where `<tag>` is unique for the DymolaBlock.

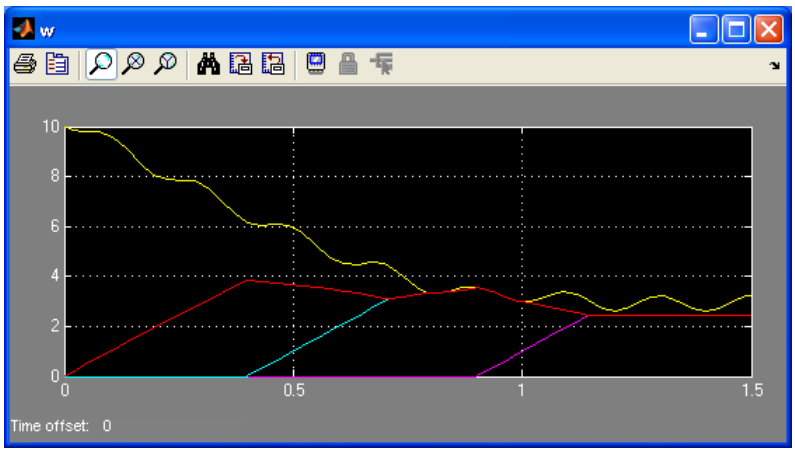
After compiling the model, input and output ports are added to the DymolaBlock, corresponding to the input and output connectors in Dymola. The names of the ports are the names of the Modelica input/output variables. The order of the graphical inputs and outputs correspond to the order in which they were added to the model. **Note:** Dymola may convert output variables to states, in which case the outputs will disappear from the DymolaBlock. Apply the `der()`-operator to an intermediate variable to avoid this problem.

Simulink model of the coupled clutches example.



The input ports are connected to external signal sources and a scope is connected to the output ports for plotting the rotational speed of the inertias. After configuring the signal sources according to the Dymola model (amplitude, frequency, phase, and step times), a 1.5-second simulation in Simulink produces the following output.

Simulink simulation output.



To recreate the corresponding Dymola simulation you should be aware that the parameterizations differ between the Modelica Standard Library models and Simulink blocks, e.g., the frequency of the Simulink Sine-block is measured in rad/s, which is commonly known as angular frequency and should thus be 2*pi times the frequency in the corresponding source in Modelica.

Implementation notes

The DymolaBlock relies on a number of callbacks and tags in order to allow copying, renaming, etc. This allows storing of settings for model name, parameters, and initial values in the Simulink format. However, manually editing mdl-files containing DymolaBlocks is not supported neither is changing the tag of the DymolaBlock or any of the callbacks associated with the DymolaBlock.

DymolaBlock is found in DymolaBlockMaster.mdl, and you can also get DymolaBlock by running DymolaBlockMaster. In this case, you must copy DymolaBlock to another model.

The library annotation for external functions is automatically used also in Simulink.

Using DymolaBlock GUI on a computer without Dymola installation

It is possible to use the DymolaBlock GUI on a computer without a Dymola installation by copying some files to that computer. This in needed when distributing DymolaBlock S-functions using the binary export option, or for use with a runtime concept. Files to copy are:

- newDymolaGui.p
- newDymolaGuiJava.p

These files should be copied from a computer having the same Dymola version as was used for creating the model. The files are located in the Mfiles directory (e.g. `Program Files\Dymola 2023\Mfiles`). They could be copied to any directory on the target computer. In order to be found by Matlab the path to that directory should be added to the Matlab paths.

11.2.2 Other Matlab utilities

The directory `Program Files\Dymola 2023\Mfiles` and the sub-directories `dymtools` and `traj` contain a number of useful Matlab m-files that can be used to, e.g., run dymosim, load and save tables, and manipulate parameters and start values interactively from the command prompt or from Matlab scripts. The m-files concerning the use of dymosim are described in the chapter “Simulating a model”, section “Dynamic Model Simulator”, sub-section “Dymosim reference”. For additional descriptions use the Matlab *help* command on the functions below.

The handling of tables is facilitated by Matlab routines for easier construction of 2- and n-dimensional tables. The n-dimensional routines below work in combination with an n-dimensional table lookup model, TableND. Please see the chapter “Developing a model”, section “Advanced model editing”, sub-section “Using data from files” for more information about TableND.

Tables:	
load2DTable	Load a 2-dimensional table in the format suitable for 2D-table interpolation in Dymola using the model Modelica.Blocks.Tables.CombiTable2D
loadNDTable	Load the N-dimensional table in the format suitable for ND-table interpolation in Dymola
save2DTable	Saves the 2-dimensional table in a format suitable for 2D-table interpolation in Dymola using the model Modelica.Blocks.Tables.CombiTable2D
saveNDTable	Saves the N-dimensional table in a format suitable for ND-table interpolation in Dymola

The following commands are used to change parameters and start values of a simulation without using the DymolaBlock GUI. These commands are useful in constructing batch simulations, in which a series of simulations are run from a script (using the *sim* command) with parameter changes in between each simulation. You need to close and re-open the DymolaBlock GUI to display the new values set using *setParametersFDsin*.

Loading and changing parameters and start values:	
loaddsin	Loads values from dsin.txt (or <modelname> .txt file)
setParameterByName	Set parameters and start values using the name of the corresponding Modelica variable.
setParametersFDsin	Modifies the parameters and start values of a DymolaBlock.
setfromdsin	Sets parameters and initial conditions from values in dsin.txt (the same as Reset Parameters in the DymolaBlock GUI). Calls loaddsin followed by setParametersFDsin

The following is an example of using the functions for the robot demo model:

```
>> [p,x0,pnames,x0names]=loaddsin('fullRobot.txt');
>> p=setParameterByName(pnames,p,'mLoad',25);
>> p=setParameterByName(pnames,p,'mechanics.world.mue',4.5320e14);
>> x0=setParameterByName(x0names,x0,'axis1.gear.spring.phi_rel',10);
>> setParametersFDsin('untitled/DymolaBlock',pnames,p,x0names,x0);
```

11.3 Real-time Simulation

Dymola provides support for real-time simulation (e.g., Hardware-In-the-Loop) on dSPACE and Simulink Real-Time platforms. This is accomplished using the Dymola-Simulink interface in connection with Matlab Coder.

Real-time simulation also requires the Dymola *RealtimeSim* or *SourceCodeGeneration* license options. The *RealtimeSim* license option is included in the Dymola standard license

since Dymola 2017 FD01. The *SourceCodeGeneration* license option is only needed if the model is not configured to use inline integration (embedded fixed-step solvers).

Models used for real-time simulation can contain more than one DymolaBlock, enabling use of the Simulink multi-tasking feature for more efficient utilization of computational resources in real-time simulations.

In this section, we will describe the needed configurations for each of the supported platforms. The description will focus on the Dymola-specific setup and we refer to the documentation for each platform for more complete instructions. Setup and real-time simulation results for the coupled clutches Simulink model described in the previous section will be shown for the supported real-time platforms.

Restrictions

Real-time Simulation only allows export of models that use inline integration, i.e., that have embedded fixed-step integrators. Inline integration is configured in Dymola by choosing **Simulation > Setup** and selecting the **Realtime** tab. The model can still be used in normal Simulink without inline integration, and a warning is issued that the model cannot be used for real-time simulation. Advanced options for inline integration are documented at the end of this section. The inline integration restriction for real-time simulation is not imposed for users having the Source Code Generation export option.

The model code exported for real-time simulation does not include the most advanced runtime routines that are normally included from binary libraries when building simulation executables from Dymola. Most notably, models with dynamic state selection cannot be used in real-time simulations[20]. For this reason, the exported C-code is used only when compiling for real-time targets. For normal simulation in Simulink, the S-function is linked with the binary libraries, thus, allowing use of the advanced runtime routines.

To generate code for real-time simulation when compiling models for Simulink, the RealtimeSim or SourceCodeGeneration options must be available. The option RealtimeSim is always available from Dymola 2017 FD01, where it was integrated in the Dymola standard configuration.

Checking out the SorceCodeGeneration option is by default enabled. Concerning the handling of the export option SourceCodeGeneration, please see section “Enabling Export” on page 1335.

[20] For more information about state selection, please see the chapter “Advanced Modelica Support”, section “Means to control the selection of states”.

11.3.1 dSPACE systems

Compatibility

Dymola 2023 officially supports the DS1005, DS1006, MicroLabBox, and SCALEXIO systems for HIL applications. For these systems, Dymola 2023 generated code has been verified for compatibility with the following combinations of dSPACE and Matlab releases.

- dSPACE Release 2017-A with Matlab R2017a
- dSPACE Release 2017-B with Matlab R2017b
- dSPACE Release 2018-A with Matlab R2018a
- dSPACE Release 2018-B with Matlab R2018b
- dSPACE Release 2019-A with Matlab R2019a
- dSPACE Release 2019-B with Matlab R2019b
- dSPACE Release 2020-A with Matlab R2020a
- dSPACE Release 2020-B with Matlab R2020b
- dSPACE Release 2021-A with Matlab R2020b and R2021a
- dSPACE Release 2021-B with Matlab R2020b, R2021a, and R2021b

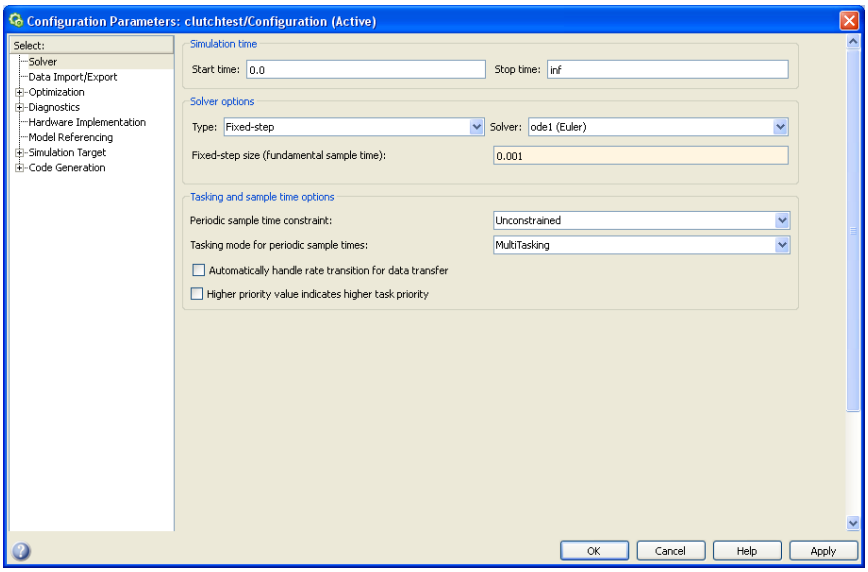
The selection of supported dSPACE releases focuses on releases that introduce support for a new Matlab release and releases that introduce a new version of a cross-compiler tool. In addition, Dymola always supports the three latest dSPACE releases with the three latest Matlab releases. Although not officially supported, it is likely that other combinations should work as well.

Configuration

The appearance of the dialogs presented in the following description may differ slightly depending on the configuration of Matlab and dSPACE releases. Also, note that for dSPACE SCALEXIO, the selection of system target file and the configuration of tasks is performed from dSPACE ConfigurationDesk (see the dSPACE documentation for details).

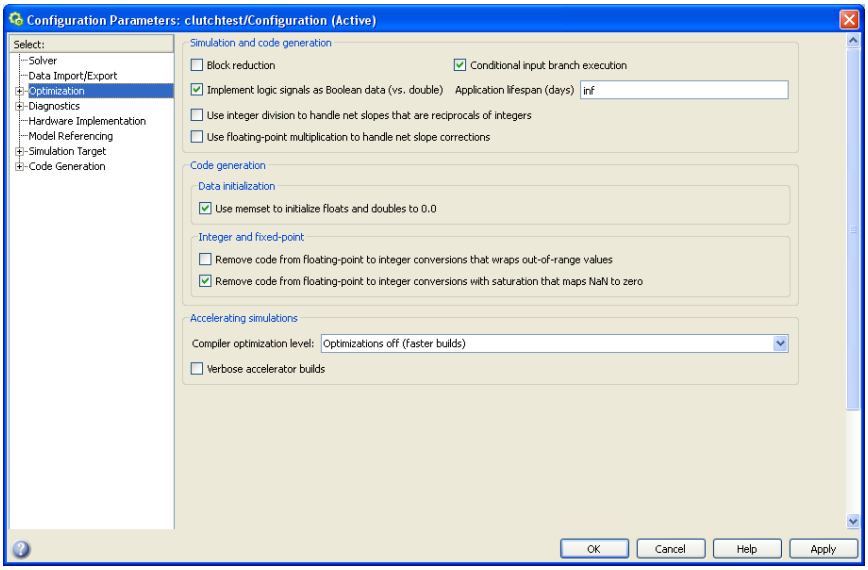
Open the Configurations Parameters dialog (in the Simulation menu of the Simulink model) and select Solver. Set the start time to 0 and the stop time to `inf`. The solver needs to be a fixed-step solver and the fixed-step size is configurable by the user. Make sure that the check box “Higher priority value indicates higher task priority” is unchecked.

Configuring solver settings.



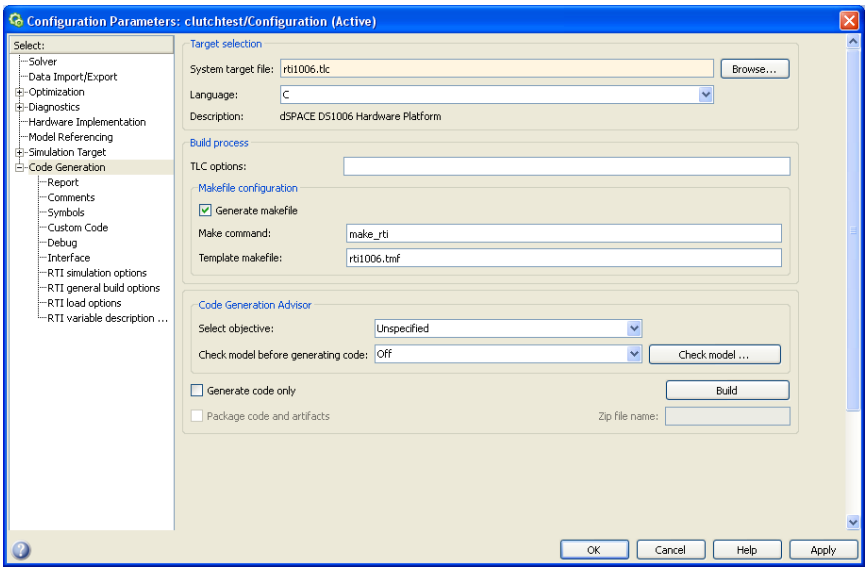
Next, select the Optimization tab and make sure that the check boxes “Block reduction” and “Signal storage reuse” (for older Simulink versions) are unchecked.

Configuring optimization settings.



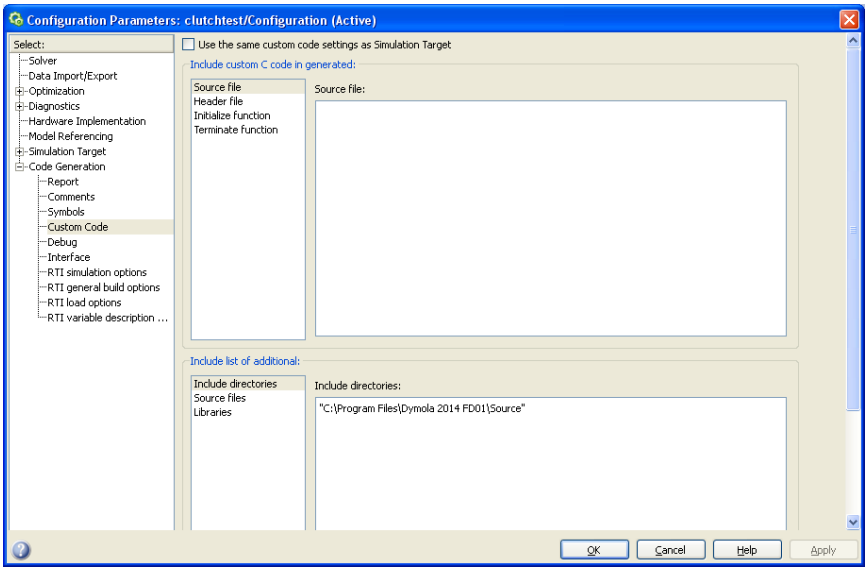
The remaining settings are made for the code generation. Select Code Generation and expand the tree. First, click Browse to select the appropriate RTW target. In our case, we have chosen the `rti1006.tlc` target file for the dSPACE DS1006 platform. For the DS1005 platform, choose `rti1005.tlc`. The default build process can be used without change.

Configuring Simulink Coder.



Second, the include directory for custom code should be configured. This is the directory where additional C source and header files needed to compile Dymola models are found. The default should be the `Source` subdirectory of the Dymola distribution. Note that the path must be surrounded by quotes if it contains spaces.

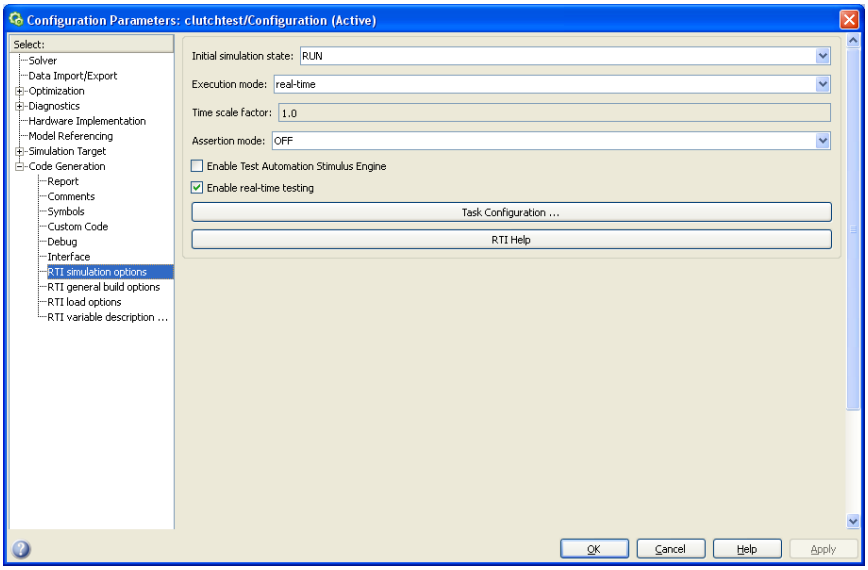
Configuring Custom Code.



The final configurations are related to the dSPACE real-time interface (RTI). Under RTI simulation options, it is, e.g., possible to set the initial simulation state and execution mode. The dSPACE system can run in different execution modes, for example in real-time or as fast

as possible. To configure RTI real-time tasks (including the overrun strategy, which will be described below), press the “Task Configuration” button.

Configuring the dSPACE RTI.



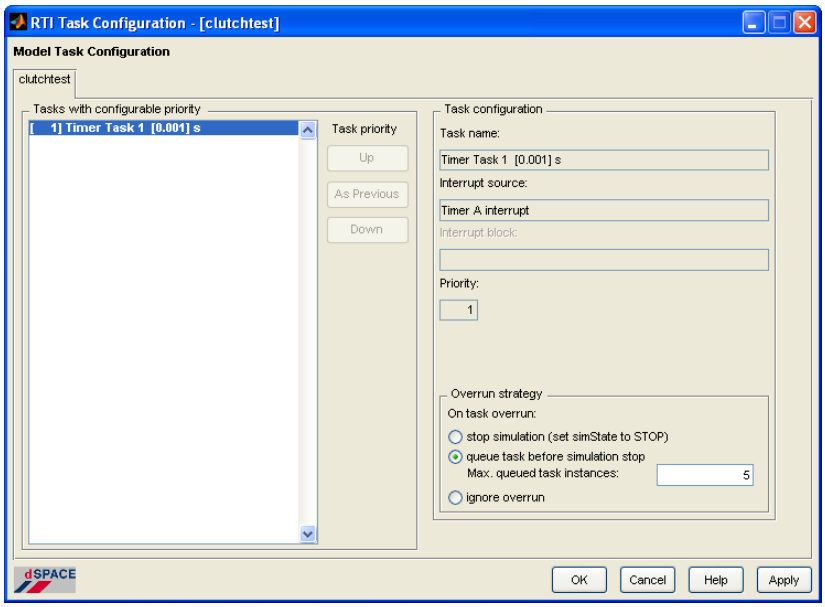
Overruns

An overrun situation may arise for several reasons, as documented in the dSPACE manual. For Dymola models, event iteration in the model may also require additional CPU resources.

In these cases it is usually helpful to plot the turnaround time (model calculation time plus overhead in every sample), a signal provided by the dSPACE system. The turnaround time will show when additional resources are needed. In many cases occasional overrun situations are harmless.

The RTI task configuration enables you to specify the number of instances of a task that may be queued before the real-time kernel issues an overrun error. The appropriate setting can be determined by trial-and-error. The dSPACE ControlDesk also allows plotting of important real-time system variables, such as, the model turnaround time and the number of overruns.

RTI task configuration (overrun strategy).



Building and loading the model, *dym_rti_build* and *dym_rtimp_build*

Dymola provides a Matlab function `dym_rti_build` (located in `Program Files\Dymola 2023\Mfiles`) to build real-time models containing DymolaBlocks for dSPACE. This command will invoke Simulink Coder to generate code, compile, link, and download the application. The function will also generate a variable description file for improved presentation of Dymola variables (outputs, parameters, and start values) in dSPACE ControlDesk. It is also possible to build and download the model without generating the Dymola-specific variable description file by choosing *Tools -> Simulink Coder -> Build Model* (or by pressing **Ctrl+B**).

The inputs to the `dym_rti_build` function are the model name and a build command (*'C'* to generate code only, *'M'* to compile and link, *'L'* to load application, '*CM*' to generate code, compile and link, etc). Type `help dym_rti_build` in Matlab for complete instructions. Note that the name of your DymolaBlock must start with the string `DymolaBlock` in order to generate a correct variable description file.

Dymola also provides the function `dym_rtimp_build`, which combines the dSPACE `rtimp_build` command and `dym_rti_build` to build dSPACE multi-processor systems for models containing DymolaBlocks. Again, use the Matlab `help` command (`help dym_rtimp_build`) for complete instructions.

The dSPACE ControlDesk is used to start and stop the simulation, changing parameters, and plotting signals from the simulation. If the model has been built using `dym_rti_build`, the outputs, parameters, and start values of the DymolaBlock are found under *User Variables* in the ControlDesk variable browser.

Notes:

- From Dymola 2021 a new function `dym_rti_build2` is available. This function replaces `dym_rti_build` for building dSPACE applications from models containing DymolaBlocks. The new function uses the new dSPACE function `rti_build2` instead of the old function `rti_build`.

  A corresponding new multi-processor function `dym_rtimp_build2` is also available.

  These functions are supported with dSPACE Release 2019-B and later.

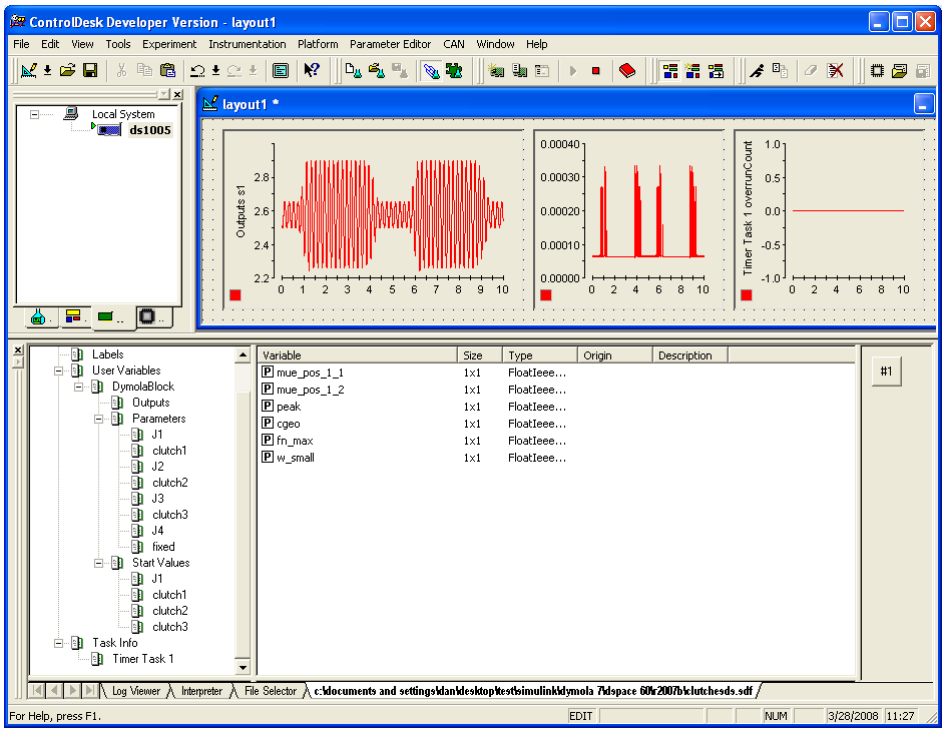
- For dSPACE Release 2017-A and later, the function `rti_usrtrcmerge` is no longer available. As a consequence, it is required to run the standard `rti_build` function (with the 'CM' command) after `dym_rti_build` to get your _usr.trc content added to the main .trc file. For example:

  ```
  >> dym_rti_build('myModel', 'CM')
  >> rti_build('myModel', 'Command', 'CM')
  ```

  (This note applies to the new functions `dym_rti_build2` and `rti_build2` as well.)

- For dSPACE SCALEXIO, the build process is configured and started from dSPACE ConfigurationDesk. However, it is still possible to use `dym_rti_build` to construct the variable description file by entering the string '*trc*' as build command. (This note applies to the new function `dym_rti_build2` as well.)

Simulation of the coupled clutches demo example using dSPACE ControlDesk.



11.3.2 Simulink Real-Time

The Simulink Real-Time (formerly xPC Target) environment allows Simulink models to be compiled at a host computer using Visual Studio or Watcom C compilers and then downloaded and executed in real-time on a target computer. The target computer can be any standard desktop PC booted with the Simulink real-time kernel. Communication between the host and target is performed either using TCP/IP or using a direct RS-232 serial connection. The Simulink Real-Time environment is controlled from the xPC Target explorer, which is started by the command `xpcexplr`.

Compatibility

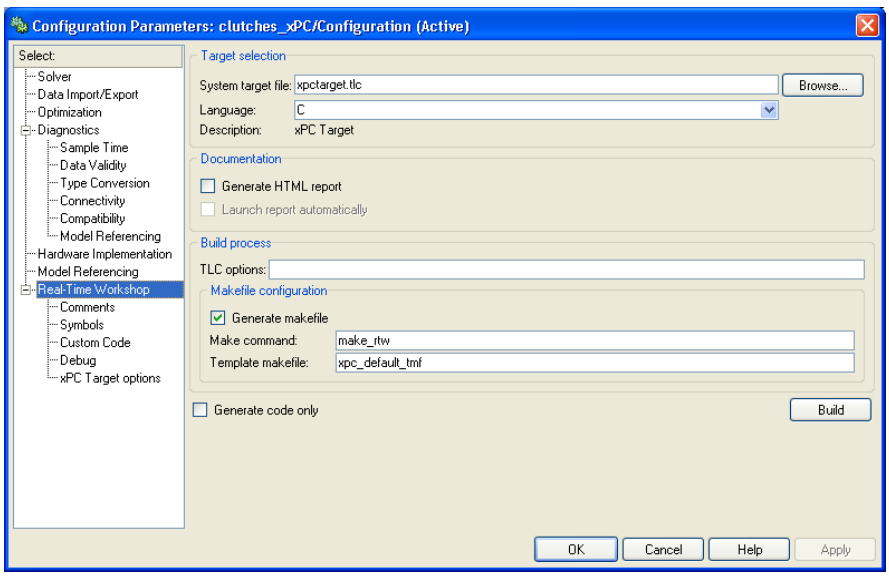
Compatibility of Dymola 2023 real-time simulation with Matlab Simulink Real-Time has been verified for all Matlab releases that are supported by the Dymola – Simulink interface, which means R2017a (Simulink Real-Time version 6.6) to R2021b (Simulink Real-Time version 7.2). Only Microsoft Visual C compilers have been tested.

Configuration

A fixed-step solver is required, which is configured by opening the Configuration Parameters dialog in Simulink and selecting the Solver tab. Also, set the start time to 0 and the stop time to the desired value (usually `inf` for real-time simulation).

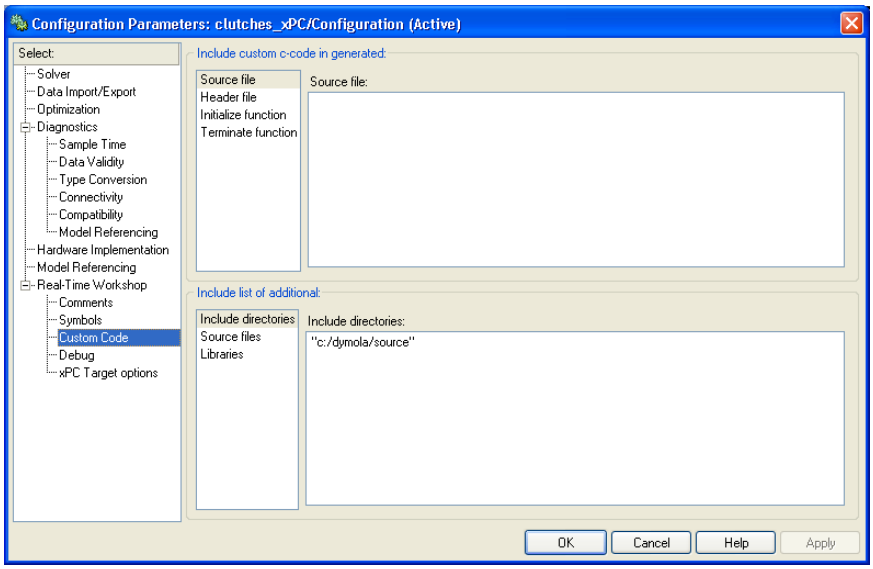
Next, select Simulink Coder to configure the target language compiler for Simulink Real-Time. Select `xpctarget.tlc` (`slrt.tlc` in Matlab R2014a and later) as 'System target file'. Use the default settings without change.

Configuring the target language compiler for Simulink Real-Time.



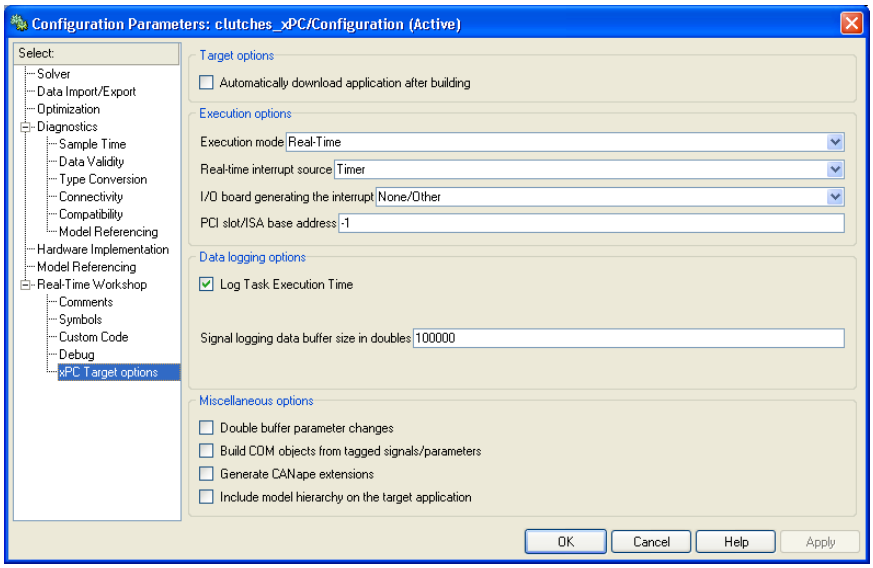
Next select the Custom Code tab to configure the additional include directory to point to the directory where additional C source and header files needed to compile Dymola models are found. The default should be the `source` subdirectory of the Dymola distribution (see below).

Configuring Custom Code.



Finally, select the xPC Target options tab to configure some options specific for xPC Target. Here it is, for example, possible to choose between real-time and free-running (as fast as possible) execution modes and to select data logging parameters.

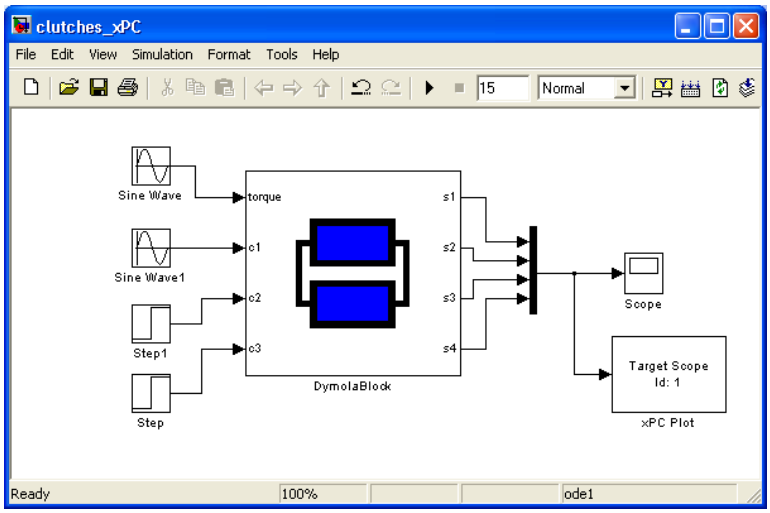
Configuring xPC Target options.



Building and loading a model

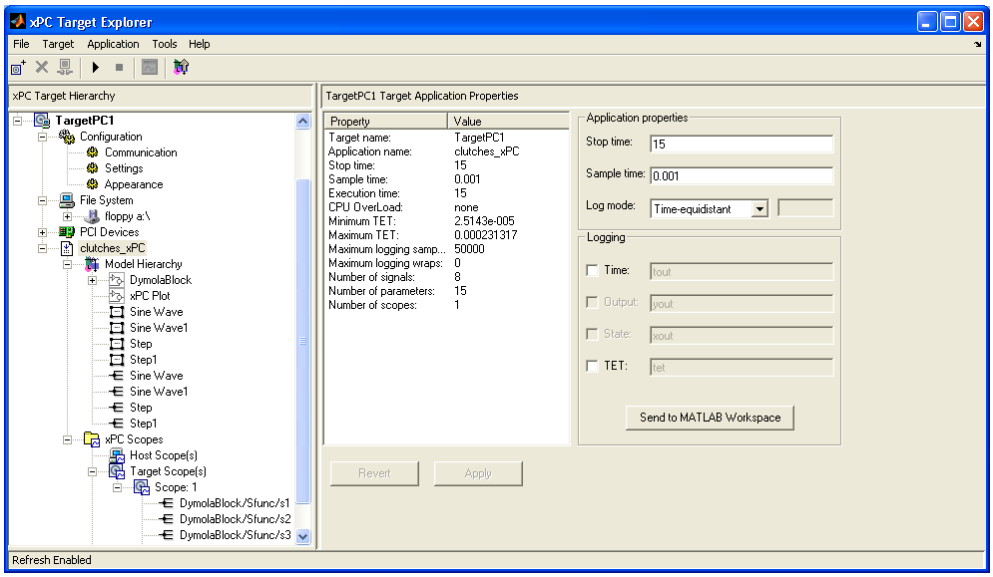
To be able to monitor signals from the simulation on the target node, an xPC Target Scope is added to the coupled clutches Simulink model before code generation and compilation.

CoupledClutches model with an xPC Target Scope.

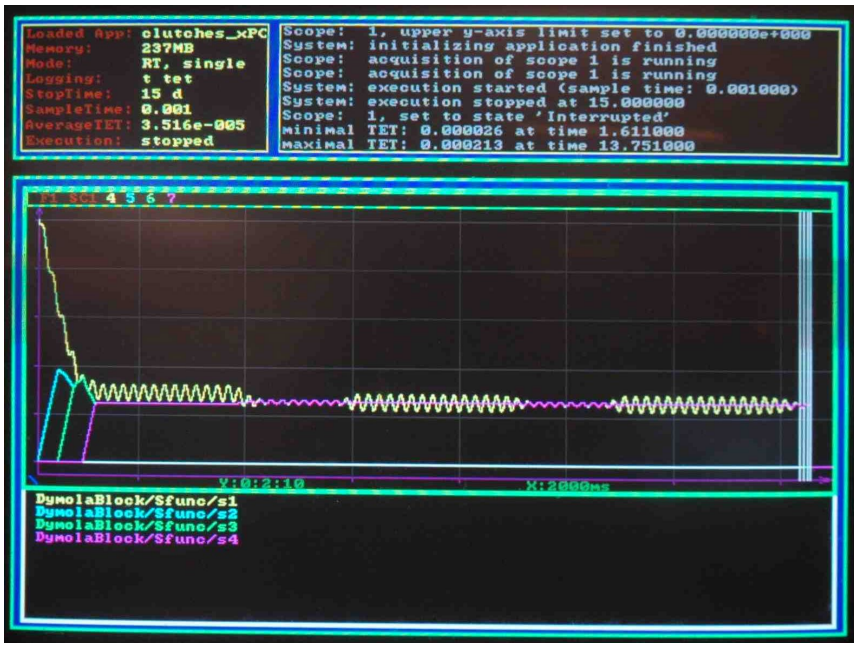


After this modification of the model, the real-time application is built by choosing *Tools -> Simulink Coder -> Build Model* (or by pressing **Ctrl+B**). This will create an executable (`.dlm` file) that then is downloaded to the target node (either automatically during the build process or manually from the xPC target explorer). The real-time simulation is then conducted either from the xPC Target explorer (see below) or directly from the Simulink model by running in External mode.

Monitoring the real-time simulation using the xPC Target explorer.



Results from a real-time simulation on xPC Target of the coupled clutches example are shown below. The range of the x-axis can be changed by modifying the value of 'Number of samples' in the xPC Target Scope.



Real-time simulation of coupled clutches on xPC Target.

11.3.3 Advanced Options for Real-Time Simulation

The section describes a number of options specifically designed for inline integration and to improve performance for real-time simulation.

Inline integration in Dymola in general is also documented in the chapter "Simulating a model", section "Improving simulation efficiency", sub-section "Inline integration".

To get a small improvement of the simulation performance, you can activate the global optimization in the compiler, by setting the flag

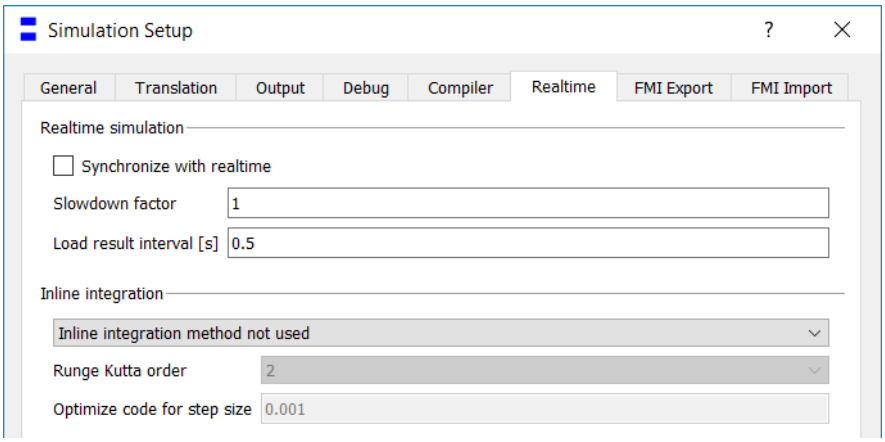
```
Advanced.Define.GlobalOptimizations = 2;
```

before generating code. (The default value of the flag is 0.)

This flag works the same for all Visual Studio compilers. Note that although the simulation performance is somewhat improved by setting the flag, but the compilation of the code might take substantially longer for large models. The setting corresponds to the compiler command `/Og`.

General options

The following options are available from the **Realtime** tab in **Simulation > Setup** menu of Dymola:



The first three selections (Realtime simulation group) are described in the index entry "*real-time simulation – options*" in this manual.

The next three selections that can be made for Inline integration in the menu above corresponds to three flags that can be set from the command input line in Dymola or from scripts.

The inline integration method corresponds to the flag

```
Advanced.InlineMethod = 1;    // 0-7
```

The order for the higher-order Runge-Kutta methods (including Rosenbrock) corresponds to the flag

```
Advanced.InlineOrder = 2;     // 2-4
```

The step size that the code is optimized for corresponds to the flag

```
Advanced.InlineFixedStep = 0.001;
```

Two other general flags that can be set from the command input line are

```
Advanced.Define.AutoRemoveAuxiliaries = true;
```

Removes code for auxiliary variables that neither influences the simulation state nor the outputs. This improves performance a bit. If the auxiliary code is used to assert correct behavior or to generate external outputs that code will not be run.

```
Evaluate = true;
```

Evaluates the code based on the given parameter values (excluding top-level parameters) preventing further modifications. This used to be very important for multi-body models, since axis of rotations are often along a co-ordinate axis. That is, however, already taken care of with specific Evaluate annotations for those variables, and thus `Evaluate=true` should be less important now.

Options for nonlinear solver

```
Advanced.Define.NewJacobian = false;
```

This avoids recomputing the Jacobian from scratch every step. The problem with keeping the Jacobian from the previous step is that at certain points major updates of the Jacobian are needed. Thus even if keeping the Jacobian is on average better it is not clear that it will improve real-time performance.

```
Advanced.Define.AimForHighAccuracy = false;
```

Normally the non-linear systems of equations are solved with additional accuracy. That additional accuracy not only improves the accuracy of the solution in itself, but also avoids spurious events. Setting the flag to false can thus cause loss of accuracy and spurious events.

```
Advanced.Define.NonLinearIterations = 2; // or 3, 4 or 5
```

Limit the number of non-linear iterations of the non-linear systems of equations. In combination with `Advanced.Define.NewJacobian = false` this improves performance substantially, but manual verification of the result is needed. The default of zero ensures that iterations are until convergence. Values 0, 2-8 are supported, but values 6-8 are not recommended; the result will be better using value 0 (default) in such cases.

Options for implicit inline integration

```
Advanced.InterpolateStatesForInline = true;
```

In case of an event we will take a smaller step to step just past the event point. This shorter step would normally slow down the non-linear solver, but this flags changes it so that we first compute as for a normal step and then only scale the states. This avoids the loss of efficiency. This flag is important to set for implicit inline integration.

```
Advanced.InlineIntegrationJacobian = false;
```

This is only useful if `Advanced.Define.NewJacobian` is false, and basically avoids generating the large amount of code needed for a Jacobian that is only needed at the initial step.

Simulink option

```
Advanced.InlineAsDiscrete = true;
```

Normally the inlined block is run based on the sampling time. With this setting the sampling time for this block is given by the fixed step-size setting of the **Realtime** tab (`Advanced.InlineFixedStep` as described above). If `Advanced.InlineAsDiscrete` is set and the fixed step-size is set at the default zero the block will have inherited sampling time in Simulink.

Break delay loops

```
Advanced.BreakDelayLoops = true;
```

Use this flag to break loops involving the `delay` operator. The delay operator is normally treated as if there were a direct coupling between the input and the output. Be aware that setting of this flag may lead to wrong simulation results if the delay times are too short.

Profiling of execution time

```
Advanced.DymosimRealTimePriority = true;
```

Use this flag to enable profiling of execution time for a simulation. This will set the priority class of the dymosim process to REALTIME_ PRIORITY_CLASS. Be aware that processes of this class may interrupt system threads managing mouse or keyboard and that they may also be difficult to terminate.

```
Advanced.Define.PrecisionTiming = true;
```

Set this flag to enable precision timing. This functionality uses the frequency counter of the processor to provide high-resolution time measurements.

```
Advanced.Define.UseRDTSCFrequency = 0.0;
```

This variable is used to specify the frequency (in Hz) of the processor. It is recommended to keep it at the default value of 0.0 (auto detection), which means that the frequency is automatically read from the text description of the processor.

Special mode handling for systems with events

For realtime simulation it is possible to activate a special mode handling for systems with events in order to efficiently handle diodes, clutches, and similar models. Code is generated statically for each combination of them being active, instead of generating code that dynamically computes and factors a Jacobian that depends on these combinations.

This generates more code – and does not significantly alter the total simulation time in most cases. It is thus not useful in offline simulations.

However, the maximum time per step can be significantly reduced – which is important for realtime simulations.

It is most easily activated by setting:

```
Advanced.Translation.Modes.Activate = true;
```

(Note that it is usually necessary to evaluate parameters as well.)

As default code mode variables are automatically detected and code is generated for all non-singular combinations of mode variables.

It is possible to disable the second part by setting:

```
Advanced.Translation.Modes.AllModes = false;
```

In that case, active modes are accumulated in a special mo-file. The file is called *ModelName*`ModeValues.mo` and it contains *ModelName*`ModeValues`.

This can be turned off by setting:

```
Advanced.Translation.Modes.Collect = false;
```

An alternative way of activating mode handling is to use such files directly.

Reference: Mattson, S. E., M. Otter, and H. Elmqvist (1999): "*Modelica Hybrid Modeling and Efficient Simulation.*" in Proceedings of the 38th IEEE Conference on Decision and

Control, CDC'99. Phoenix, Arizona, USA, Dec 7-10, 1999, under section "Real-Time Simulation".

11.4 DDE Communication

Two types of DDE communication are provided:

- An interface for Dymola for commands etc., (An additional feature is to associate new commands with the .mo file type in Windows Explorer.)
- DDE server support in Dymosim simulator.

11.4.1 DDE interface for Dymola

DDE commands can be sent to Dymola just to be executed, or with special DDE-request to allow the caller to collect the result from Dymola (since a normal execute does not allow more advanced return codes.)

Executing

Dymola accepts commands through a Windows DDE connection. The DDE connection should be opened with an application name of "dymola"; the topic name is not used. (You can use e.g. `'model'` as topic.)

The following commands are recognized:

- All Dymola commands as entered in the command input or read from a script file (including the command for running a script file).
- The command **open filename** that corresponds to the command **File > Open** in the graphical editor.

All Modelica script features are also supported.

Fetching results

The special DDE-requests are: "ModelicaString:expression" and "MatlabString:expression", in both cases the result of the expression (usually a function call) is returned as a string, containing a literal expression in one of the syntaxes (of Modelica or Matlab).

Example (the example is explained in more detail below):

```
>>  ch=ddeinit('dymola','model');
>>  res=eval(ddereq(ch,'Matlabstring:Modelica.Math.Matrices.
solve([1,2;3,4],{1,5})',[1,1]))
```

Note 1: Multiple outputs are supported.

Note 2: You should adjust the timeout in the calling program (e.g. Matlab) to allow for the command to complete.

Note 3: When using scripting, please note the usefulness of some examples in chapter "Simulating a model", section "Scripting". In particular note the use of the function `SimulateExtendedModel`.

Note 4: In Matlab, additional information about ddereq can be obtained by typing `help ddereq` in the Matlab command window.

Example: Solving a real system of linear equations (Matlab)

The function `Modelica.Math.Matrices.solve` is used for solving a real system of linear equations A*x=b where A is a matrix and x and b are vectors.

This function can be used from Matlab using DDE connection, executing the function in Dymola and fetching the resulting value to Matlab.

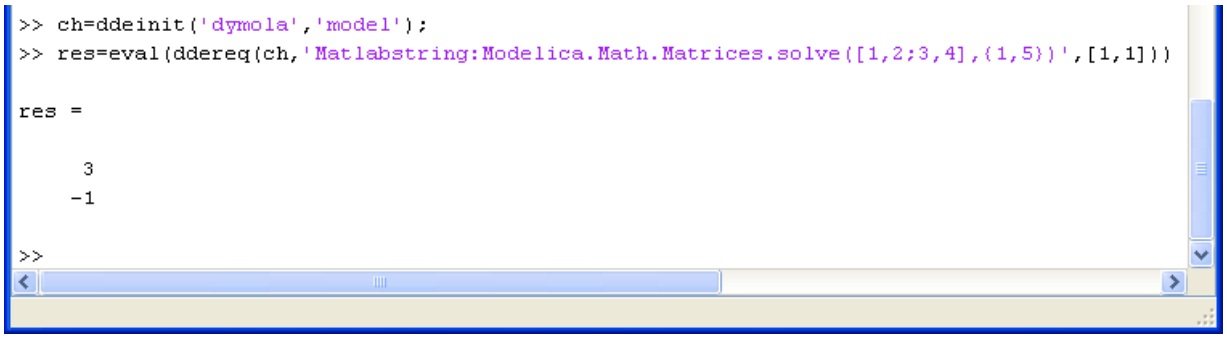
Dymola must be running.

In the Matlab command window the following command can be executed:

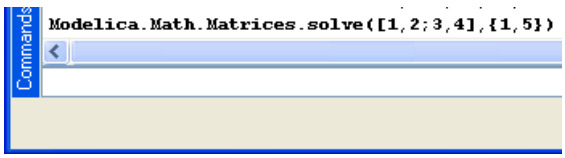
```
>>  ch=ddeinit('dymola','model');
>>  res=eval(ddereq(ch,'Matlabstring:Modelica.Math.Matrices.
solve([1,2;3,4],{1,5})',[1,1]))
```

The format [1,1] means that the clipboard format CF_TEXT is used for the request and that the result is returned as a string. `eval` is used to make Matlab to execute that resulting string as an expression or statement.

The result will be output in the Matlab command window:



Please note that the corresponding call (the resulting command in Dymola) will also be shown in Dymola command window (this makes it easier to e.g. trace errors):



The timeout of the ddereq command is by default 3000 ms. If some other value should be used it can be specified in the command. If e.g. the timeout should be 4000 ms in the example above, the last line would be:

```
>>  res=eval(ddereq(ch,'Matlabstring:Modelica.Math.Matrices.
solve([1,2;3,4],{1,5})',[1,1],4000))
```

11.4.2 Explorer file type associations

It is possible to associate new commands with the ".mo" file type. These commands are accessible through the right-mouse button menu in Explorer. We suggest this setup (the setup varies somewhat with version of MS Windows):

1. Start Windows Explorer and select View/Options. Select the File Type tab.
2. Associate new menu commands with the file type. Click on "New...", then enter:
   a. Action: Open
   b. Application used to perform action: browse for the Dymola program, typically C:\Program Files\Dymola 2023\bin64\dymola.exe.
   c. Use DDE: check
   d. DDE message: open %1
   e. Application: dymola
   f. Leave the remaining fields empty.

In a similar way, you can associate several commands with the extensions `.mo` and `.mos`:

Action	Application to perform action	DDE message	File extension
Edit	C:\windows\notepad.exe%1 or C:\winnt\notepad.exe%1		all
Open	C:\Program Files\Dymola 2023\bin64\dymola.exe	open%1	*.mo
Run	C:\Program Files\Dymola 2023\bin64\dymola.exe	@%1	*.mos

11.4.3 DDE Server support in Dymosim simulator

Dymosim can be compiled as a Windows application with built-in DDE server with realtime capability. To be able to perform such a compilation, DDE must be selected as embedded server in the **Compiler** tab in the simulation setup menu (reached by the command **Simulation > Setup**).

For more information about Dymosim in general, please see chapter "Simulating a model", section "Dynamic Model Simulator". Note in particular the flag `Advanced.CompileWith64` that can be used in Dymola to select if dymosim with DDE server should be compiled as 32-bit or 64-bit application.

Without any action from the user, Dymosim will be built and executed in the usual way, i.e., without real-time synchronization. The noticeable differences are that the current simulation

time is shown in the minimized Dymosim window, and that the dymosim application has a graphical user interface (see below).

Real-time simulation

If the environment variable DYMOSIMREALTIME is defined, Dymosim will start in real-time mode, where the simulation time is synchronized with real (or wall-clock) time. The time display will also include the time the simulation is delayed at each accepted output point, relative real-time. A negative value indicates that the simulation is faster than real-time, i.e., that there is spare time for additional computations. (In this case the simulation is actually delayed by the system in order not to accumulate the time difference.)

Dymosim DDE server

Dymosim compiled as a Windows application will act as a DDE server, allowing some other application to retrieve data values or set parameters. Dymosim must be started (e.g. by running the simulation in Dymola) before it can accept DDE operations; Matlab's ddeinit will not start Dymosim automatically, for example.

A DDE connection is established by sending a WM_DDE_INITIATE messages with the application name "dymosim" (any topic can be used).

Matlab example: `channel=ddeinit('dymosim','xxx')`

After a Stop command or at the end of simulation, Dymosim will send a WM_DDE_TERMINATE message.

Matlab example: `ddeterm(channel)`

Note that in addition to running Dymosim with DDE server from Dymola, it can also be called directly by the user from a command window. To facilitate this, a “Distribution” directory containing all dependencies needed to run it is generated in the working directory when translating the model.

It is possible to start the Dymosim DDE server with a graphical user interface, as described in section “Graphical user interface” on page 1333, by using the batch script `ddegui.bat` in the Distribution directory.

Note that all transactions between the Dymosim DDE server and the DDE client are logged in the Dymosim window. They can also be logged on file (see later).

Simulator commands

The following commands can be sent to Dymosim using WM_DDE_EXECUTE messages:

“run”	Start simulation (if simulation is not started automatically), or resume simulation after a pause command.
“stop”	Stop simulation.
“pause”	Pause simulation. The simulation is temporarily halted until a “run” command is given. Note that DDE requests are handled while pausing.
“logon”	Enables logging to file, if logging is off.
“logoff”	Disables logging to file, if logging is on.

Matlab example: `ddeexec(channel,'run')`

Setting parameters

Parameters may be set by sending a WM_DDE_POKE message with the name of the parameter and its new value (the string representation of a number).

There are four special variables:

realtime_	Set to "1" to enable realtime mode, or to "0" to disable realtime mode.
tscale_	The scale factor between simulated time and real time. In real-time mode the simulator will maintain the relationship `real-time = tscale_ * simulated-time`
abstol_	Absolute tolerance for hot linked variables (default 0).
reltol_	Relative tolerance for hot linked variables (default 0).

Matlab example: `ddepoke(channel,'tscale_','2.0')`

When simulation is started, also simulation parameters, auxiliary variables and states may be set. However, these are not set immediately (but almost) but only at certain points in the simulation that allows for updates. Hence, when a client requests a certain e.g. variable, an update for another or the same client may be pending. Any such pending update value will be reported in the response.

Requesting variables

When simulation is started, the values of simulation parameters/variables/states etc. at the last accepted output point are available by sending a WM_DDE_REQUEST message with the name of the variable. Dymosim will then return a message with a current value of the variable etc. (the string representation of the number), or zero if no such variable exists.

It is also possible to request the value of special variables. In addition to the four special variables mentioned in the table in the previous section, the following special variables can also be requested:

delayed_	Returns the time the simulation was delayed at the last accepted output point.
status_	Returns the state of the DDE server. The state is composed as the sum of the following parts: 1 Simulation started (running) 10 Simulation paused 100 Current simulation time = 0 E.g. 11 means that the simulation is started but paused at a simulation time greater than 0.
time_	Returns the current simulation time in seconds.

Matlab example: `ddereq(channel, 'time_')`

Hot linking variables

Variables can be "hot linked" using message WM_DDE_ADVISE. The linked variables will be sent to the client at output points when a significant change has occurred. A significant change of a variable is determined from absolute and relative tolerances (settable by the DDE client) as follows (x0 = value last sent to client, x = current value):

```
absmax = max(abs(x0), abs(x));
absdiff = abs(x - x0);
changed = absmax < 1 ? absdiff > abstol_
                     : absdiff/absmax > reltol_;
```

The variable is sent to the client when "changed" is true. The variable is also sent at the first output point following the hot-link operation.

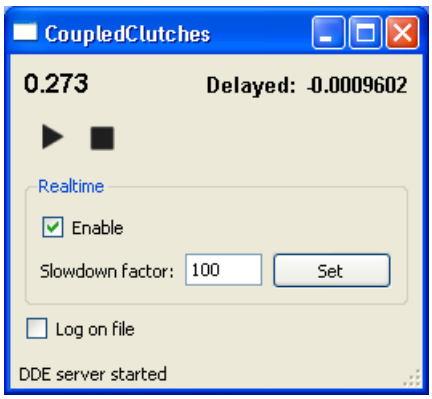
Matlab example: `ddeadv(channel, 'time_', 't=[t x];', 'x')`

Dymosim counts the number of links to a variable, and any corresponding "unlink" messages. Updates for the variable are sent while the link count is greater than zero.

Graphical user interface

By clicking on the minimized Dymosim window a graphical user interface for the Dymosim application will be displayed:

Example of Dymosim interface.



The example above shows the GUI for the model CoupledClutches, simulated with real-time (synchronization) using a slowdown factor of 100. The slowdown factor can be modified anytime. The simulation has been paused at 0.273 seconds by the user. The time the simulation is delayed at the accepted output point, relative to real time is displayed.

The simulation can be resumed by clicking on the “play” button or using the short key **Ctrl+P**. (When the simulation is running, a “pause” button is shown instead of the “play” button.) Logging of DDE events to file is not activated. The status bar shows in this example the Dymola request to Dymosim.

If the environment variable DYMOSIMGUI is defined, simulation will not start automatically when the program is executed; instead the user must give a Run command.

Logging of transactions

If the environment variable DYMOSIMLOGDDE is defined, all DDE communication to/from the simulator will be logged to a file ddelog.txt in the current directory. The file is created when the program starts, if not existing. Once Dymosim has started the setting in Dymosim GUI take precedence over DYMOSIMLOGDDE.

For each new execution of Dymosim, the logs are appended to the file, i.e. old messages are kept.

Limitations

There are some limitations of the feature:

- When clients request data from the DDE server, they specify the desired data format. Dymosim currently only support the following formats:
    - Plain text, used by e.g. Matlab.
    - XITable, used by e.g. Microsoft Office Excel.
- DDE Server cannot be combined with Export model as DLL.

11.5 OPC Communication

From Dymola 2021x, OPC communication is not supported. For alternatives, please contact support: https://www.3ds.com/support.

11.6 Code and Model Export

11.6.1 Introduction

When a model is translated in Dymola, model C-code is produced in the file `dsmodel.c`. In the normal case, this file does not contain simulation runtime code, i.e., the code needed to solve the model equations in order to compute outputs and derivatives of state variables. Instead, this simulation runtime code is included in binary link libraries (compiled using Microsoft C compilers), which are used when building `dymosim.exe` in Dymola or building a Simulink S-function. The code in the binary libraries also contains license checks that are performed when the model is executed.

To facilitate export of models from Dymola, e.g., to be able to compile the C-code for non-Windows platforms or to run the model on another computer without license check, certain export licenses are provided. These export options are **Real-time Simulation** (see Section "Real-time Simulation" starting on page 1313), **Binary Model Export**, and **Source Code Generation.** Depending on the export option available, Dymola will generate special extended versions of `dsmodel.c` during model translation.

The Binary Model Export option allows the model to be exported to other Windows computers without requiring a Dymola license at the target system. The simulation functionality of the exported model is the same as on a computer having a Dymola license.

The Source Code Generation is intended for advanced model-based development including, e.g., rapid prototyping of control systems. The code exported with this option can be used on any platform without the need of a Dymola license at the target system. A special translation command should be used with Source Code Generation and a number of built-in flags are available that can be used to modify the contents of the generated model code. The Source Code Generation option includes the functionality provided by Real-time Simulation (without restrictions) and Binary Model Export when models are translated in Dymola or Simulink.

Dassault Systèmes provides a template project on how to interface the models exported with Binary Model Export and Source Code Generation to standard integration routines to build stand-alone applications. This package is called **StandAloneDymosim**, and is described in detail in section "The StandAloneDymosim project" starting on page 1340. The simulation code provided in the StandAloneDymosim project makes use of the `dsmodel.c` model API, which is described in detail in the section "Interfacing to dsmodel.c" starting on page 1342.

Note that for large models, additional files are created to dsmodel.c (dsmodelext1.c, dsmodelext2.c, etc.). To prevent this, instead generating a larger dsmodel.c, the following flag can be set:

```
Advanced.SeparateFilesCcode=false;
```

The default value of the flag is true.

Note that some compilers may have problems with compiling large files.

Required License Options

The following license features are required for the export options

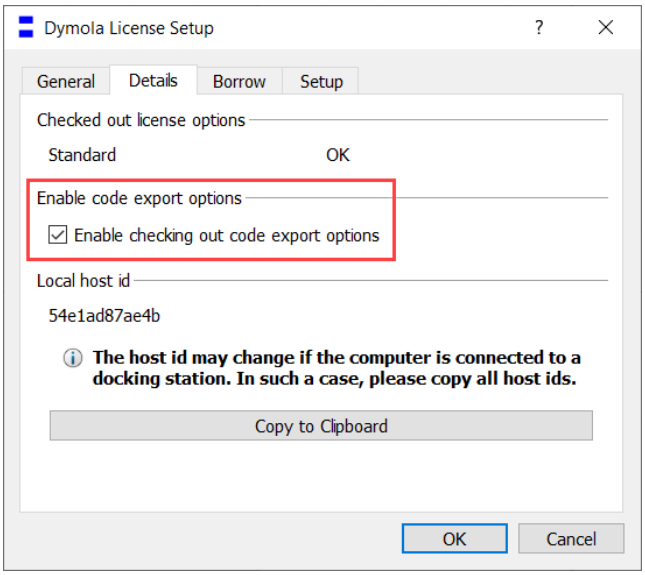
- Dymola Real-Time Simulation
- Dymola Binary Model Export
- Dymola Source Code Generation

Models developed by users that lack export options can still be run at other computers using a runtime concept. Dymola runtime concept requires the user of the model to have a Dymola license. The license file should be defined by the environment variable `DYMOLA_RUNTIME_LICENSE`, for example

```
set DYMOLA_RUNTIME_LICENSE=C:\My Documents\dymola.lic
```

Enabling Export

In the **Details** tab of the Dymola License Setup (reached through the command **Tools > License Setup**), it is possible to enable or disable the code export (BinaryModelExport and SourceCodeGeneration). Code export is by default enabled:



The reason to disable code export options is to avoid unintentionally checking out export options from a sharable license.

The code export can also be enabled or disabled by the following corresponding flag on the command line; the below is the default value

```
Advanced.EnableCodeExport = true
```

The setting to enable code export is remembered between Dymola sessions.

11.6.2 Binary Model Export

No additional simulation runtime code is added to `dsmodel.c` for Binary Model Export and the simulation applications are built by linking with Visual Studio compatible binary libraries. Instead, the main functionality provided by Binary Model Export is to disable the runtime license checking during execution of the model.

Binary Model Export allows a developer to create stand-alone applications for the following purposes:

- to generate a *dymosim.exe* application that can be executed on other Windows computers without a license.
- to generate a dymosim DLL with an extended co-simulation API (the DLL includes the Dymola DAE solvers and routines to run a simulation for a given time and to load/save simulation setups (snapshots)).
- to generate Matlab/Simulink models including DymolaBlocks to be run in other Matlab environments without requiring a license.
- to interface the model code (`dsmodel.c`) to custom integration routines and compile stand-alone applications by linking with binary libraries for the runtime routines.

For the last alternative Dassault Systèmes provides a template project, *StandAloneDymosimBinary* (see section "The StandAloneDymosim project" starting on page 1340), which shows how to interface standard solvers to the model code exported by Binary Model Export.

Running the dymosim executable with the command line option `-h` will print if binary model export is activated or if a Dymola license is required for execution. It is also displayed which libraries that the executable uses.

XML Interface

Binary Model Export and Source Code Generation (see below) supports export of symbol table information, e.g., model structure, variable names, types, and units as an XML file. The feature is enabled by setting the flag

```
Advanced.GenerateXMLInterface = true
```

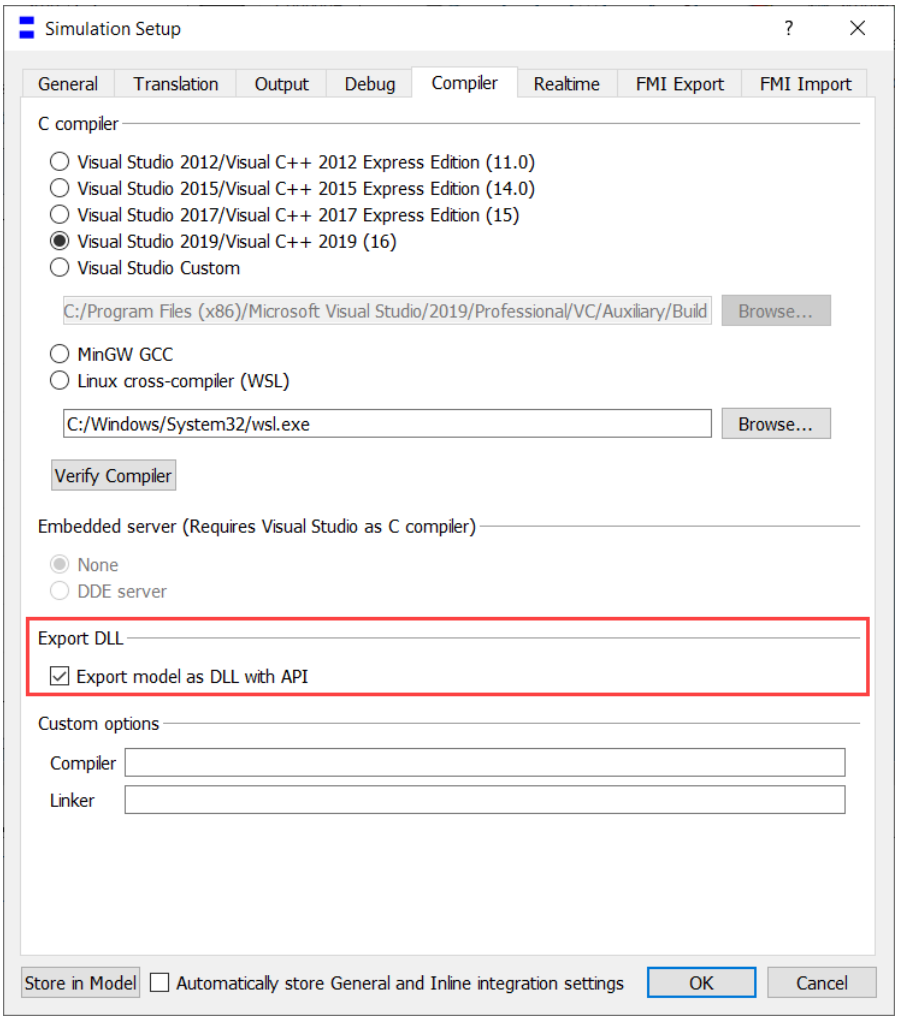
At translation an XML file with the same name as the model is generated. It contains a description of each variable on scalar level. An actual variable description may look as

```
<Variable>
  <Name>J1.phi</Name>
  <Description>Absolute rotation angle of
component</Description>
  <DataType>Real</DataType>
  <Category>State</Category>
  <Quantity>Angle</Quantity>
  <Unit>rad</Unit>
  <ValueInterpretation>
    <UnitConversion>
      <DisplayUnit>deg</DisplayUnit>
      <Gain>57.2957795130823</Gain>
      <Offset>0.0</Offset>
    </UnitConversion>
  </ValueInterpretation>
</Variable>
```

A description includes only the attributes that have been defined for the variable in the Modelica model. The description may include min and max value, but values for these attributes have not been specified for J1.phi in the example above.

Generating a dymosim DLL

With the Binary Model Export (or Source Code Generation) option, it is possible to generate a dynamic link library (dymosim.dll) from a model. To enable the DLL export, mark the checkbox **Export model as DLL with API** in the **Compiler** tab of the **Simulation Setup**, (reached by the command **Simulation > Setup**).



The API of the DLL is given by the header file `$DYMOLA/source/dymosim.h`. For examples on how to use the API (dynamically loading a model, using the snapshot solver, etc.), see the file `dymosim.c` in the same directory.

Note that Export model as DLL cannot be combined with embedded DDE server.

Important. Any new development of this functionality will be to use FMI instead since the FMI supports Co-simulation using Dymola solvers. This also means that the possibility to generate a dymosim DLL in the way it is done today will be removed from Dymola 2023x.

11.6.3 Source Code Generation

Source code generation by export

The Source Code Generation contains the functionality provided by Real-time Simulation and Binary Model Export. However, the restriction concerning inline integration for Real-time Simulation is not imposed if the user has the Source Code Generation option. Furthermore, Source Code Generation allows export of readable and well-documented code facilitating inspection, debugging, profiling, etc. This makes this export option suitable for advanced model-based applications, such as rapid prototyping.

Dymola has a special built-in function that should be used with Source Code Generation. The function is called

```
translateModelExport
```

and takes the model name as input.

When a model is translated using this function, the required simulation runtime code is automatically added to the generated `dsmodel.c` c-file in the same way as for Real-time Simulation. This removes the need to link with binary libraries. Furthermore, during translation using the `translateModelExport` function, the following three built-in Dymola flags are enabled

```
Advanced.UseModelicaNamesForC
```

To keep the original variable names in the generated C-code to make it readable (default value true).

```
Advanced.OutputEquationTrace
```

To generate comments in the generated C-code about original equation and component origin for increased traceability (default value true).

```
Advanced.SubstituteVariablesUsedOnce
```

If an intermediate variable is used only once, the right-hand side expression is substituted to remove the need to store the intermediate variable (default value `false`).

Using these flags allows for more readable, traceable, and efficient code that can be used for debug purposes. It is, however, also possible to generate standard `dsmodel.c` code by setting all flags to `false`. This can be used if it is desired to distribute the code in obscured form. Source Code Generation also enables the XML export described above.

If compilation and producing an executable is not of interest, the compilation can be disabled by setting the flag

```
Advanced.CompileAfterTranslation=false
```

This will save some time. However, note that the compilation can detect some potential errors.

As for real-time simulation, it should be noted that the simulation runtime code does not contain the most advanced routines from the binary link libraries. It is, e.g., not possible to export models with dynamic state selection. For models with dynamic state selection, `translateModelExport` gives an error during translation.

A template project, *StandAloneDymosim*, is provided to describe how to interface standard integrators to the model code exported by Source Code Generation or Binary Model Export.

Source code generation features for normal translation

A flag is available to be able to obtain the same features for normal translation as for source code generation when it comes to creating more readable, traceable, and efficient code that can be used for debug purposes.

These features correspond to the flags

```
Advanced.UseModelicaNamesForC
Advanced.OutputEquationTrace
Advanced.SubstituteVariablesUsedOnce
```

To be able to activate the corresponding features for normal translation, set

```
Advanced.SourceCodeExportNormal=true
```

The flag is by default false.

Note. To be able to use this flag:

- You must have the SourceCodeGeneration license option.
- Code export checkout must enabled in **Tools > License Setup**, the **Details** tab.

11.6.4 The StandAloneDymosim project

The *StandAloneDymosim* project is intended to show how to build stand-alone simulation applications by interfacing to `dsmodel.c`, the model code generated by Dymola. As described above, building general stand-alone applications for arbitrary platforms requires the Source Code Generation license feature to have Dymola include all necessary source code in `dsmodel.c`. The model should be translated using the command `translateModelExport` which takes the model name as input.

The project also shows how to create stand-alone applications from the `dsmodel.c` code generated with Binary Model Export by linking with the Visual Studio-compatible binary libraries included in the Dymola distribution. The Binary Model Export option will make sure that runtime license checking is disabled in the executable. On Windows, this option is only possible on platforms that are link-compatible with Visual Studio.

The example template in the project describes two options, standard Euler integration and an interface to Daskr (a successor of Dassl/Dassrt).

The project also includes a demo exemplifying how the information provided by `dsmodel.c` can be used to build sparse numeric model Jacobians to be used by sparse solvers. See section “Using sparse Jacobians” on page 1345.

Included Files

The zip-archive StandAloneDymosim.zip (located in `Program Files\Dymola 2023\bin\external`) contains two directories, *proj* and *source.* Below follows a short description of the files.

proj

- *readme.txt* : The instructions contained in this section in a condensed format.
- *StandAloneDymosim.vcproj*: Project file for Visual Studio 2012 (11.0).
    - Assumes that Dymola is installed in `C:\Program Files\Dymola`
    - Assumes that the StandAloneDymosim source files are located in `C:\dev\source\dymosim\standalone`
    - You have to modify the project if this is not the case (see compilation and linking below).
    - You should use the configuration: Win32 Release
    - The project can be imported to newer versions of Visual Studio.
- *StandAloneDymosimBinary.vcproj*: Project file to link application with binary libraries for the runtime routines.
- *StandAloneDymosim.sln*: Visual Studio 2012 (11.0) solution file.

source

- *StandAloneDymosim.c*: Main program. Contains interface, call of Euler, and call of Daskr integration.
- *inline_Int.h, inline_Int.c*: Interface to dassl-routines.
- *daux.c, ddaskr.h, ddaskr.c:* f2c converted files for ddaskr (successor of ddasrt). Separate License file provided. Available in Fortran form from www.netlib.org
- `sparse_Jacobian.h`, `sparse_Jacobian.c`: exemplify how the structural sparsity pattern provided by `dsmodel.c` can be used to construct sparse numeric Jacobians.

The main program, `StandAloneDymosim.c`, contains documentation of the routines exported by `dsmodel.c` and shows how to use these routines to simulate the model using standard Euler integration. This documentation is also given in section "Interfacing to dsmodel.c" starting on page 1342 in this document.

The main program is intended as an example, and all sections marked *CHANGE* in `StandAloneDymosim.c` indicate places where it might be a good idea to change something or, e.g., add code for external I/O.

Compilation and Linking

You should follow the steps below in order to compile and link the example code. **Note** that you will need to adapt the include paths and location of the libraries if you use the project file *StandAloneDymosim.vcproj*.

- Adjust the line (at the end of `StandAloneDymosim.c`)

```
#include "c:/my documents/dymola/dsmodel.c"
```

to refer to the correct model. Note that for big models, additional files (dsmodelext1.c, dsmodelext2.c etc.) are created; those have to be included as well.

- Add `C:\Program Files\Dymola\source` to the include path.
- Add `C:\dev\source\dymosim\standalone\daskr\solver` to the include path.
- For models exported with Binary Model Export also link with
    - `libds.lib`
    - `ModelicaExternalC.lib` (only required for some models)
- Define `INCLUDE_EULER` or `INCLUDE_DASSL` to use the different solvers.
- Define `INCLUDE_MAIN` if you want to have a main-loop with calls to the solvers.
- If you want to generate a result file usable in Dymola you can also define:
    - `GenerateResultInNonDymosim=1` (generates `dsres.mat`)
    - `GenerateResultInNonDymosim_DT=0.01` (minimum distance in result file)

The needed libraries are located in `C:\Program Files\Dymola\bin\lib`.

Interfacing to dsmodel.c

This section contains documentation of the routines exported by `dsmodel.c` that are used by the simple Euler integration routines in `StandAloneDymosim.c`.

dsblock

The main routine used to compute outputs, derivatives, etc is called `dsblock_` and has the following interface:

```
long dsblock_(long *idemand_, long *icall_,
double *time, double x_[], double xd_[], double u_[],
double dp_[], long ip_[], Dymola_bool lp_[],
double f_[], double y_[], double w_[], double qz_[],
double duser_[], long iuser_[], Dymola_bool luser_[],
long *ierr_);
```

The inputs to the function are

- `idemand_`:
    - 0: start of integration, initial equations are solved
    - 1: compute outputs `(y_)`
    - 2: compute derivatives `(f_)`
    - 3: compute auxiliary variables `(w_)`
    - 4: compute crossing functions `(qz_)`
    - 5: event handling

- 7: 'terminal()' is true

Note that, e.g., for `*idemand_==1` some derivatives may be computed. Thus it is not legal to pass `f_` as nil-pointer in that case.

- `icall_`: should normally be set to 0 before each call to `dsblock`. If you set `*icall_ > 0` it indicates that the previous call had the same inputs, except that `idemand` has increased. This can be used internally to skip redundant computations.
- `time`: time in simulation
- `x_`: state variables (input/output when *idemand_ is 0 or 5)
- `u_`: inputs (input/output when *idemand_ is 0)
- `dp_`: parameters (input)
- `f_`: derivatives (output / must be kept between calls)
- `y_`: outputs (output / must be kept between calls)
- `w_`: auxiliary variables (output / must be kept between calls)
- `qz_`: crossing functions (output / must be kept between calls)
- `duser_`: pointer to struct of real-valued simulation flags (`struct BasicDDymosimStruct`)
- `iuser_`: pointer to struct of integer-valued simulation flags (`struct BasicIDymosimStruct`)
- `*ierr_`: Output: start with `*ierr_=0`
    - 0 indicates success
    - -999 terminate integration successfully (call of terminate in model)
    - otherwise error

GetDimensions

```
void GetDimensions(long*nStates, long*nx2, long*nInputs,
long*nOutputs, long*nAuxiliary, long*nParameters,
long*nRelations,long*ncons,long*dae);
```

- `*nStates`: number of states
- `*nInputs`: number of inputs
- `*nOutputs`: number of outputs
- `*nAuxiliary`: number of additional variables
- `*nParameters`: number of parameters
- `*nRelations`: number of relations

The number of crossing functions is 2*nRelations (+1 if simple handling of timed events). The inputs and outputs correspond to top-level inputs/outputs of the model. The parameters correspond to parameters that can be changed after compilation (they must be bound to a literal value and not evaluated).

declare_

```
void declare_(double* states, double* parameters,
              void* cuser_[], long* QiErr);
```

This function is used to get default literal values of states and parameters (`cuser_` can be set to 0). The states and parameter vectors may be uninitialized, and must be at least as long as indicated by `GetDimensions`.

`*QiErr` is set to non-zero to indicate failure.

FindEvent_

```
getBasicIDymosimStruct()->mFindEvent;
```

Keep at 0 as default.

Set this flag to 1 to indicate that the solver is a fixed-step solver, and a high resolution of state events is required (higher than the step size). This causes the derivatives to be rescaled next to events, such that the states exactly hit the event point.

NextTimeEvent

```
getBasicDDymosimStruct()->mNextTimeEvent;
```

Set to `1e37` before call with `*idemand_ 0`, 4, or 5.

If the returned value is smaller, it indicates the time of the next time-event and the simulation should stop at that point and trigger an event (unless there is a state event or end of integration before).

Logging functionality

The functions below are examples of available routines for generation of log messages from the compiled models. More routines are available in `Program Files\Dymola 2023\Source\dsutil.h`

```
void DymosimMessage(const char* message);
```

Prints a message string.

```
void DymosimMessageDouble(const char* message, double d);
```

Prints a message string followed by a double value.

```
void DymosimMessageLong(const char* message, long i);
```

Prints a message string followed by a long integer.

```
void VariableChanged(char* var, double val, double t);
```

Used to print the name and value of a variable at a given time point.

Using sparse Jacobians

The example project *StandAloneDymosim* also includes a demo exemplifying how the information provided by dsmodel.c can be used to build sparse numeric model Jacobians. These Jacobians can then be used by sparse solvers for efficient simulation of large-scale models (not demonstrated here). The demo is implemented by the files `sparse_Jacobian.h` and `sparse_Jacobian.c`, which are described above.

To enable the demo, include sparse_Jacobian.c in the project and define the flag `DYN_SPARSE_JACOBIAN_TEST` (as well as `INCLUDE_EULER`). When activated a call is made in StandAloneDymosim.c to construct a sparse numeric Jacobian at the end of the explicit Euler simulation. The Jacobian is not used by explicit Euler, but the result of the simulation is used to exemplify sparse Jacobian construction.

Trouble-shooting

- When building an application at one computer and executing it at another, it is essential that the target machine has the required *redistributable runtime libraries*. These depend on the compiler used to build the application. Check the Visual Studio documentation for further details. As an example, the Visual Studio 2012 distribution contains an installer program, *vcredist_x86.exe*, for redistributable libraries.
- If you use the *StandAloneDymosim.vcproj* project, make sure to use the corresponding *Win32 Release* configuration. Debug builds may contain libraries that are not redistributable.
- Make sure to adapt the include paths and locations of libraries.
- Your license must include Source Code Generation or Binary Model Export (`DymolaSourceCodeGeneration` or `DymolaBinaryModelExport`).
- If you have the Source Code Generation feature, you should use the command `translateModelExport` to have Dymola generate model code that includes the simulation runtime routines.

12 SCRIPTING AND REPORTING

12 Scripting and Reporting

12.1 Introduction

This chapter describes external interfaces. The following external interfaces are detailed:

- Java interface for Dymola (page 1350).
- Python interface for Dymola (page 1365).
- JavaScript interface for Dymola (page 1375).
- The Dymola report generator (page 1376).

Note that information about external functions in other languages (C, Java, C++, and FORTRAN 77) is available in the chapter “Advanced Modelica Support”.

12.2 Java Interface for Dymola

The Java interface is an API for executing commands in Dymola using a Java program. It contains a number of functions to perform operations such as simulating, setting variables, plotting, and exporting data.

The Java interface is included in the Dymola distribution. Go to the installation folder for Dymola, then to the subfolder `Modelica\Library\java_interface`. The following files and folders are included:

- dymola_interface.jar
  The Java archive that contains the Java interface for Dymola.
- json-simple-1.1.1.jar
  A required third party library. JSON Simple is copyrighted under Apache License 2.0 and may be freely distributed. The license is found at: `licenses/LICENSE-2.0.txt`.
- examples
  A folder that contains examples that illustrate how to use the Java interface. Feel free to copy and modify the examples.
- doc
  This folder contains the documentation for the Java interface in Javadoc format. Double-click on the file `index.html` to view the documentation in your favourite web browser.

To quickly get started using the Java interface, you can build and run the examples that come with the Dymola distribution. You find step-by-step instructions below.

The Java interface is supported on both Windows and Linux.

Getting Started with an Example on Windows

1. Install Java Development Kit (JDK). It can be downloaded for free at http://www.oracle.com/technetwork/java/javase/downloads/index.html
   The Java interface was built with Java 6. You find JDK 6 under "Previous Releases" and then "Java SE 6". Note that you need to login in order to download Java 6 platforms.
2. Open a Command Prompt (`cmd.exe`).
3. Update your PATH environment variable to include the Java tools, for example:

   ```
   set PATH=C:\Program Files\Java\jdk1.6.0_43\bin;%PATH%
   ```

4. Update your Java CLASSPATH to include the library for the Java interface, i.e., dymola_interface.jar. An example of this could be the following:

   ```
   set CLASSPATH=%CLASSPATH%;C:\Program Files\Dymola
   2023\Modelica\Library\java_interface\dymola_interface.jar
   ```

   Also make sure that the current directory is included in the class path. If not, then include it:

   ```
   set CLASSPATH=.;%CLASSPATH%
   ```

5. Copy the example `DymolaExample.java` to another folder. You find the example in the subfolder `examples`. The reason that the example should be copied is that Windows might not allow you to create files in the distribution directory. Change current directory to the folder that contains `DymolaExample.java`.

   The Java interface will automatically find Dymola if you install it in the default location. Otherwise you need to edit the example to point to the path of the Dymola executable. Open `DymolaExample.java` and edit the line

   ```
   dymola = DymolaWrapper.getInstance();
   ```

   to take the full path to the Dymola executable as argument. For example:

   ```
   dymola = DymolaWrapper.getInstance("C:\\Dymola
   2023\\bin64\\Dymola.exe");
   ```

   Remember to use double backslashes \\ in the path.

6. Build the example:

   ```
   javac.exe DymolaExample.java
   ```

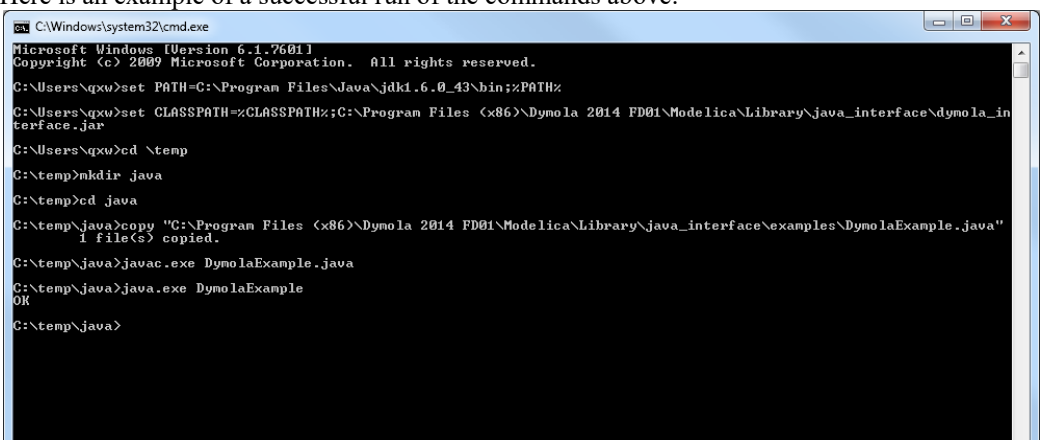
   A file `DymolaExample.class` should be generated in the same folder as `DymolaExample.java`.

7. Run the example:

   ```
   java.exe DymolaExample
   ```

   If the example ran successfully then OK should be displayed after a few seconds. If you get an error that the Dymola installation could not be found, you need to specify the path to Dymola.exe yourself. See step 5 above for how to edit the example.

   Here is an example of a successful run of the commands above:



See the Javadoc documentation for the Java interface for available functions and how to use them.

The example is reproduced below. It is also available in the Dymola distribution. You find it in the folder `Modelica\Library\java_interface\examples` as `DymolaExample.java`.

```
import com.dassault_systemes.dymola.DymolaException;
import com.dassault_systemes.dymola.DymolaInterface;
import com.dassault_systemes.dymola.DymolaWrapper;

public class DymolaExample
{
   public static void main(String[] args)
   {
          DymolaInterface dymola = null;
          try {
                 // Set this flag to false if you want Dymola to be visible.
                 // By default Dymola is hidden.
                 //DymolaWrapper.nowindow = false;

                 // Instantiate the Dymola interface and start Dymola
                 dymola = DymolaWrapper.getInstance();

                 // Call a function in Dymola and check its return value
                 boolean result =
dymola.translateModel("Modelica.Mechanics.Rotational.Examples.CoupledClutches");
                 if (!result) {
                        System.err.println("Translation failed.");
                        // Get the translation log and print it
                        String log = dymola.getLastError();
                        System.err.println(log);
                        return;
                 }

                 // Simulate the model
                 result = dymola.simulateModel("", 0, 1.2);
                 if (!result) {
                        System.err.println("Simulation failed.");
                        // Get the simulation log and print it
                        String log = dymola.getLastError();
                        System.err.println(log);
                        return;
                 }

                 // Plot a few variables
                 result = dymola.plot(new String[]{"J1.w", "J2.w", "J3.w", "J4.w"});
                 if (!result) {
                        System.err.println("Plot failed.");
                        return;
                 }

                 // Save the plot as a PNG file
                 result = dymola.ExportPlotAsImage("C:/temp/plot.png");
```

```
                if (!result) {
                        System.err.println("Failed to save the plot.");
                        return;
                }

                System.out.println("OK");
            } catch (DymolaException e) {
                System.err.println("Connection to Dymola failed. " + e.getMessage());
            } finally {
                // The connection to Dymola is closed and Dymola is terminated
                dymola = null;
            }
    }
}
```

Getting Started with an Example on Linux

Performing the above example on Linux, there are some differences:

- Installation on Linux depends on the particular Linux variant. To check which Java compiler and Java runtime environment is currently installed, do:
    - For compiler version: `javac -version`
    - For runtime environment: `java -version`

The above example on Linux (without color indication of the text):

```
import com.dassault_systemes.dymola.DymolaException;
import com.dassault_systemes.dymola.DymolaInterface;
import com.dassault_systemes.dymola.DymolaWrapper;

public class DymolaExample
{
    public static void main(String[] args)
    {

        // determine OS
        String osName = System.getProperty("os.name");
        Boolean isWindows = osName.substring(0,3).equals("Win");

        DymolaInterface dymola = null;
        try {
            // Set this flag to false if you want Dymola to be visible.
            // By default Dymola is hidden.
            //DymolaWrapper.nowindow = false;

            // Instantiate the Dymola interface and start Dymola
            dymola = DymolaWrapper.getInstance();

            // Call a function in Dymola and check its return value
            boolean result =
dymola.translateModel("Modelica.Mechanics.Rotational.Examples.CoupledClutches");
            if (!result) {
                System.err.println("Translation failed.");
```

```
            // Get the translation log and print it
            String log = dymola.getLastError();
            System.err.println(log);
            return;
        }

        // Simulate the model
        result = dymola.simulateModel("", 0, 1.2);
        if (!result) {
            System.err.println("Simulation failed.");
            // Get the simulation log and print it
            String log = dymola.getLastError();
            System.err.println(log);
            return;
        }

        // Plot a few variables
        result = dymola.plot(new String[]{"J1.w", "J2.w", "J3.w", "J4.w"});
        if (!result) {
            System.err.println("Plot failed.");
            return;
        }

        // Save the plot as a PNG file
        String plotPath;
        if (isWindows) {
            plotPath = "C:/temp/plot.png";
        } else {
            plotPath = "/tmp/plot.png";
        }
        result = dymola.ExportPlotAsImage(plotPath);
        if (!result) {
            System.err.println("Failed to save the plot.");
            return;
        }

        System.out.println("OK");
    } catch (DymolaException e) {
        System.err.println("Connection to Dymola failed. " + e.getMessage());
    } finally {
        // The connection to Dymola is closed and Dymola is terminated
        dymola = null;
    }
  }
}
```

The Java Interface

To illustrate the Java interface, consider the function `simulateModel`. Its description in Dymola is:

```
Function simulateModel "simulate a Modelica model"
  input String problem := "" "Name of model, e.g. Modelica.Mechanics
.Rotational.Components.Clutch";
```

```
  input Real startTime := 0.0 "Start of simulation";
  input Real stopTime := 1.0 "End of simulation";
  input Integer numberOfIntervals := 0 "Number of output points";
  input Real outputInterval := 0.0 "Distance between output points";
  input String method := "Dassl" "Integration method";
  input Real tolerance := 0.0001 "Tolerance of integration";
  input Real fixedstepsize := 0.0 "Fixed step size for Euler";
  input String resultFile := "dsres" "Where to store result";
  output Boolean result "true if successful";
  external "builtin";
  annotation(Documentation(info="If not done: translate a model from
Modelica into simulation code (see translateModel).
Then simulate with the given parameters"));
end simulateModel;
```

The corresponding function in the Java interface is:

simulateModel

```
boolean simulateModel(java.lang.String problem,
                      double startTime,
                      double stopTime,
                      int numberOfIntervals,
                      double outputInterval,
                      java.lang.String method,
                      double tolerance,
                      double fixedstepsize,
                      java.lang.String resultFile)
                      throws DymolaException
```

If not done: translate a model from Modelica into simulation code (see translateModel). Then simulate with the given parameters

Parameters:

`problem` - Name of model, e.g. Modelica.Mechanics.Rotational.Components.Clutch. Default `""`.
`startTime` - Start of simulation. Default `0.0`.
`stopTime` - End of simulation. Default `1.0`.
`numberOfIntervals` - Number of output points. Default `0`.
`outputInterval` - Distance between output points. Default `0.0`.
`method` - Integration method. Default `"Dassl"`.
`tolerance` - Tolerance of integration. Default `0.0001`.
`fixedstepsize` - Fixed step size for Euler. Default `0.0`.
`resultFile` - Where to store result. Default `"dsres"`.

Returns:

true if successful

Throws:

DymolaException

There is a one-to-one correspondence between the parameters in the Dymola command and the parameters in the Java method. If a parameter has a default value, it is shown in the documentation for that parameter. Commands that have default parameters are overloaded in the Java interface. The overloaded methods use the default parameter values. In the example above, simulateModel is called with three parameters:

```
result = dymola.simulateModel("", 0, 1.2);
```

This means that for the other six parameters, the default values are used. Note that the Javadoc documentation only contains the base method and not the overloaded methods, even though they exist in the interface. Also note that Java does not support named parameters.

For a complete list of available commands, see the Javadoc documentation for the Java interface. You find it in the subfolder `Modelica\Library\java_interface\doc` in the Dymola distribution. Double-click on the file `index.html` to view the documentation in your favourite web browser.

ExecuteCommand

If a command that is not a part of the Java interface should be executed, the method `ExecuteCommand` can be used. It takes a string parameter that can contain any command or expression. For example:

```
dymola.ExecuteCommand("a=1");
```

The command is not type checked so you are responsible for making sure the command is valid. It is not possible to retrieve the output from the command.

getLastError

The translation and simulation log is available through the function `getLastError`. Call it after a translate, and/or simulate, command to retrieve the log in text format. The function can also be used to retrieve other error messages. Note that `getLastError` is cleared if a new command is issued, so it should be called directly after the check/translate/simulate command. `getLastError` is defined as follows:

getLastError

```
java.lang.String getLastError()
                          throws DymolaException
```

Returns the error message from the last command. If the last command was successful an empty string is returned. For check, translate, etc, the log is returned.

Returns:
The error message from the last command.
Throws:
DymolaException

An example of usage is:

```
boolean result =
dymola.translateModel("Modelica.Mechanics.Rotational.Examples.Couple
dClutches");
if (!result) {
  System.err.println("Translation failed.");
  // Get the translation log and print it
  String log = dymola.getLastError();
  System.err.println(log);
  return;
}
```

Getting an Instance of Dymola

You get an instance of the Dymola interface by calling the method `DymolaWrapper.getInstance`. This method has a number of overloads.

```
public static DymolaWrapper getInstance()
                                    throws DymolaException

public static DymolaWrapper getInstance(int port)
                                    throws DymolaException

public static DymolaWrapper
getInstance(java.lang.String dymolaExePath)
                                    throws DymolaException

public static DymolaWrapper
getInstance(java.lang.String dymolaExePath, int port)
                                    throws DymolaException
```

By default, the port number used for communication between Dymola and the Java interface is 8082.

The first two overloads above assume that Dymola is installed in the default location. If you installed Dymola in another location, you need to use one of the two last overloads and specify the path to Dymola.exe. Please remember to use double backslashes \\ in the path.

For more details about the instantiation methods and their parameters, see the Javadoc documentation.

Exiting Dymola

Dymola will automatically exit when the Java program exits. It is good practice to enclose the interface method calls in a try/catch block and set the DymolaWrapper object to null in the `finally` block.

```
DymolaInterface dymola = null;
try {
  dymola = DymolaWrapper.getInstance();
  ...
} catch (DymolaException e) {
  System.err.println("Connection to Dymola failed. " + e.getMessage());
} finally {
  dymola = null;
}
```

Alternatively you can exit Dymola yourself by using the method `close`. It will wait until Dymola is closed before returning. The close command is useful, for example, if you want to start another instance of Dymola.

```
dymola.close();
dymola = null;
```

Note. From Dymola 2022x, you don´t need to call the method `close()` to close Dymola. Dymola is closed automatically when the script is terminated, or when the variable with the

Dymola interface falls outside its scope. (If you call the method `close()` after Dymola is closed, nothing happens.)

Commands with Two Output Parameters

The commands `simulateExtendedModel` and `simulateMultiExtendedModel` each have two output parameters. Since a Java function can have only one output parameter, the values are returned as an array of type Object[].

simulateExtendedModel

Here is an example of a call to `simulateExtendedModel` in Dymola:

```
simulateExtendedModel("Modelica.Mechanics.Rotational.Examples.C
oupledClutches", initialNames={"J1.J","J2.J"},
initialValues={2,3}, finalNames={"J1.w","J4.w"});
 = true, {6.213412958654296, 1.000000000000004}
```

The function returns two values, a Boolean status flag and a vector of values. The corresponding call in the Java interface is shown below. The output parameters are available as elements in the Object[] array.

```
Object[] output =
dymola.simulateExtendedModel("Modelica.Mechanics.Rotational.Exa
mples.CoupledClutches", 0.0, 1.0, 0, 0.0, "Dassl", 0.0001, 0.0,
"test", new String[]{"J1.J","J2.J"}, new double[]{2,3}, new
String[]{"J1.w","J4.w"}, true);

boolean status = (Boolean) output[0];

double[] values = (double[]) output[1];
double J1_w = values[0];
double J4_w = values[1];
```

simulateMultiExtendedModel

Here is an example of a call to `simulateMultiExtendedModel` in Dymola:

```
simulateMultiExtendedModel("Modelica.Mechanics.Rotational.Examp
les.CoupledClutches", initialNames={"J1.J","J2.J"},
initialValues=[2,3;3,4;4,5], finalNames={"J1.w","J4.w"});
 = true,
[6.213412958654296, 1.000000000000004;
7.483558191010655, 1.000000000000003;
8.107446379737779, 0.9999999999999931]
```

The function returns two values, a Boolean status flag and a two-dimensional array of values. The corresponding call in the Java interface is shown below:

```
double[][] initialValues = {{2,3},{3,4},{4,5}};
Object[] output =
dymola.simulateMultiExtendedModel("Modelica.Mechanics.Rotationa
l.Examples.CoupledClutches", 0.0, 1.0, 0, 0.0, "Dassl", 0.0001,
0.0, "dsres", new String[]{"J1.J","J2.J"}, initialValues, new
String[]{"J1.w","J4.w"});
```

```
boolean status = (Boolean) output[0];

double[][] values = (double[][]) output[1];

double[] result1 = values[0];
double J1_w1 = result1[0];
double J4_w1 = result1[1];

double[] result2 = values[1];
double J1_w2 = result2[0];
double J4_w2 = result2[1];

double[] result3 = values[2];
double J1_w3 = result3[0];
double J4_w3 = result3[1];
```

Extended versions of both examples are available in the file `SimulateExtendedExample.java` in the folder `Modelica\Library\java_interface\examples`.

Multithreading

The Java interface supports multithreading. It is possible to instantiate more than one Dymola and run them in parallel.

Each Dymola instance needs to have a unique port number. You can either assign a port number yourself, or let the Java interface find an available port. To set the port, call one of the overloaded `DymolaWrapper.getInstance` methods that take a port as argument. To automatically find a free port, simply call a `DymolaWrapper.getInstance` method that does not take a port.

Two Dymola instances that are run simultaneously cannot share the same working directory. You need to assign a unique working directory to each instance. Use the interface method `cd` to set the working directory.

The Java interface is not thread-safe. The only exception is `DymolaWrapper.getInstance`. Each thread should run its own instance of Dymola.

Below is an example that illustrates how to use multithreading in the Java interface (one example on Windows, one on Linux). The example on Windows is available as `MultithreadingExample.java` in the folder `Modelica\Library\java_interface\examples`.

Example on Windows:

```
import java.io.File;

import com.dassault_systemes.dymola.DymolaException;
import com.dassault_systemes.dymola.DymolaInterface;
import com.dassault_systemes.dymola.DymolaWrapper;

public class MultithreadingExample
```

```
{
	public static class CoupledClutchesThread implements Runnable
	{
		@Override
		public void run()
		{
			DymolaInterface dymola = null;
			try {
				System.out.println("1: Starting Dymola instance");
				dymola = DymolaWrapper.getInstance();
				System.out.println("1: Using port " + ((DymolaWrapper)
dymola).portnumber);

				String path = "C:/temp/Dymola/CoupledClutches";
				System.out.println("1: Change working directory to " + path);
				File folder = new File(path);
				if (!folder.exists()) {
					folder.mkdirs();
				}
				boolean result = dymola.cd(path);
				if (!result) {
					System.err.println("1: Failed to change working
directory");
				}

				System.out.println("1: Simulating model");
				result =
dymola.simulateModel("Modelica.Mechanics.Rotational.Examples.CoupledClutches");
				if (!result) {
					System.err.println("1: Simulation failed");
					String log = dymola.getLastError();
					System.err.println(log);
				}

				System.out.println("1: Plotting");
				result = dymola.plot(new String[] { "J1.w", "J2.w", "J3.w",
"J4.w" });
				if (!result) {
					System.err.println("1: Plot failed");
				}

				System.out.println("1: Saving the plot");
				result = dymola.ExportPlotAsImage(path + "/plot.png");
				if (!result) {
					System.err.println("1: Failed to save the plot");
				}

				System.out.println("1: Saving log");
				dymola.savelog(path + "/log.txt");

				System.out.println("1: OK");
			} catch (DymolaException e) {
				System.err.println("Connection to Dymola failed. " +
e.getMessage());
```

```
                } finally {
                        dymola = null;
                }
        }
    }

    public static class FullRobotThread implements Runnable
    {
        @Override
        public void run()
        {
                DymolaInterface dymola = null;
                try {
                        System.out.println("2: Starting Dymola instance");
                        dymola = DymolaWrapper.getInstance();
                        System.out.println("2: Using port " + ((DymolaWrapper)
dymola).portnumber);

                        String path = "C:/temp/Dymola/fullRobot";
                        System.out.println("2: Change working directory to " + path);
                        File folder = new File(path);
                        if (!folder.exists()) {
                                folder.mkdirs();
                        }
                        boolean result = dymola.cd(path);
                        if (!result) {
                                System.err.println("2: Failed to change working
directory");
                        }

                        System.out.println("2: Simulating model");
                        result =
dymola.simulateModel("Modelica.Mechanics.MultiBody.Examples.Systems.RobotR3.fullRobot");
                        if (!result) {
                                System.err.println("2: Simulation failed");
                                String log = dymola.getLastError();
                                System.err.println(log);
                        }

                        System.out.println("2: Plotting");
                        result = dymola.plot(new String[] { "mechanics.q[1]",
"mechanics.q[2]" });
                        if (!result) {
                                System.err.println("2: Plot failed");
                        }

                        System.out.println("2: Saving the plot");
                        result = dymola.ExportPlotAsImage(path + "/plot.png");
                        if (!result) {
                                System.err.println("2: Failed to save the plot");
                        }

                        System.out.println("2: Saving log");
                        dymola.savelog(path + "/log.txt");
```

```
                        System.out.println("2: OK");
                    } catch (DymolaException e) {
                        System.err.println("Connection to Dymola failed. " +
e.getMessage());
                    } finally {
                        dymola = null;
                    }
                }
        }

        public static void main(String[] args)
        {
                // Set this flag to false if you want Dymola to be visible.
                // By default Dymola is hidden.
                //DymolaWrapper.nowindow = false;

                Thread coupledClutchesThread = new Thread(new CoupledClutchesThread());
                Thread fullRobotThread = new Thread(new FullRobotThread());

                coupledClutchesThread.start();
                fullRobotThread.start();
        }
}
```

Example on Linux (without color indication of text):

```
import java.io.File;

import com.dassault_systemes.dymola.DymolaException;
import com.dassault_systemes.dymola.DymolaInterface;
import com.dassault_systemes.dymola.DymolaWrapper;

public class MultithreadingExample
{
    static String basePath;

      public static class CoupledClutchesThread implements Runnable
      {
            @Override
            public void run()
            {
                  DymolaInterface dymola = null;
                  try {
                        System.out.println("1: Starting Dymola instance");
                        dymola = DymolaWrapper.getInstance();
                        System.out.println("1: Using port " + ((DymolaWrapper)
dymola).portnumber);
                String path = basePath + "/CoupledClutches";
                        System.out.println("1: Change working directory to " +
path);
                        File folder = new File(path);
                        if (!folder.exists()) {
                              folder.mkdirs();
```

```
                        }
                        boolean result = dymola.cd(path);
                        if (!result) {
                              System.err.println("1: Failed to change working
directory");
                        }

                        System.out.println("1: Simulating model");
                        result =
dymola.simulateModel("Modelica.Mechanics.Rotational.Examples.CoupledClutches");
                        if (!result) {
                              System.err.println("1: Simulation failed");
                              String log = dymola.getLastError();
                              System.err.println(log);
                        }

                        System.out.println("1: Plotting");
                        result = dymola.plot(new String[] { "J1.w", "J2.w", "J3.w",
"J4.w" });
                        if (!result) {
                              System.err.println("1: Plot failed");
                        }

                        System.out.println("1: Saving the plot");
                        result = dymola.ExportPlotAsImage(path + "/plot.png");
                        if (!result) {
                              System.err.println("1: Failed to save the plot");
                        }

                        System.out.println("1: Saving log");
                        dymola.savelog(path + "/log.txt");

                        System.out.println("1: OK");
                  } catch (DymolaException e) {
                        System.err.println("Connection to Dymola failed. " +
e.getMessage());
                  } finally {
                        dymola = null;
                  }
            }
      }

      public static class FullRobotThread implements Runnable
      {
            @Override
            public void run()
            {
                  DymolaInterface dymola = null;
                  try {
                        System.out.println("2: Starting Dymola instance");
                        dymola = DymolaWrapper.getInstance();
                        System.out.println("2: Using port " + ((DymolaWrapper)
dymola).portnumber);
                String path = basePath + "/fullRobot";
```

```
                        System.out.println("2: Change working directory to " +
path);
                        File folder = new File(path);
                        if (!folder.exists()) {
                              folder.mkdirs();
                        }
                        boolean result = dymola.cd(path);
                        if (!result) {
                              System.err.println("2: Failed to change working
directory");
                        }

                        System.out.println("2: Simulating model");
                        result =
dymola.simulateModel("Modelica.Mechanics.MultiBody.Examples.Systems.RobotR3.fullRobot"
);
                        if (!result) {
                              System.err.println("2: Simulation failed");
                              String log = dymola.getLastError();
                              System.err.println(log);
                        }

                        System.out.println("2: Plotting");
                        result = dymola.plot(new String[] { "mechanics.q[1]",
"mechanics.q[2]" });
                        if (!result) {
                              System.err.println("2: Plot failed");
                        }

                        System.out.println("2: Saving the plot");
                        result = dymola.ExportPlotAsImage(path + "/plot.png");
                        if (!result) {
                              System.err.println("2: Failed to save the plot");
                        }

                        System.out.println("2: Saving log");
                        dymola.savelog(path + "/log.txt");

                        System.out.println("2: OK");
                  } catch (DymolaException e) {
                        System.err.println("Connection to Dymola failed. " +
e.getMessage());
                  } finally {
                        dymola = null;
                  }
            }
      }

      public static void main(String[] args)
      {
            // Set this flag to false if you want Dymola to be visible.
            // By default Dymola is hidden.
            //DymolaWrapper.nowindow = false;
```

```
            // determine proper base path
            String osName = System.getProperty("os.name");
            if (osName.substring(0,3).equals("Win")) {
                  basePath = "C:/temp/Dymola";
            } else {
                  basePath = "/tmp/Dymola";
            }

            Thread coupledClutchesThread = new Thread(new CoupledClutchesThread());
            Thread fullRobotThread = new Thread(new FullRobotThread());

            coupledClutchesThread.start();
            fullRobotThread.start();
      }
}
```

12.3 Python Interface for Dymola

The Python interface is an API for executing commands in Dymola using a Python program. It contains a number of functions to perform operations such as simulating, setting variables, plotting, and exporting data.

The Python interface is included in the Dymola distribution. Go to the installation folder for Dymola, then to the subfolder `Modelica\Library\python_interface`. The following files and folders are included:

- dymola.egg
  The Python package that contains the Python interface for Dymola. The package is tested with Python 3.9.
- examples
  A folder that contains examples that illustrate how to use the Python interface. Feel free to copy and modify the examples.
- doc
  This folder contains the documentation for the Python interface in Sphinx format. Double-click on the file `index.html` to view the documentation in your favourite web browser.

To quickly get started using the Python interface, you can build and run the examples that come with the Dymola distribution. You find step-by-step instructions below.

The Python interface is supported on Windows and Linux. The instructions for Windows in this section apply to Linux as well, with adjustments for the Linux structure. For example:

- The Dymola binary is supposed to be located as `/usr/local/bin/dymola` if the path is omitted when using the interface.
- The PYTHONPATH should for the Linux version of this release be set according to:

  ```
  PYTHONPATH=$PYTHONPATH:/opt/Dymola-2023-
  x86_64/Modelica/Library/python_interface/dymola.egg
  ```

Getting Started with an Example

1. Install Python 3.9. It can be downloaded for free at http://www.python.org/download/.
2. Open a Command Prompt (`cmd.exe`).
3. Update your PATH environment variable to include the Python environment, for example:

   ```
   set PATH=C:\Python39;C:\Python39\Scripts;C:\Python39\Tools\
   Scripts;%PATH%
   ```

4. Update your PYTHONPATH to include the package with the Python interface, i.e., dymola.egg. An example of this could be the following:

   ```
   set PYTHONPATH=%PYTHONPATH%;C:\Program Files\Dymola
   2023\Modelica\Library\python_interface\dymola.egg
   ```

   Note that the path to `dymola.egg` should not be surrounded by double-quotes.

5. Copy the example `DymolaExample.py` to another folder. You find the example in the subfolder `examples`. The reason that the example should be copied is that Windows might not allow you to create files in the distribution directory. Change current directory to the folder that contains `DymolaExample.py`.

   The Python interface will automatically find Dymola if you install it in the default location. Otherwise you need to edit the example to point to the path of the Dymola executable. Open `DymolaExample.py` and edit the line

   ```
   dymola = DymolaInterface()
   ```

   to take the full path to the Dymola executable as argument. For example:

   ```
   dymola = DymolaInterface("C:\\Dymola 2023\\bin64\\Dymola.exe")
   ```

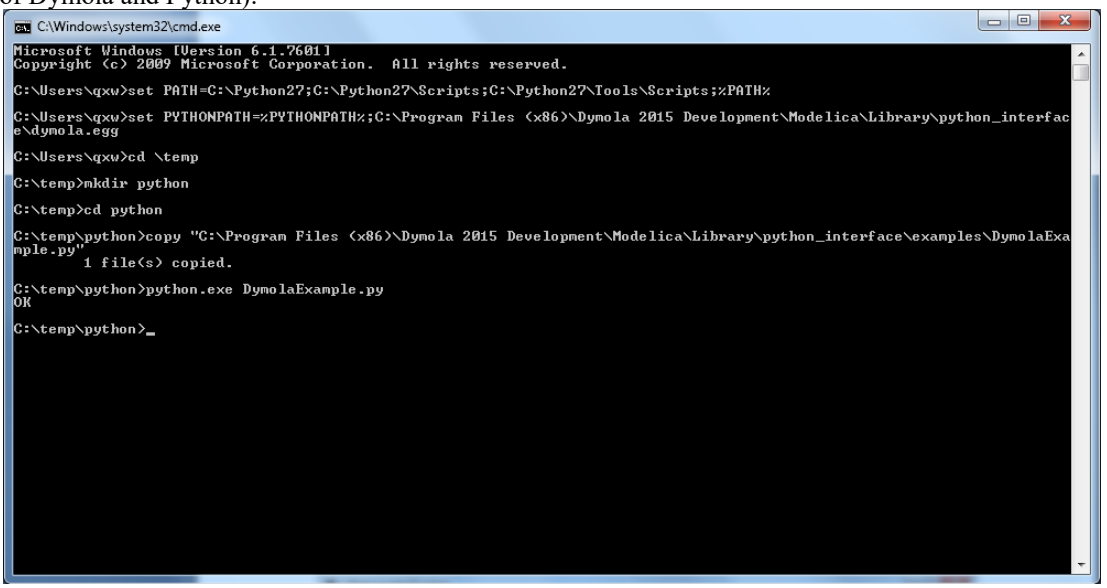
   Remember to use double backslashes \\ in the path.

6. Run the example:

   ```
   python.exe DymolaExample.py
   ```

   If the example ran successfully then OK should be displayed after a few seconds. If you get an error that the Dymola installation could not be found, you need to specify the path to Dymola.exe yourself. See step 5 above for how to edit the example. Below an example of a successful run of the commands above (note, this in from an earlier version

of Dymola and Python):



See the documentation for the Python interface for available functions and how to use them.

The example is reproduced below. It is also available in the Dymola distribution. You find it in the folder `Modelica\Library\python_interface\examples` as `DymolaExample.py`.

```
from dymola.dymola_interface import DymolaInterface
from dymola.dymola_exception import DymolaException

dymola = None
try:
    # Instantiate the Dymola interface and start Dymola
    dymola = DymolaInterface()

    # Call a function in Dymola and check its return value
    result =
dymola.simulateModel("Modelica.Mechanics.Rotational.Examples.Coupl
edClutches")
    if not result:
        print("Simulation failed. Below is the translation log.")
        log = dymola.getLastError()
        print(log)
        exit(1)

    dymola.plot(["J1.w", "J2.w", "J3.w", "J4.w"])
    dymola.ExportPlotAsImage("C:/temp/plot.png")
    print("OK")
except DymolaException as ex:
    print("Error: " + str(ex))
finally:
    if dymola is not None:
```

```
dymola.close()
dymola = None
```

The Python Interface

To illustrate the Python interface, consider the function `simulateModel`. Its description in Dymola is:

```
Function simulateModel "simulate a Modelica model"
  input String problem := "" "Name of model, e.g. Modelica.Mechanics
.Rotational.Components.Clutch";
  input Real startTime := 0.0 "Start of simulation";
  input Real stopTime := 1.0 "End of simulation";
  input Integer numberOfIntervals := 0 "Number of output points";
  input Real outputInterval := 0.0 "Distance between output points";
  input String method := "Dassl" "Integration method";
  input Real tolerance := 0.0001 "Tolerance of integration";
  input Real fixedstepsize := 0.0 "Fixed step size for Euler";
  input String resultFile := "dsres" "Where to store result";
  output Boolean result "true if successful";
  external "builtin";
  annotation(Documentation(info="If not done: translate a model from
Modelica into simulation code (see translateModel).
Then simulate with the given parameters"));
end simulateModel;
```

The corresponding function in the Python interface is:

simulateModel(*problem=''*, *startTime=0.0*, *stopTime=1.0*, *numberOfIntervals=0*, *outputInterval=0.0*, *method='Dassl'*, *tolerance=0.0001*, *fixedstepsize=0.0*, *resultFile='dsres'*)

If not done: translate a model from Modelica into simulation code (see `translateModel()`). Then simulate with the given parameters

Parameters:

- **problem** (*str*) - Name of model, e.g. Modelica.Mechanics.Rotational.Components.Clutch. Default `""`.
- **startTime** (*float*) - Start of simulation. Default `0.0`.
- **stopTime** (*float*) - End of simulation. Default `1.0`.
- **numberOfIntervals** (*int*) - Number of output points. Default `0`.
- **outputInterval** (*float*) - Distance between output points. Default `0.0`.
- **method** (*str*) - Integration method. Default `"Dassl"`.
- **tolerance** (*float*) - Tolerance of integration. Default `0.0001`.
- **fixedstepsize** (*float*) - Fixed step size for Euler. Default `0.0`.
- **resultFile** (*str*) - Where to store result. Default `"dsres"`.

Returns: true if successful

Raises : `DymolaException`

There is a one-to-one correspondence between the parameters in the Dymola command and the parameters in the Python method. If a parameter has a default value, it is shown in the

documentation for that parameter. Note that in the Python interface documentation, default values and array dimensions are formatted as in Modelica. An example of a call to simulateModel is:

```
result = dymola.simulateModel("MotorDriveTest", 0, 1.2,
resultFile="MotorDriveTest")
```

For a complete list of available commands, see the documentation for the Python interface. You find it in the subfolder Modelica\Library\python_interface\doc in the Dymola distribution. Double-click on the file index.html to view the documentation in your favourite web browser.

ExecuteCommand

If a command that is not a part of the Python interface should be executed, the method `ExecuteCommand` can be used. It takes a string parameter that can contain any command or expression. For example:

```
dymola.ExecuteCommand("a=1")
```

The command is not type checked so you are responsible for making sure the command is valid. It is not possible to retrieve the output from the command.

getLastError

The translation and simulation log is available through the function `getLastError`. Call it after a translate, and/or simulate, command to retrieve the log in text format. The function can also be used to retrieve other error messages. Note that `getLastError` is cleared if a new command is issued, so it should be called directly after the check/translate/simulate command. `getLastError` is defined as follows:

getLastError()

Returns the error message from the last command. If the last command was successful an empty string is returned. For check, translate, etc, the log is returned.

Returns: The error message from the last command.

Raises : DymolaException

An example of usage is:

```
result =
dymola.translateModel("Modelica.Mechanics.Rotational.Examples.C
oupledClutches")
if not result:
    print("Translation failed.")
    log = dymola.getLastError()
    print(log)
```

Getting an Instance of Dymola

You get an instance of the Dymola interface by initializing `DymolaInterface`.

```
dymola.dymola_interface.DymolaInterface(dymolapath="", port=-1,
showwindow=False, debug=False, allowremote=False,
nolibraryscripts=False)
```

By default, the port number used for communication between Dymola and the Python interface is 8082.

For more details about the instantiation methods and their parameters, see the documentation.

Exiting Dymola

You exit Dymola by calling the method `DymolaInterface.close()`.

It is good practice to enclose the interface calls in a try/except block and close Dymola in the `finally` block.

```
dymola = None
try:
    dymola = DymolaInterface()
    ...
except DymolaException as ex:
    print("Error: " + str(ex))
finally:
    if dymola is not None:
        dymola.close()
        dymola = None
```

Note. From Dymola 2022x, you don´t need to call the method `close()` to close Dymola. Dymola is closed automatically when the script is terminated, or when the variable with the Dymola interface falls outside its scope. (If you call the method `close()` after Dymola is closed (as in the example above), nothing happens.)

Commands with Two Output Parameters

The commands `simulateExtendedModel` and `simulateMultiExtendedModel` each have two output parameters.

simulateExtendedModel

Here is an example of a call to `simulateExtendedModel` in Dymola:

```
simulateExtendedModel("Modelica.Mechanics.Rotational.Examples.C
oupledClutches", initialNames={"J1.J","J2.J"},
initialValues={2,3}, finalNames={"J1.w","J4.w"});
 = true, {6.213412958654296, 1.000000000000004}
```

The function returns two values, a Boolean status flag and a vector of values. The corresponding call in the Python interface is shown below. The output parameters are available as elements in a list.

```
output =
dymola.simulateExtendedModel("Modelica.Mechanics.Rotational.Exa
mples.CoupledClutches", 0.0, 1.0, 0, 0.0, "Dassl", 0.0001, 0.0,
"test3", ["J1.J", "J2.J"], [2, 3], ["J1.w", "J4.w" ], True)
```

```
status = output[0]

values = output[1]
J1_w = values[0]
J4_w = values[1]
```

simulateMultiExtendedModel

Here is an example of a call to `simulateMultiExtendedModel` in Dymola:

```
simulateMultiExtendedModel("Modelica.Mechanics.Rotational.Examp
les.CoupledClutches", initialNames={"J1.J","J2.J"},
initialValues=[2,3;3,4;4,5], finalNames={"J1.w","J4.w"});
 = true,
[6.213412958654296, 1.000000000000004;
7.483558191010655, 1.000000000000003;
8.107446379737779, 0.9999999999999931]
```

The function returns two values, a Boolean status flag and a two-dimensional array of values. The corresponding call in the Python interface is shown below:

```
initialValues = [[2, 3], [3, 4], [4, 5]]
output =
dymola.simulateMultiExtendedModel("Modelica.Mechanics.Rotationa
l.Examples.CoupledClutches", 0.0, 1.0, 0, 0.0, "Dassl", 0.0001,
0.0, "dsres", ["J1.J", "J2.J"], initialValues, ["J1.w",
"J4.w"])

status = output[0]

values = output[1]

result1 = values[0]
J1_w = result1[0]
J4_w = result1[1]

result2 = values[1]
J1_w = result2[0]
J4_w = result2[1]

result3 = values[2]
J1_w = result3[0]
J4_w = result3[1]
```

Extended versions of both examples are available in the file `SimulateExtendedExample.py` in the folder `Modelica\Library\python_interface\examples`.

Multithreading

The Python interface supports multithreading. It is possible to instantiate more than one Dymola and run them in parallel.

Each Dymola instance needs to have a unique port number. You can either assign a port number yourself, or let the Python interface find an available port. To set the port, instantiate

`DymolaInterface` with the `port` argument. To automatically find a free port, instantiate `DymolaInterface` either without the `port` argument, or set `port` to `-1`.

Two Dymola instances that are run simultaneously cannot share the same working directory. You need to assign a unique working directory to each instance. Use the interface method `cd` to set the working directory.

The Python interface is not thread-safe. The only exception is the instantiation method for `DymolaInterface`. Each thread should run its own instance of Dymola.

Below is an example that illustrates how to use multithreading in the Python interface. The example is available as `MultithreadingExample.py` in the folder `Modelica\Library\python_interface\examples`.

```
import os
import threading

from dymola.dymola_interface import DymolaInterface
from dymola.dymola_exception import DymolaException

def CoupledClutchesThread():
    dymola = None
    try:
    print("1: Starting Dymola instance")
    dymola = DymolaInterface()
    print("1: Using Dymola port " + str(dymola._portnumber))

        path = "C:/temp/Dymola/CoupledClutches"
        print("1: Change working directory to " + path)
        try:
                os.makedirs(path)
        except OSError as ex:
                print("1: " + str(ex))
        result = dymola.cd(path)
        if not result:
                print("1: Failed to change working directory")

        print("1: Simulating model")
        result =
dymola.simulateModel("Modelica.Mechanics.Rotational.Examples.CoupledClutches")
        if not result:
            print("1: Simulation failed")
            log = dymola.getLastError()
            print(log)

        print("1: Plotting")
        result = dymola.plot(["J1.w", "J2.w", "J3.w", "J4.w"])
        if not result:
            print("1: Plot failed")

        print("1: Saving the plot")
        result = dymola.ExportPlotAsImage(path + "/plot.png")
```

```
        if not result:
                print("1: Failed to save the plot")

        print("1: Saving log")
        dymola.savelog(path + "/log.txt")

        print("1: OK")
    except DymolaException as ex:
        print("1: Error: " + str(ex))
    finally:
        if dymola is not None:
            dymola.close()
            dymola = None

def FullRobotThread():
    dymola = None
    try:
        print("2: Starting Dymola instance")
        dymola = DymolaInterface()
        print("2: Using Dymola port " + str(dymola._portnumber))

        path = "C:/temp/Dymola/fullRobot"
        print("2: Change working directory to " + path)
        try:
                os.makedirs(path)
        except OSError as ex:
                print("2: " + str(ex))
        result = dymola.cd(path)
        if not result:
                print("2: Failed to change working directory")

        print("2: Simulating model")
        result =
dymola.simulateModel("Modelica.Mechanics.MultiBody.Examples.Systems.RobotR3.fullRobot")
        if not result:
                print("2: Simulation failed")
                log = dymola.getLastError()
                print(log)

        print("2: Plotting")
        result = dymola.plot(["mechanics.q[1]", "mechanics.q[2]"])
        if not result:
                print("2: Plot failed")

        print("2: Saving the plot")
        result = dymola.ExportPlotAsImage(path + "/plot.png")
        if not result:
                print("2: Failed to save the plot")

        print("2: Saving log")
        dymola.savelog(path + "/log.txt")

        print("2: OK")
    except DymolaException as ex:
```

```
            print("2: Error: " + str(ex))
        finally:
            if dymola is not None:
                dymola.close()
                dymola = None


if __name__ == '__main__':
    coupled_clutches_thread = threading.Thread(target=CoupledClutchesThread)
    coupled_clutches_thread.daemon = True
    coupled_clutches_thread.start()

    full_robot_thread = threading.Thread(target=FullRobotThread)
    full_robot_thread.daemon = True
    full_robot_thread.start()

    coupled_clutches_thread.join()
    full_robot_thread.join()
```

12.4 JavaScript interface for Dymola

The class DymolaInterface provides a JavaScript API for accessing the most useful built-in functions in Dymola.

To use the JavaScript interface, Dymola must be started specifying server port 8082, for example by adding this port as the last part of **Target** in a shortcut for starting Dymola:

```
…\Dymola.exe" -serverport 8082
```

There is a one-to-one correspondence between the parameters in a Dymola command and the parameters in the corresponding JavaScript method. **Note** that JavaScript does not support named parameters.

If you want to execute a command that is not part of the JavaScript interface, you can use the method `ExecuteCommand`. It takes a string parameter that can contain any command or expression. For example:

```
dymola.ExecuteCommand("a=1");
```

The command is not type checked so you are responsible for making sure that the command is valid. It is not possible to retrieve the output from the command.

The JavaScript interface is supported on Windows and Linux. The instructions for Windows in this section apply to Linux as well, with adjustments for the Linux structure.

The JavaScript interface has been tested on Firefox, Google Chrome, and Internet Explorer 11.

Below an example of how to use the JavaScript interface:

```
try {
     var dymola = new DymolaInterface();
     var result =
dymola.simulateModel("Modelica.Mechanics.Rotational.Examples.CoupledClutches");
     if (result) {
          result = dymola.plot(["J1.w", "J2.w", "J3.w", "J4.w"]);
          if (result) {
               result = dymola.ExportPlotAsImage("C:/temp/plot.png");
          }
     } else {
          alert("Simulation failed.");
          var log = dymola.getLastError();
          alert(log);
     }
} catch (e) {
     alert("Exception: " + e);
}
```

For more information about the JavaScript interface, open the file `DymolaInterface.html`, located in

```
Program Files\Dymola 2023\Modelica\Library\
javascript_interface\doc
```

with your favorite browser.

12.5 Report generator

12.5.1 Fundamentals

In Dymola a report generator is available. It is based on Dymola running as a server. It enables a HTML page loaded in a browser to call Modelica functions using JavaScript. It is possible to insert a model diagram, change parameters, simulate a model, show plots, show animations, etc. It can be used as a "notebook" since it´s possible to re-execute function calls, for example to make a simulation with changed parameters and observe the changed plots. The resulting report can then be stored and sent to anyone (the reader does not need Dymola to read the report).

XHTML can be used, as well as HTML5, SVG, WebGL, MathML and X3D (successor standard for VRML). For example, 3D animations can be made directly in a browser using X3DOM which renders 3D animations represented as X3D. It is written in JavaScript and uses WebGL. For animation commands available in the animation, see section "Mouse and keyboard commands available for animation in reports" on page 1382 below.

In order to allow calling Modelica functions from HTML, a web server version of Dymola has been developed. All **functions** available in the package DymolaCommands are possible to call from JavaScript in the client web browser. An automatic JavaScript generator has been developed. It creates the code for the parameter exchange.

Important. To support the report generator, Dymola must be started specifying server port 8082, for example by adding this port as the last part of **Target** in a shortcut for starting Dymola:

```
…\Dymola.exe" -serverport 8082
```

Note! The report generator is currently only supported on Windows.

The report generator has been tested on Firefox, Google Chrome, and Internet Explorer 11.

12.5.2 JavaScript functions

A set of special report JavaScript functions has been developed which are suitable to include in the HTML code. When the HTML page is opened, the browser communicates with Dymola to retrieve various information, such as model diagrams, plots, and animations. This information is inserted in the HTML page. It is possible to save the HTML code including this information for use without having Dymola running. It is also possible to re-execute a function call, for example to re-run a simulation after changing parameters.

The functions add content (innerHTML) of HTML div-blocks. The structure of the functions is:

```
insertXXX(result_block, model, ...);
```

The `id` of the div-block is a parameter `block-_id`. The model path is given as the parameter `model`. A typical structure of a HTML page is thus:

```
<p>Text</p>
<div id="diagram"></div>
<script type="text/javascript">insertDiagram(diagram, "MyModel", "svg");
```

```
</script>
```

The functions are:

insertDiagram(result_block, model, format, width, height)

Inserts a Modelica diagram.

The `format` is either "`PNG`" or "`SVG`". The dimensions in pixels are given by `width` and `height`.

insertIcon(result_block, model, format, width, height)

Inserts a Modelica icon.

The `format` is either "`PNG`" or "`SVG`". The dimensions in pixels are given by `width` and `height`.

insertText(result_block, model)

Inserts pretty printed Modelica text.

The annotations are omitted from the Modelica text.

insertClass(result_block, model, width, height)

Inserts a Modelica text editor for a given model.

The text of the model can be edited and submitted to Dymola. If the model is read-only, the editor is disabled and the model is not possible to edit.

The dimensions in pixels are given by `width` and `height`.

insertEquations(result_block, model, format)

Inserts the equations and algorithms of a Modelica model.

The `format` is either "`PNG`" or "`MathML`".

insertDocumentation(result_block, model, width, height)

Inserts the formatted documentation of a Modelica model.

The dimensions in pixels are given by `width` and `height`.

insertParameterDialog(result_block, model)

Inserts an editor for the top-level parameters in a model.

The parameter values can be changed and submitted to Dymola.

insertCommand(result_block, width, height)

Inserts a command window.

The bottom part is a command-line where any command may be entered. The top part shows the result.

The dimensions in pixels are given by `width` and `height`.

insertPlot(result_block, model, variables, format, width, height)

Inserts a plot.

The array of variables to plot is given by `variables`.

The `format` is either "`PNG`" or "`SVG`". The dimensions in pixels are given by `width` and `height`.

insertVariableValue(model, variable, time)

Inserts a variable value. The value is read from the result file.

The variable path is given by `variable`. The time in seconds is given by `time`.

insertSignalOperatorValue(model, variable, signalOperator)

Inserts a signal operator value.

The variable path is given by `variable`. The `signalOperator` is an enumeration value. Here is a list of available signal operators:

```
SignalOperator.Min
SignalOperator.Max
SignalOperator.ArithmeticMean
SignalOperator.RectifiedMean
SignalOperator.RMS
SignalOperator.ACCoupledRMS
SignalOperator.SlewRate
SignalOperator.THD
SignalOperator.FirstHarmonic
```

insertAnimation(result_block, model, format, width, height)

Inserts an animation.

The animation is automatically running. You can rotate the animation object by pressing left button and moving the mouse. Pan by also pressing **Ctrl**. Zoom by pressing **Alt**.

The `format` supported is "`X3D`". The dimensions in pixels are given by `width` and `height`.

The following utility function is also available:

setClassText(package_path, Modelica_text);

Creates or changes a Modelica class.

The complete text definition of a Modelica class is given. It can be inserted in a package. If the `package_path` is an empty string, a top level class is created.

12.5.3 Example of HTML report sections

Below a small example of how a HTML report can look like:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
   <meta http-equiv="Content-Type" content="application/xhtml+xml; charset=utf-8"/>
   <title>Dymola Report</title>
   <link rel="stylesheet" type="text/css" href="dymola_report.css"/>
   <script type="text/javascript" src="utils.js"></script>
   <script type="text/javascript" src="dymola_interface.js"></script>
   <script type="text/javascript" src="dymola_report.js"></script>
   <script type="text/javascript"
src="http://cdn.mathjax.org/mathjax/latest/MathJax.js?config=TeX-AMS-
MML_HTMLorMML"></script>
</head>

<body>
   <h1>Dymola Report</h1>
   <p>This is a sample report to demonstrate the Dymola Report features.</p>

   <h2>Diagram</h2>
   <p>The model diagram for the Modelica.Blocks.Examples.PID_Controller is shown
below:</p>
   <div id="dymola_example_diagram"></div>
   <script
type="text/javascript">insertDiagram(document.getElementById("dymola_example_diagram
"), "Modelica.Blocks.Examples.PID_Controller", "svg");</script>

   <h2>Plot</h2>
   <p>The angular velocities of
Modelica.Mechanics.Rotational.Examples.CoupledClutches are shown below:</p>
   <div id="dymola_example_plot"></div>
   <script
type="text/javascript">insertPlot(document.getElementById("dymola_example_plot"),
"Modelica.Mechanics.Rotational.Examples.CoupledClutches", ["J1.w", "J2.w", "J3.w",
"J4.w"], "svg", 600, 350);</script>

   <h2>Animation</h2>
   <p>The animation view of
Modelica.Mechanics.MultiBody.Examples.Systems.RobotR3.fullRobot is shown below:</p>
   <div id="dymola_example_animation"></div>
   <script
type="text/javascript">insertAnimation(document.getElementById("dymola_example_anima
tion"), "Modelica.Mechanics.MultiBody.Examples.Systems.RobotR3.fullRobot", "xhtml",
600, 300);</script>

   <div id="dymola_report_created"></div>
</body>
</html>
```

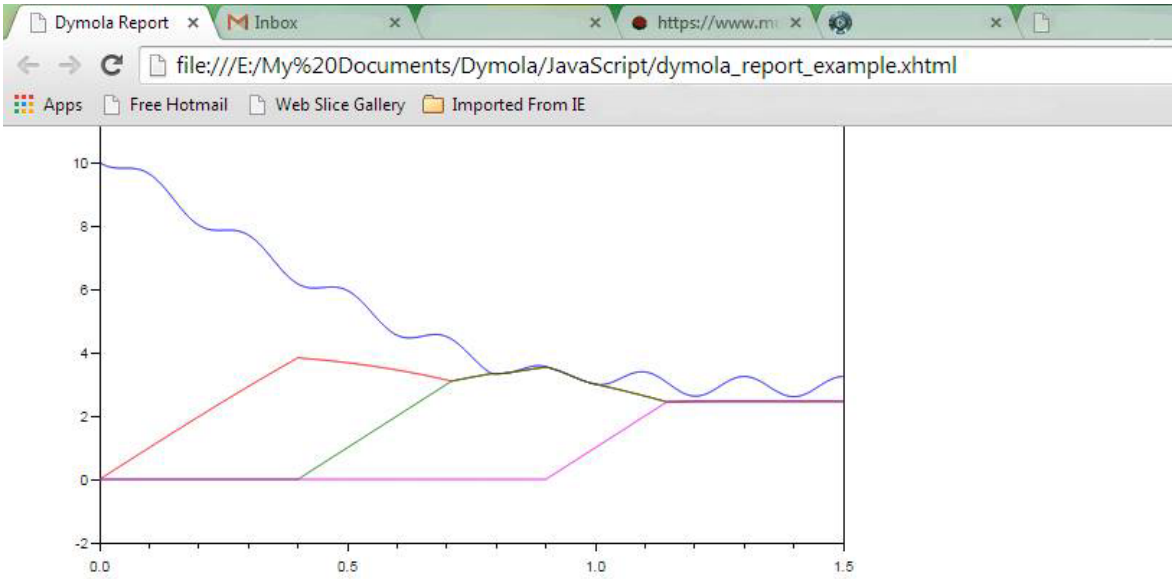
This example only includes a model diagram, a plot and an animation. For an example with more features, please open the file `dymola_report_example.xhtml` in the folder `Modelica\Library\javascript_interface` in the distribution. Note that this file also displays the resulting report.

For more information about the report generator, open the file `global.html`, located in

```
Program Files\Dymola 2023\Modelica\Library\
java_interface\doc_report
```

with your favorite browser.

An example of how a report could look like when generated is:



Variable Value

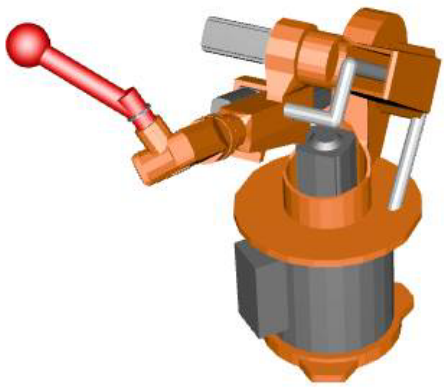
The value of J1.w in Modelica.Mechanics.Rotational.Examples.CoupledClutches at time 0.21 is 7.94091.

Signal Operators

The model Modelica.Mechanics.Rotational.Examples.CoupledClutches has been simulated from 0 to 1.5 s. Max for J1.a is 10 and min is -19.8244.

Animation

The animation view of Modelica.Mechanics.MultiBody.Examples.Systems.RobotR3.fullRobot is shown below:





Successfully created Dymola report on Mon Sep 15 2014 09:59:12 GMT+0200 (Romance Daylight Time)



12.5.4 Mouse and keyboard commands available for animation in reports

The implementation of X3DOM for animation in reports provides some generic interaction and navigation methods. Navigation is controlled by specific predefined modes.

Examine mode (activate with key "e")

Function	Button
Rotate	Left Button / Left Button + Shift
Pan	Mid Button / Left Button + Ctrl
Zoom	Right Button / Wheel / Left Button + Alt
Set center of rotation	Left Button double-click

Walk mode (activate with key "w")

Function	Button
Move forward	Left Button
Move backward	Right Button

Fly mode (activate with key "f")

Function	Button
Move forward	Left Button
Move backward	Right Button

Look at mode (activate with key "l")

Function	Button
Move closer	Left Button
Move back	Right Button

Non-interactive movement

Function	Button
Reset view	r
Show all	a
Upright	u

description). There is a corresponding flag `Advanced.FMI.Integrate` that has the same alternatives as the `integrate` parameter.

13 ADVANCED MODELICA SUPPORT

13 Advanced Modelica Support

13.1 Declaring functions

In Modelica it is possible to define and use functions, and the functions can be used in equations of a model, in defining parameters, and called interactively from the command line.

Functions inherently reduce the possibility for symbolic manipulations, and should thus not be used unless necessary. In particular index reduction requires that all functions are differentiated, which cannot be done in a straightforward way without help. This "help" is provided in form of an annotation giving the derivative of the function.

The functions themselves can either be written in Modelica or as external functions in C, Java, C++ or FORTRAN 77. In most cases the external function is provided in a binary library with a corresponding header defining the function.

13.2 User-defined derivatives

In order to reduce the index it is necessary to differentiate arbitrary expressions with respect to time. This derivative must be very accurate in order to not introduce unnecessary numerical errors. The functions are seldom simple scalar functions from one scalar input, but instead

have vectors and/or records as input, in many case the functions also have several inputs and/or outputs. Often second order differentials are also needed.

For solving non-linear system of equations derivatives can also increase performance, by allowing us to compute an analytical Jacobian.

The first choice is of course to rely on the automatic differentiation of functions in Dymola.

If this is not sufficient, there is an annotation available for the user to explicitly define the derivative of a function, and Dymola can use this both to reduce the index and to compute Jacobians for non-linear system of equations.

The annotation declaring the derivative function can be given both for functions written in Modelica as well as for external functions.

13.2.1 Analytic Jacobians

For non-linear systems of equations it is possible to avoid numeric Jacobians and instead rely on Dymola to automatically differentiate the functions. Compared to writing derivative functions this is much easier for the modeler, easier to understand for the user, and also considerably less error-prone.

In order to enable Dymola's automatic differentiation feature, the modeler writing functions must declare the smoothness of the function by providing a smoothOrder annotation corresponding to the smooth operator in Modelica. A basic limitation of automatic differentiation is that it can provide a derivative even at points where the function does not have a derivative. Verifying that a function with branches (if-statements, if-expressions, or while-statements) is continuous is a difficult problem. The person providing the smoothOrder annotation is guaranteeing that the function is at least that smooth. When using the function, its derivative is only constructed if it is found to be needed because of index reduction or to generate an analytic Jacobian.

The basics of automatic differentiation and the implementation choices in Dymola are discussed in H. Olsson, H. Tummescheit and H. Elmqvist: "Using automatic differentiation for partial derivatives in Modelica", Proceedings of the 4th International Modelica Conference, Hamburg-Harburg, Germany, 2005, pp. 105-112.

Example

We will use a simple function that just inverts a strictly positive number for illustration:

```
function MyDivision
  input Real x;
  output Real y;
  annotation (smoothOrder=1000);
algorithm
  assert(x>0, "x should be positive");
  y:=1/x;
end MyDivision;
```

We then write a simple model where this function must be differentiated in order to solve a non-linear equation:

```
model TestDivision3
  Real x;
equation
 MyDivision(x)=1+time;
end TestDivision3;
```

Translating this example gives a translation log with:

```
…
Sizes of nonlinear systems of equations: {1}
Sizes after manipulation of the nonlinear systems: {1}
Number of numerical Jacobians: 0
```

An analytic Jacobian is automatically constructed and used since needed. The derivative function is:

```
function P.TestDivision3.MyDivision:derf
  input Real x;
protected
  Real y;
public
  input Real x_der2;
  output Real y_der2;
algorithm
  assert(x > 0, "x should be positive");
  y_der2 :=  -x_der2/x^2;
annotation (smoothOrder=999);
end P.TestDivision3.MyDivision:derf;
```

Removing the smoothOrder annotation instead gives:

```
…
Sizes of nonlinear systems of equations: {1}
Sizes after manipulation of the nonlinear systems: {1}
Number of numerical Jacobians: 1
```

13.2.2 How to declare a derivative

The following define how to declare the derivative to a function, and finally how to verify that the derivative is consistent with the function. It is strongly influenced by forward mode automatic differentiation and well suited for differentiation with respect to one variable, as in index reduction.

It can furthermore be used to efficiently compute all interesting derivatives in a straightforward way as will be explained later.

Important.

Note It is today recommended for Modelica functions to use the annotation `smoothOrder` instead of the annotation `derivative` – the `smoothOrder` annotation lets Dymola construct the derivative function automatically. For more information about `smoothOrder`, see the Modelica Language Specification.

A function declaration can have an annotation derivative specifying the derivative function with an optional order attribute indicating the order of the derivative (default 1), e.g.:

```
function f0 annotation(derivative=f1); end f0;
function f1 annotation(derivative(order=2)=f2); end f1;
function f2 end f2;
```

It is also necessary to write the derivative function for a given function, this is described in a procedural form below, and with examples that make it clearer.

The lookup for the derivative annotation follow the normal lookup rules of Modelica.

Inlining of derivative functions

Functions with annotations that indicate that they have inverses or can be differentiated are by default inlined after that information is used.

This behavior can be disabled by setting

```
Advanced.Translation.DelayInliningForInverseDerivatives = false;
```

Disabling inlining using the annotation `Inline=false` or specifying it using any of the annotations `LateInline=true` or `InlineAfterIndexReduction=true` takes precedence over this behavior.

First order derivative

The inputs to the derivative function of order 1 are constructed as follows:

- First are all inputs to the original function listed and after them, in order, one derivative for each input containing Real variables is appended.
- The outputs are constructed by starting with an empty list and then in order appending one derivative for each output containing Real variables.

As an example consider the following:

```
function foo0
  input Real x;
  input Boolean linear;
  input Real z;
  output Real y;
algorithm
  if linear then
    y:=z+x;
  else
    y:=z+sin(x);
  end if;
  annotation(derivative=foo1);
end foo0;

function foo1
  input Real x;
  input Boolean linear;
  input Real z;
  input Real der_x;
  input Real der_z;
  output Real der_y;
  annotation(derivative(order=2)=foo2);
```

```
algorithm
   der_y:=der_z+(if linear then der_x else cos(x)*der_x);
end foo1;
```

This implies that given the following equation

$$y(t) = foo0(x(t), b, z(t))$$

we know that

$$\dot{y}(t) = foo1(x(t), b, z(t), \dot{x}(t), \dot{z}(t))$$

A more complex example involving records and matrices is

```
record R
  Real M[2,2];
  Real x[2];
end R;

function recordFunction
  input R x;
  output Real y[2];
algorithm
  y:=x.M*x.x;
  annotation(derivative=recordFunction_d);
end recordFunction;

function recordFunction_d
  input R x;
  input R der_x;
  output Real der_y[2];
algorithm
   der_y:=x.M*der_x.x+der_x.M*x.x;
   // Since (A*B)'=A'*B+A*B' for matrices.
end recordFunction_d;
```

Thus if

```
y(t)=recordFunction(x(t));
```

we have

```
der(y(t))=recordFunction_d(x(t),der(x(t)));
```

Second and higher order derivatives

If the Modelica function is a nth derivative (n>=1) the derivative annotation indicates the (n+1):th derivative, and order=n+1.

The input arguments are amended by the (n+1):th derivatives, which are constructed in order from the nth order derivatives.

The output arguments are similar to the output argument for the n:th derivative, but each output is one higher in derivative order.

Continue the example above with:

```
function foo1
  ...
  annotation(derivative(order=2)=foo2);
  ...
end foo1;

function foo2
  input Real x;
  input Boolean linear;
  input Real z;
  input Real der_x;
  input Real der_z;
  input Real der_2_x;
  input Real der_2_z;
  output Real der_2_y;
algorithm
  der_2_y:=der_2_z+(if linear then der_2_x else
      cos(x)*der_2_x-sin(x)*der_x^2);
end foo1;
```

This allows us to conclude that

$$\ddot{y}(t) = \text{foo2}\left(x(t), b, z(t), \dot{x}(t), \dot{z}(t), \ddot{x}(t), \ddot{z}(t)\right)$$

Restrictions

An input or output to the function may be any predefined type (Real, Boolean, Integer or String) or a record, provided the record does not contain both real and non-real predefined types. Allowing mixed records would require that we automatically constructed a new record from the parts containing real types, which would be difficult to describe.

The function must have at least one input containing a real type, since we must have something to take the derivative with respect to.

The output list of the derivative function may not be empty, since we otherwise have no derivative. This can occur if the function e.g. returns a Boolean value.

Verifying Derivatives

In order to verify that a derivative is consistent with the function it is recommended to follow the following test procedure. The basic idea is to compare the integral of the derivative with the original function.

Assume one has a model using foo0:

```
model B
  Real x;
equation
   x=foo0(time, false, -0.1*time);
end B;
```

and we want to verify that the derivative of foo0 is correct. We do that by extending B as follows:

```
model VerifyFirstDerivative
  extends B;
  Real y;
equation
  der(y)=der(x);
initial equation
   y=x;
end VerifyFirstDerivative;
```

That the derivative is correct can be verified by comparing x (which is computed directly) and y (which is computed as an integral of the derivate). By setting second derivatives equal one can verify the second derivative as well. Note that this procedure does not modify the original model and can therefore be used even when the input arguments to the function are given internally.

13.3 External functions in other languages

13.3.1 C

In addition to functions written in Modelica, Dymola also allows external functions written in ANSI/ISO C. For each external function it is necessary to declare a Modelica interface. This declaration provides the required information needed to call the function from a Modelica model, and in some simple cases provide argument conversion.

Existing libraries may require wrapper functions.

External functions are declared as Modelica functions, but with a body that defined the interface. The Modelica specification defines the details of how function arguments in Modelica are mapped to similar data types in C, and how values returned from the functions are mapped back to Modelica types. When the interface of the external function does not match the Modelica specification, a wrapper function must be written in C to perform the required conversions.

In most cases the external function is provided in a binary library with a corresponding header file declaring the function. In order to support this one can specify a header, overriding the usual definition of the function, and a library that will automatically be linked with.

Note – when using external functions in parallel code, it is assumed that the functions are not thread-safe. For handling the case that they are, see the chapter “Simulating a model”, section “Improving simulation efficiency”, subsection “Simulation speed-up” (or use index entry “*thread-safe: external functions*” in this manual).

Including external functions

In simple cases it is possible to translate the C code of the function with the model itself. The main advantage of this approach is that it does not require any additional effort to build a library. The disadvantage is that definitions in the implementation of the C function may interfere with the generated model code and cause the compilation to fail.

As an example we will use the following trivial function that returns the sum of two real numbers. Its implementation in ANSI/ISO C is called `add2.c`.

```
#ifndef ADD2_C
#define ADD2_C
double add2(double x, double y)
{
    return x + y;
}
#endif
```

This function requires a declaration that provides a mapping between Modelica and C, and also specifies the name of the file containing the implementation.

```
function add2 "Sum of two numbers"
  input Real x, y;
  output Real sum;
  external "C";
  annotation(Include="#include <add2.c>");
end add2;
```

The first two declarations define the input and output arguments and their types in the Modelica context. The `external` declaration identifies this as an external function. The types of the parameters in the external function must be compatible with the Modelica specification.

The `Include` annotation is a string which in this case includes the implementation of the function. The contents of the `Include` string is inserted into the generated C code, hence it should contain valid C code. It can even contain header line breaks in order to include several files or even preprocessor macros. There is no guarantee that the header will only be included once, and thus necessary to guard against multiple inclusion with `#ifndef` and `#endif` wrapping.

The code (`add2.c`) should be located either in the current directory, in a relative location (`#include <../source/add2.c>`), or in directory `dymola/source`. However, using Modelica Standard Library version 3.2 and later the annotation `IncludeDirectory` can be used to define where the code is located. Please also see example below.

Linking with external library

This section refers only to the Windows version of Dymola.

For most application it is best to build a library with function definitions using software development tools outside of the Dymola environment, and then compile the model linking with the library. In this way libraries are easily shared between various applications.

A major benefit compared to including function definitions is that the risk of interference between the code generated by Dymola and the code of the external functions is greatly reduced. The only parts included in the compilation of the model are header files declaring the external functions in C, and the implementation is compiled separately.

DLLs are handled by linking with their wrapper libraries, or directly. Please see section “Linking with DLLs” below.

Building an external library

Using an external library the C code consists of two parts: a header file declaring the function, and an implementation. It is common to use a single header file to declare a group of related functions. The header file for the example above would be:

```
#ifndef ADD2_H
#define ADD2_H
extern double add2(double x, double y);
#endif
```

The implementation is very similar to the code used for inclusion in the model code, but the header file should be included to ensure compatibility in the event of changes in the interface. The `#ifndef` and `#endif` wrappers are not needed.

```
#include <add2.h>
double add2(double x, double y)
{
    return x + y;
}
```

Library annotation

The Modelica interface uses two special annotations, `Include` and `Library`, to specify the header file and the name of the library:

```
function add2 "Sum of two numbers"
  input Real x, y;
  output Real sum;
  external "C";
  annotation(Include="#include <add2.h>", Library="ext");
end add2;
```

Prefix and extension are not given.

Note that the library name is "ext"; the "lib" prefix is added by the linker, and the extension depends on the used compiler (`.lib` for Microsoft C). This ensures portability of the Modelica interface to different platforms and compilers.Note that the syntax in the `Include` annotation for linking with multiple static libraries is `Library={"lib1","lib2","lib3"}`

As a more complex example consider an interface to National Instruments `AI_VRead` in its Ni-Daq library. A protected variable is used to pick up the status code returned from the function.

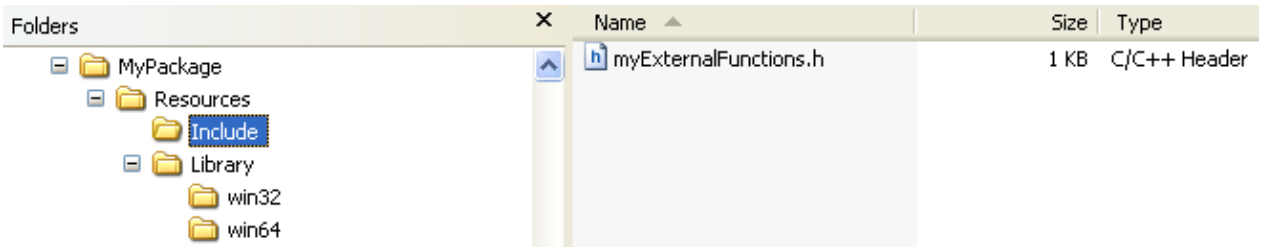
```
function AI_VRead "Analog in"
  annotation (
    Include="#include <nidaqex.h>",
    Library={"nidaq32"});
  input Integer iDevice=1;
  input Integer iChannel=1;
  input Integer iGain=1;
  output Real dVoltage;
protected
  Integer iStatus;
  external "C" iStatus = AI_VRead(iDevice, iChannel, iGain,
    dVoltage);
end AI_VRead;
```

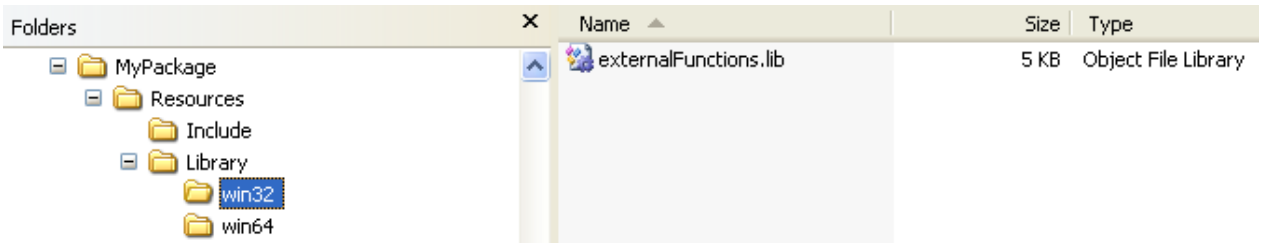
The Library annotation is either a single string or a vector or strings that name several binary libraries and the compiler will link with all listed libraries.

Note that for this example to work the header and library files must be in the search path of the compiler. This could be accomplished by placing the header in `%DYMOLA%/source` and the library in the correct sub-directory of `%DYMOLA%/bin` or by placing both of them in the current directory. However, using Modelica Standard Library version 3.2 and later the annotations `IncludeDirectory` and `LibraryDirectory` can be used to define where the header and library files are located; as in the next example.

Using annotations for location of external library and include files, and recommended file locations

Wanting to distribute a package MyPackage with external C functions, a recommended way of locating the header and implementation files is:

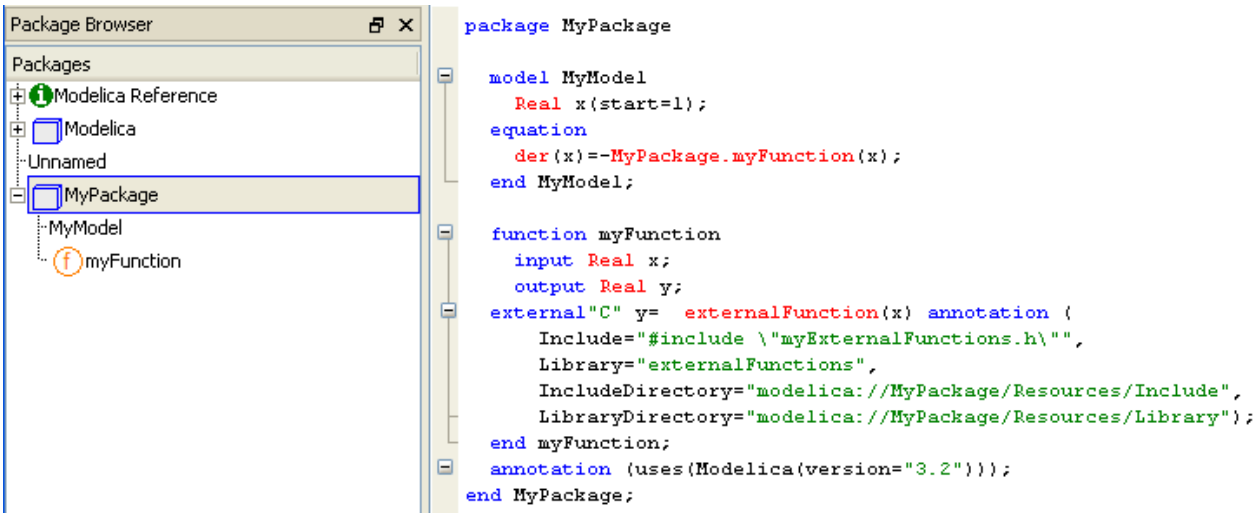




(The win64 folder is analogous to the win32 one.)

To specify these locations the annotations `IncludeDirectory` and `LibraryDirectory` can be used.

A package `MyPackage` with a model `MyModel` calling a function `myFunction` that uses an external C function can be written:



For more information, (and examples) please see "Modelica Language Specification", version 3.4, section 12.9.4. This document is available using the command **Tools > Help Documents** in Dymola.

Important! The location of the external libraries must be the recommended one (in win32/win64 subfolder), otherwise the compilation might not be the wanted one. The flag Advanced.CompileWith64 can be used to decide if models should be compiled as 32-bit or 64-bit executables. The default value 0 of the flag states that if external libraries only exists as 32-bit or 64-bit, the variant that allows linking is selected. However, if a 64-bit external library is not located in the win64 subfolder, it is compiled as 32-bit anyway.

Support for different external C++ library versions

Dymola has improved support for different C++ library versions for different version of the Visual Studio compiler. This eases the use of some commercial libraries.

This support is standardized in Modelica – by automatically switching from win32 (or win64) to the relevant subdirectory, e.g. vs2012 (for Microsoft Visual Studio 2012). Note that this is specific for external C++ code, since there is normally no need to have multiple versions of C libraries.

Different versions of the MinGW gcc compiler are also supported, for example subdirectory gcc530 for gcc version 5.3.0. However, different flavours of gcc such as TDG-GCC are not supported and hence subdirectories such as gcctdm530 will be ignored.

A warning appears if mismatch occurs.

Linking with DLLs

Linking with DLLs by linking with their wrapper libraries

DLLs can be handled by linking with their wrapper libraries. The library must be multithreaded and with DLL-linking with the I/O (/DLL switch).

The library specified using `annotation(Library="MyLib")` is on Windows normally a .lib file (i.e. "MyLib.lib") in a directory specified by the annotation `LibraryDirectory`, its default value, or in the win32/win64 subfolder. Note the ***Important!*** note about recommended location above.

If the `LibraryDirectory` contains "MyLib.dll" file Dymola will automatically delay-load the corresponding DLL file (using "-delay..."). This ensures that the DLL is not loaded when generating the default parameter settings.

Linking with DLLs by direct linking

Direct linking with a DLL (without import-library) is supported when all the following is fulfilled:

- The specified lib-file does not exist.
- A corresponding dll-file exists.
- There is an external function call.
- There is no include-annotation.

13.3.2 Java

Note – the below implementation of Java is intended for calling Java functions from Modelica, or Modelica functions from Java functions. A newer Java interface is available for accessing Dymola remotely, please see, in the chapter "Scripting and Reporting", section "Java interface for Dymola".

Dymola can call functions written in Java - either interactively or from models. This requires that you:

- Install either Java Development Kit or J2SE Software Development Kit. It is not sufficient to install the Java Runtime Environment.
- Ensure that the environment variable JAVA_HOME points to this directory.
- For each function written in Java that you want to call from Modelica you add an external declaration in Modelica.

Note! The above Java implementation supports 64-bit Windows. It uses JDK 1.8.0_122. (The newer Java interface supports both Windows and Linux.)

Calling Java functions from Modelica

The functions must be static member functions and the syntax in Modelica is:

```
function f
  input Real u;
  output Real y;
  external "Java" y='MyPackage.MyClass.Myfunction'(u);
end f;
```

In Java this function can be implemented as:

```
package MyPackage;
```

```
public class MyClass
{
   // A simple function in Java
   public static double Myfunction(double d)
   {
          return d * 2;
   }
}
```

Note: Dymola just performs a normal function call, and thus have user interaction from functions written in Java dialogs must use **modal** dialogs (i.e. they only return once the action is complete).

Mapping of data structures

Simple types in Modelica (e.g. Real) are normally mapped to corresponding simple types in Java. Strings are non-simple objects in Java, the mapping is still direct, and the character mapping is made simpler by the fact that Java Virtual Machine and Dymola internally use the same UCS-8 implementation of Unicode strings.

Note: The UCS-8 mapping is the result of applying the UTF-8 mapping to UTF-16 strings, and the recommendation is that even though it can be used internally in programs it should not be used for interfaces. In this case we make an exception in order to be compatible with the pre-existing C interface of the Java Virtual Machine.

Arrays in Modelica correspond to (possibly nested) arrays in Java. Heterogeneous arrays in Java cannot be sent to Modelica. Zero-sized matrices sent from Java should be avoided (if possible) since the non-zero dimensions cannot always be completely determined.

Records in Modelica are mapped to a class implementing a map interface in Java. This ensures that the semantics of Modelica records (named based type equivalence) is preserved.

To summarize we first present how arguments are mapped when calling a function written in Java from Modelica.

Modelica	**Java**	
	normal case	**for record contents**
Real	double	java.lang.Double
Integer	int	java.lang.Integer
Boolean	boolean	java.lang.Boolean
String	java.lang.String	java.lang.String
record	com.dynasim.record	com.dynasim.record
Real[]	double[]	double[]
Integer[]	int[]	int[]
Boolean[]	boolean[]	boolean[]

String[]	java.lang.String[]	java.lang.String[]
record[]	com.dynasim.record[]	com.dynasim.record[]

The record class, com.dynasim.record implements the map interface, and the content is mapped as above (the difference is for simple data types, since the simple data types are not objects in Java). However, for easy access to simple variables there are also special functions, getDouble, getInt, and getBoolean.

The map for records is straightforward to use and by being name based avoid issues with declaration order and future extensions of the records in Modelica.

Mapping of errors

Exceptions thrown from Java functions called from Modelica are automatically mapped to assertions, which is the normal error handling primitive in Modelica. Currently an assertion stop Dymola's interpreter as there is no way of catching the error inside Modelica.

Calling Modelica functions from Java functions

For interactive calls to Modelica functions it is possible to call Dymola's interpreter from Java, by using the function com.dynasim.dymola.interpretMainStatic. Calls of Modelica functions (and Dymola's API functions callable as Modelica functions) from Java go through one function accessible in Java as com.dynasim.dymola.interpretMainStatic, found in `Program Files\Dymola 2023\Modelica\Library\java`. The exact details are explained later. For other Modelica functions a wrapper in Java can be constructed in a mechanical way that maps arguments, calls this bridge function, and maps the result.

Having one entry point to Modelica from Java makes it straightforward to transparently redirect all calls to a remote instance of Dymola, i.e. remote method invocation.

Dymola's interface for Java functions is found in `Program Files\Dymola 2023\Modelica\Library\java`, this directory should be included in the class path when compiling Java programs that use the Dymola interface, e.g.:

```
javac –classpath
".;c:\Program Files\Dymola 2023\Modelica\Library\java" *.java
```

This has to be adapted depending on where Dymola is installed. There are also other settings in the Java compilers for making this easier to use, but we will not consider these here. The quotes around the path are used to ensure that the string is one command line argument.

Thus as examples we will implement additional functions using standard Modelica functions:

```
package MyPackage;
public class CallDymola
{

   // A function calling Dymola:
   public static double Norm(double[][] m, double p)
   {
          // Construct argument list.
          Object objs[] = new Object[2];
          objs[0] = m;
          objs[1] = new Double(p);
          // Call function
          Object res = com.dynasim.dymola.interpretMainStatic(
            "Modelica.Math.Matrices.norm", objs);
          // Go back
          double returnValue = ((Double)(res)).doubleValue();
          // Additional computations
          return returnValue;
   }

   // A matrix to test the norm
   public static double[][] Test()
   {
          double[][] d=new double[2][];
          d[0] = new double[2];
          d[1] = new double[2];
          d[0][0] = 1;
          d[0][1] = 2;
          d[1][0] = 3;
          d[1][1] = 4;
          return d;
   }

   // Using records.
   //
   // The functions getDouble, getInt, getBoolean
   // are also useful
   public static double[][] Rotate90()
   {
          Object objs[] = new Object[3];
          objs[0] = new Integer(1);
          objs[1] = new Double(1.57);
          objs[2] = new Double(0);
          com.dynasim.record res = (com.dynasim.record)(
        com.dynasim.dymola.interpretMainStatic(
           "Modelica.Mechanics.MultiBody.Frames.axisRotation",
           objs));
          return (double[][])(res.get("T"));
   }
}
```

To test all of these functions in Dymola we need:

```
package TestJava
  function f
    input Real u;
    output Real y;
    external "Java" y='MyPackage.MyClass.Myfunction'(u);
  end f;

  function Rotate90
    output Real y[3,3];
    external "Java" y='MyPackage.CallDymola.Rotate90'();
  end Rotate90;

  function Test
    output Real y[:,:];
    external "Java" y='MyPackage.CallDymola.Test'();
  end Test;

  function Norm
    input Real u[:,:];
    input Real p;
    output Real y;
    external "Java" y='MyPackage.CallDymola.Norm'(u,p);
  end Norm;
end TestJava;
```

Mapping of data structure

The mapping when a function in Java calls interpretMainStatic is identical to the mapping of record contents when calling functions written in Java. This is necessary since the simple types such as double are not objects and thus cannot be part of the generic argument list of interpretMainStatic.

Modelica	**Java**
Real	java.lang.Double
Integer	java.lang.Integer
Boolean	java.lang.Boolean
String	java.lang.String
record	com.dynasim.record
Real[]	double[]
Integer[]	int[]
Boolean[]	Boolean[]
String[]	java.lang.String[]
record[]	com.dynasim.record[]

The contents of records are also mapped in this way, and this mapping is identical to the contents of records when calling functions written in Java.

Mapping of errors

When an assertion (or other error) is trigged in Modelica originating from a call to interpretMainStatic this is mapped to an exception in Java as follows:

- com.dynasim.DymolaException base-class of the other exception – introduced in order to make it easy to catch all exceptions.
- com.dynasim.DymolaNoSuchFunction(<name of function>) when the function is not found by interpretMainStatic.
- com.dynasim.DymolaIllegalArgumentException for problems with transforming results or argument between Java and Dymola, and incorrect type of arguments to function.
- com.dynasim.DymolaEvaluationException when evaluation fails – e.g. assertions and division by zero.

These exception classes all inherit from java.lang.RunTimeException, this ensures that no 'throws' clause is needed for routines calling Dymola functions.

This corresponds to the Modelica environment where a function does not have to declare whether it may fail (e.g. using assert).

Accessing Dymola remotely

A newer interface for accessing Dymola remotely is available, please see the chapter "Scripting and Reporting", section "Java Interface for Dymola".

13.3.3 C++

Functions written in C++ are supported provided the C compiler supports cross-linkage with C++. When using languages other than C, provisions must be made to ensure that the required runtime libraries are linked.

Functions written in C++ must declared as `extern "C"` to be linkage compatible with C. Wrapper functions are needed for example to use virtual functions or other C++ features that are not present in C.

13.3.4 FORTRAN

Functions written in FORTRAN 77 are supported provided the C compiler supports cross-linkage with FORTRAN. When using languages other than C, provisions must be made to ensure that the required runtime libraries are linked.

FORTRAN code can be linked in two ways. Perhaps the most straight-forward approach is to convert the FORTRAN code to C using a tool called `f2c`. This tool translates the code into portable C code, and also includes libraries for common FORTRAN runtime routines. The alternative is to use a link compatible FORTRAN compiler.

In either case, wrapper functions are most likely required to map argument types.

13.4 Means to control the selection of states

Dymola supports automatic state selection according to the specification of Modelica.

Variables being subtypes of Real has an attribute, stateSelect, to give hints or even imperatively control the selection of variables to use as continuous time state variables.

Note that Modelica allows the state selection to be separated from the specification of initial conditions. The fixed attribute should exclusively be used for specifying start conditions and it should not influence the selection of states at all.

13.4.1 Motivation

The general view is that selection of states ought to be done automatically. This is also possible and unproblematic in most models, and we thus clearly understand that manual state selection can easily be overused. However, there are several reasons for allowing model library developers as well as users to influence or control the state selection:

- **Accuracy:** There are often many sets of state variables that will work from a pure mathematical point of view. However, they may have drastically different numerical properties. For mechanical systems it is favourable to use relative positions as state variables. If absolute coordinates are used then accuracy is lost when taking differences to calculate relative positions. The effect is drastic in rotating machinery systems and power systems where angular positions are increasing with time, but relative positions are rather constant, at least in normal operation. Say that two rotating bodies are connected by a spring such that the relative distance between them is 1 and that their angular speed is 1000. If the positions are calculated with a relative accuracy of 0.001, after one second there is hardly any accuracy in calculating the distance by taking the difference. The difference behaves irregularly and gives an irregular torque. The simulation stops. It is very difficult for a tool to find this out without actually doing simulation runs. Model developers for mechanical systems and power systems know it very well. It would be easy for them to indicate that absolute positions are bad choices when selecting states.
- **Efficiency by avoiding inverting functions**: The relations between possible sets of state variables may be non-linear. For some choices it may be necessary to invert non-linear functions, while for another set it is straightforward to calculate others. A typical example is thermodynamic problems, where you have property functions. They often assume two variables to be inputs (for example pressure and enthalpy) and calculate other properties (such as temperature, density etc). Thus, if such variables are selected as state variables it is “simply” calling property functions to calculate other need variables. If not it is necessary to solve equation systems to calculate the input variables. A model library developer knows this and it is straightforward to him to indicate good choices when selecting dynamic states.
- **Selecting a less nonlinear representation**: Different sets, x, of states gives an ODE, der(x) = f(x) where the right hand side f have different properties. In general, the problem is simpler to solve if f is a less nonlinear problem. The Park transformation for three-phase power systems is a classical way of transforming a nonlinear time-varying ODE into a time-invariant linear ODE. For control design it is very favourable to have linear time-invariant models, because there are lot of analysis and design methods and tools for such

models. When using linearized versions of Modelica models it is important that the set of state variables is insensitive to minor changes in the model.

- **Avoiding dynamic state selection**: When selecting states the problem consists of a set of algebraic state constraints that relate dynamic variables. It may be remarked that these constraints are equations that are differentiated by Pantelides's algorithm. The task when selecting states is actually to use the algebraic constraints to solve for some of the variables, which thus are deselected as states and the remaining dynamic variables become state variables. A subset of dynamic variables can be deselected locally if its Jacobian is non-singular. In the general case the state selection must be made dynamic, but in many real applications it is possible to make a static selection of states. If the Jacobian has constant elements it is straightforward to make it automatically. However, for non-linear problems such as closed kinematics loops it is difficult to establish that a time-varying Jacobian always is non-singular. For reasons of efficiency it would be favourable to avoid the overhead of dynamic state selection and allow a user to inform that a certain selection of states will always work. Tools can support such an explicit control. Using dynamic state selection and making off-line simulations one can find a fixed choice that will work for real-time simulation, where efficiency is really needed. Note: models with dynamic state selection cannot be used in real-time simulation.
- **The reinit construct:** The construct reinit(x) requires that x is state.
- **Use auxiliary variables as states:** To avoid unnecessary differentiation, it is useful to consider only variables appearing differentiated in a model as candidates when selecting states. It means that if a user would like to see an auxiliary variable, v, as a state variable, he has today to introduce another variable, say derv and an equation derv = der(v) to make the derivative der(v) appear in the model. It would be convenient to have a simpler way to introduce a variable as a state candidate.
- **Sensors:** A sensor for measuring speed, v, makes a variable differentiated, v = der(r) and in most cases it is not desirable to have the variable of the sensor model as a state variable. Introduction of variable for just plotting should not influence the state selection.

13.4.2 The state select attribute

A variable being subtype of Real variable has an attribute stateSelect to indicate its possible use as state variable. Its value can be:

never	Do not use as a state at all.
avoid	Avoid it as state in favour of those having the *default value.*
default	If the variable does not appear differentiated in the model this means *no.*
prefer	Prefer it as state over those having the *default value.*
always	Do use it as a state.

The values of the stateSelect attribute are to be given as

```
Real y(stateSelect = StateSelect.never);
Real y(stateSelect = StateSelect.avoid);
Real y(stateSelect = StateSelect.default);
Real y(stateSelect = StateSelect.prefer);
Real y(stateSelect = StateSelect.always);
```

The two extreme values *never* and *always* have clear and context independent meanings. If stateSelect is *always*, the variable will be a state. If such a variable does not appear differentiated in the model, the index reduction procedure will differentiate equations in order to be able to calculate the derivative. A model with two variables, x and y, with attribute *stateSelect* being *always* and being algebraically constrained, is thus erroneous. It is compulsory for variables appearing as arguments in reinit expressions. It supports explicit control of the selection of states and gives the user full control. It eliminates use of dynamic state selection. A dynamic state selection problem should only include variables having *stateSelect* being *prefer, default* or *avoid.*

The value *never* forbids the variable to be used as a state and it solves the sensor problem:

```
Real r(stateSelect = StateStateSelect.never);
Real v = der(r);
```

The value *prefer* indicates that the variable should be used as a state when possible. The ambiguity lies in that there may be several candidates with prefer when selecting states. It solves the problem of giving preference to relative positions in mechanical problems. It is also useful for thermodynamic problems to avoid nonlinear equation systems. However, here the value *never* may be useful to rule out other candidates as well.

The value *default* means *never* for algebraic variables of the model. The index reduction procedure may introduce derivatives of algebraic variables when differentiating equations. However, this should not make them candidates for being state variables. Neither should higher order derivatives make derivatives candidates for being state variables. For example in mechanics we have

```
der(r) = v;
m*der(v) = F;
```

The index procedure may introduce the second order derivative of r, but we should then not consider der(r) as candidate for being state variable.

The priorities for state selection are thus *always*, *prefer*, *default and avoid* for variables appearing differentiated.

13.4.3 Avoiding dynamic state selection when having only states that can be integrated separately from other states

If your model contain “trivial states”, that is, states that could be integrated separately of other states, it may be possible to avoid dynamic state selection by setting the flag

```
Advanced.PreferSelectingTrivialStates=true
```

(The flag is false by default.)

In some cases a filtered trivial state is part of the dynamic state selection due to the stateSelect-attribute and the details of the model. Such a state is only selected as trivial if this selection avoids the dynamic state selection, and avoids having to modify the stateSelect-attribute to force this choice.

Note. Using this flag may not mean that the simulation is faster.

13.5 Using noEvent

Note that this is an advanced section, and in most cases one should not use noEvent. This section describes the exceptions, and how to correctly use noEvent.

13.5.1 Background: How events are generated

By default Dymola generates events for the relational operators (>,>=,<,<=) and certain built-in functions: ceil, floor, div, mod, rem and integer. A simple optimization ensures that events are only generated if the arguments are varying continuously. Events are generated *after* the Boolean expression have changed value, and it is thus necessary that expressions involving relations are valid and smooth a certain amount past the actual event.

Events are generated for code for equations and algorithms outside of functions. For algorithms there are currently some minor limitations for events in for-loops and severe limitations inside while-loops.

The problem of using events for *all* relations is that one cannot use an expression to guard against errors, e.g. square root of a negative number, since the Boolean guard would keep the value from the previous event. Furthermore, the events can lead to undesirable degrading of performance, if the derivatives are sufficiently smooth. On the other hand, if the derivatives are not smooth removing events would degrade performance even more.

13.5.2 Guarding expressions against evaluation

Certain numerical operations have a limited range of allowed input values, e.g. one cannot divide by zero and one cannot take the square root of a negative number. To guard against this one must use noEvent surrounding the guard condition.

As an example consider guarding against taking the square root of a negative number. An idealized model of a tank that is emptied through a hole in the bottom is:

$$\dot{h} = \begin{cases} -c\sqrt{h}, & \text{if } h \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

The model is best written as

```
  Real h "Height of water in tank";
equation
  der(h)=if noEvent(h>=0) then -c*sqrt(h) else 0;
```

As an alternative we could in this particular case rewrite it using the max function resulting in equations that are more compact, but less readable.

```
  Real h "Height of water in tank";
equation
  der(h)=-c*sqrt(max({0,h}));
```

Note that we guard against h being negative even if the exact solution to the differential equation has h(t)>=0. The reason is that the numerical solution generates an approximate h(t) which will be slightly negative. In general one must not only guard against the possible values for the exact solution, but also for all nearby approximate values.

Another example concern guarding against division by zero:

```
  Real x;
  Real sinc_x;
equation
  sinc_x=
   if noEvent(abs(x)<=Modelica.Constants.eps) then
     1
   else
     sin(x)/x;
```

In these two examples the noEvent is necessary in order to make it possible to evaluate the model for all values. Note that in the second example we do not compare abs(x) with 0 but with the slightly larger Modelica.Constants.eps. This provides a guard against roundoff errors. The extra error is insignificant since the error in the sinc approximation it is proportional to the square of x, and thus lost in round-off.

13.5.3 How to use noEvent to improve performance

In some cases the derivatives are sufficiently smooth that events need not be generated, as an example we have piece-wise interpolation polynomials.

```
  parameter Real cpos,cneg;
  Real x,y;
equation
  der(x)=noEvent(if x>0 then cpos*x else cneg*x)
      +if time>=1 then y else 0;
```

For this to be useful the right hand side must be a sufficiently smooth function of x. In this case we note that the switch between the branches occurs for x=0 in which case both branches are zero. Note that as for all performance optimizations one should measure the performance to verify that the optimization is beneficial.

Additionally noEvent only surrounds the expression involving x and not the second term, which will generate a time event exactly at the time 1.

A note on style

When using the noEvent operator to improve performance we are implicitly stating that the expression is sufficiently ‘smooth’. Dynasim initiated work in the Modelica design process to introduce an operator ‘smooth’ for this purpose, and this thinking explains why noEvent is used surrounding the entire expression in the examples above and not only the actual

relational operator. When viewing it as smooth it only makes sense to view a real-valued expression as smooth. This also guards against accidentally introducing events in one of the sub-expressions. The smooth-operator is now available and should be used in most cases.

13.5.4 Combined example for noEvent

A more subtle example is if the noEvent is necessary in order to guarantee that we can solve for all algebraic variables. This is more complex than merely being able to evaluate the model. A simple example occurs for the turbulent flow equation

$$\Delta P_{loss} = c\dot{m}|\dot{m}|$$

```
  Real mdot,Ploss;
  parameter Real c;
equation
  Ploss = c* mdot*noEvent(abs(mdot));
```

Remember that abs(mdot) is expanded into an if-expression, in this case leading to:

```
  Ploss = c* mdot*noEvent(if mdot>=0 then mdot else -mdot);
```

The noEvent allows us to use this equation to implicitly solve mdot from Ploss. Without the noEvent it would not be possible to solve the equation past the point where mdot changes sign (since it would be tantamount to taking the square root of a negative number).

We can also manually solve this equation for Ploss resulting in

```
  Real mdot,Ploss;
  parameter Real c;
equation
  mdot = noEvent(if Ploss/c>=0 then sqrt(Ploss/c) else -sqrt(-
Ploss/c));
// Or: mdot = noEvent(sign(Ploss/c)*sqrt(abs(Ploss/c)));
```

We have here not considered the possibility that c might be zero.

The right hand side is here continuous when Ploss passes through zero, and thus noEvent can be seen as a performance improvement. Additionally we guard against taking the square of a negative number by using noEvent.

An additional note on style

In this example we note an additional reason for using noEvent around the entire expression: if one of the branches of the if-expression contain any relations these relations should also be inside noEvent.

Consider the out-commented part of the previous example where we use abs and sign for Ploss/c. When using that form it is easy to forget that not only abs but also sign generates events and only have noEvent surrounding sqrt. Although it does generate errors, we unnecessarily lose performance due to the sign events. By having noEvent around the entire expression this is automatically handled.

13.5.5 Constructing anti-symmetric expressions

Quite a few expressions are naturally anti-symmetric in some variable. In most cases this requires no extra thought, and only requires one to write the expression in the natural form, and it will be anti-symmetric.

However, in some cases it is known that a formula is anti-symmetric, and it straightforward to give a formula for positive values, but more difficult to give a simple formula valid for both positive and negative values. The natural idea in these cases is to extend the expression for positive values to an anti-symmetric formula valid for all values.

As a generic example consider

$$\dot{x} = \begin{cases} g(x,p), & x \geq 0 \\ -g(-x,p), & x \leq 0 \end{cases}$$

As a model this is written as:

```
  Real x;
  parameter Real p;
equation
  der(x)=noEvent(if x>=0 then g(x,p) else -g(-x,p));
  assert(noEvent(abs(g(0,p))<=1e-10),
        "Formula requires g(0,p)=0");
```

In most cases the assert-statement would be removed for efficiency reasons, and the function g replaced by an expression involving x, p, and perhaps other variables.

There are several details worth explaining in this example.

First and foremost neither abs nor sign are used; the reason is that by having only one test for the sign of x we guarantee that Dymola can correctly differentiate the expression (provided g(0,p)=0), and use it to reduce the index and to compute Jacobians for non-linear system of equations involving this equation.

Second noEvent is used. The reason is that we assumed that g(x,p) was not valid for negative x, either because it cannot be evaluated or because it generated incorrect results. Thus using noEvent guarantees that g(x,p) is only evaluated for positive values of x.

Thirdly, the formula is only valid if g(0,p)=0. There are anti-symmetric expressions that do not obey this, such as friction force depending on relative velocity. In those cases one should introduce an extra locked state, as described in the friction logic models, and under no circumstances use noEvent. Using the above formula without thinking would for friction lead to a sliding mode, and many unnecessary events.

Although one has to be careful one can construct a variant of this model, where we introduce an auxiliary variable for the sign. In this form it is also possible to have expressions generating events in the expanded form of g(x,p), provided they do not depend on x.

```
  Real x,sign_x;
  parameter Real p;
equation
  sign_x=noEvent(if x>=0 then 1 else -1);
  der(x)=sign_x*g(sign_x*x,p);
```

```
assert(noEvent(abs(g(0,p))<=1e-10),
       "Formula requires g(0,p)=0");
```

Similar remarks as for the first example apply to this example.

13.5.6 Mixing noEvent and events in one equation

There are no restrictions on using variables computed using noEvent to trigger events in other equations, or to applying noEvent to expressions involving variables computed using events. This allows a modeller to combine models independently of whether they internally use events or have turned them off using noEvent.

Sometimes it makes sense to mix noEvent and events in *one* equation. This is, however, only allowed under certain restrictions since the number of active relations generating events can only change at events. We would otherwise not be able to determine if the expression had changed value or not and thus would be unable to correctly determine when to trigger the event.

Thus if the condition of an if-expression, if-clause, while-clause, or the indices of a for-loop is a relation where events have been turned off by using noEvent the bodies should not contain relations generating events. They can, of course, depend on variables that are computed using events. As a concrete example consider the following:

$$\dot{x} = \begin{cases} -1, & \text{if } x \geq 1 \\ -x, & \text{if } x \geq 0 \wedge x \leq 1 \\ 2x, & \text{if } x \leq 0 \end{cases}$$

Assuming that we do not want events when x passes through 1 an illegal example would be:

```
model ILLEGAL
  Real x(start=2);
equation
  der(x)=
    if noEvent(x<1) then
       if x>0 then -x else 2*x
    else -1;
end ILLEGAL;
```

In order to explain why this is illegal consider what happens after one second when x becomes less than one and x>0 is evaluated for the first time (without triggering an event because of the noEvent). The next if-expression should trigger an event if x>0 does not have the value from the last event. However, we did not evaluate it at the previous event and thus we do not know if it has changed value or not and thus we do not know when to trigger an event.

We are not allowed to evaluate the relations inside the wrong branch of the if-expression since that might involve unsafe operations, such as square roots of negative number, indexing outside of bounds, etc.

To solve this problem we can either surround the entire right-hand side by noEvent or introduce an auxiliary variable for '`if x>0 then -x else 2*x`'.

Conditional use of events

An extreme example of mixing noEvent and events is to have one variable control whether events should be generated or not. This is only applicable if noEvent is introduced for performance reasons.

We revisit the interpolation example from "How to use noEvent to improve performance" on page 1406, and let a Boolean parameter control whether we generate events or not.

```
  Real x;
  parameter Boolean generateEvents;
  parameter Real cpos,cneg;
equation
  der(x)=
    (if (if generateEvents then x>0 else noEvent(x>0))
       then cpos*x
       else cneg*x)
    +if time>=1 then y else 0;
```

If generateEvents is not a boolean parameter, but changes continuously we have to be more careful.

```
  Real x;
  Real level;
  parameter Real eventLimit;
  parameter Real cpos,cneg;
equation
  level=noEvent(abs(der(x-time))*2+abs(x-time));
  der(x)=
    (if (if level>eventLimit then x>0 else noEvent(x>0))
       then cpos*x
       else cneg*x)
    +if time>=1 then y else 0;
```

Here level must be computed using noEvent since we do not want to introduce extra events every time der(x) or x changes sign, but level is used without any noEvent in the if-expression since the number of relations generating events would otherwise change between events.

It is not possible to store the expression in parenthesis in the if-expression in a Boolean variable, since it can change its value between events. To re-use it introduce an extra Real variable as follows.

```
  Real x;
  Real level;
  parameter Real eventLimit;
  Real xIsPositive;
  parameter Real cpos,cneg;
equation
  level=noEvent(abs(der(x-time))*2+abs(x-time));
  xIsPositive=if
   (if level>eventLimit then x>0 else noEvent(x>0)) then
     1
   else
    -1;
```

```
der(x)=(if noEvent(xIsPositive>0) then cpos*x else cneg*x)
 +if time>=1 then y else 0;
```

Note the noEvent on the last line. Without it we would always have an event when x changed sign. If level>eventLimit the expression x>0 will introduce events and thus xIsPositive need not introduce an additional event.

13.6 Equality comparison of real values

Following the Modelica specification Dymola does not allow you to compare two real values for equality. The reason is not that it would be difficult to allow it, but that the desired result depends on circumstances and there is not *one* correct way of re-writing it as legal code. Instead of automatically generating a result that would only work for some cases, you are required to manually select the desired result.

13.6.1 Type of variables

In many cases it does not matter whether variables are declared as Integer or Real. However, only integers may be used as indices and compared for equality. Thus some equality comparisons between real expressions can be removed by replacing Real variables by Integer variables (and perhaps a fixed scaling).

In some cases all variables and constants appearing in the expression are integers, but some operations generate a real valued result, e.g. division. If the result is known to be an integer one can replace these by integer division (div) or use integer to convert a real-valued expression to an integer-valued one.

13.6.2 Trigger events for equality

In some cases one want to perform some special action triggered when two real expressions have the same value. This is not possible, but it is for continuous varying variables equivalent to triggering the condition as follows, which is a part of car model where we want to terminate the simulation when the velocity is equal to 100km/h.

```
  Modelica.SIunits.Velocity velocity;
  constant Modelica.SIunits.Velocity stopAt=100/3.6;
equation
  when {velocity>=stopAt,velocity<=stopAt} then
    if not initial() then
      terminate("Velocity is 100km/h");
    end if;
  end when;
```

Assuming this is a normal test of accelerating to 100km/h it is possible to remove the second triggering condition velocity<=stopAt, because we know that the original velocity is less than 100km/h. This also allows us to remove if-statement. Similar reasoning applies to many similar cases.

13.6.3 Locking when equal

In some models, e.g. bouncing balls it is natural to enter another state when the relative velocity is zero in order to avoid chattering. By necessity this should not occur when the relative velocity is exactly zero, but when it is small enough. The first challenge is thus to guarantee that the ball stops bouncing in this case, the second is that if we apply a relative force it should start bouncing anew. A model demonstrating this is given below.

```
model BouncingBall
  import Modelica.SIunits;
  SIunits.Height x(start=1);
  SIunits.Velocity v;
  SIunits.Force f;
  parameter SIunits.Mass m=2;
  parameter Real ebounce=0.5;
  parameter SIunits.Velocity vsmall=1e-4;
  Real fext=if time<10 then 0 else 350*sin(time)/(1+time);
  Boolean locked(start=false);
equation
  der(x)=v;
  m*der(v)=if locked then 0 else f;
  f=-m*Modelica.Constants.g_n+fext;
  when {x<=0,locked} then
    reinit(x,0);
    reinit(v,if locked then 0 else -ebounce*v);
  end when;
  locked=if pre(locked) then f<=0 else x<=0 and
    abs(v)<=vsmall and f<=0;
end BouncingBall;
```

Note that the logic here is more complex due to the fact that we have an external force in addition to the inherent bouncing of the ball. When writing such state machine logic it is vital that the expression used to enter the locked state is still true in the state (in order to avoid chattering), and to be safer one can rewrite it as

```
locked = f<=0 and if pre(locked) then true else
  x<=0 and abs(v)<=vsmall;
```

A more advanced model would have these as relative quantities and allow both the ball and the surface to move.

13.6.4 Guarding against division by zero

The sole exception where comparison of two real expressions would make sense is when it is used to guard against a single exceptional value, e.g. division by zero. By careful analysis it is in general possible to show that the guard can and should be applied to slightly larger values, and use the technique in “Guarding expressions against evaluation” on page 1405.

In some cases only one exact value is exceptional, and it is not possible to apply the guard for other values. In those rare cases one can use the same technique without any extra epsilon, i.e. ‘noEvent(abs(x)<=0)’.

13.7 Some supported features of the Modelica language

Dymola's support for the Modelica language is extensive. Some of the supported features are dealt with below.

13.7.1 Support for Modelica Language version 3.5

Dymola 2023 is compliant with the Modelica Language Specification version 3.5.The most important supported new features are:

- Public variables in functions must be input or output
- Hierarchical Evaluate
- Final for classes

Examples of other supported features are:

- Impure functions as function arguments
- Introduce ModelicaDuplicateString
- Element for convertElement
- Add SourceDirectory
- Test-case annotations for invalid models
- Dialog-annotations for classes
- Svg-bitmaps
- Unit for %par
- Deprecate byte-order mark
- Assertion-level should be a structural parameter

For more information, see https://www.modelica.org/modelicalanguage. The specification is also included in the Dymola distribution, click here.

13.7.2 GUI support for removing any existing modification of a parameter by using the keyword break

The keyword **break** for removing existing modifications of a parameter is supported. The keyword is a proposed extension of the Modelica language (MCP-0009).

The basic idea is that you can write `x=break` to remove all modifications of a parameters `x`. Often you want to remove unsuitable start values as well, which means that for the resistance parameter R in a resistor R1 you may need to write, as an example:

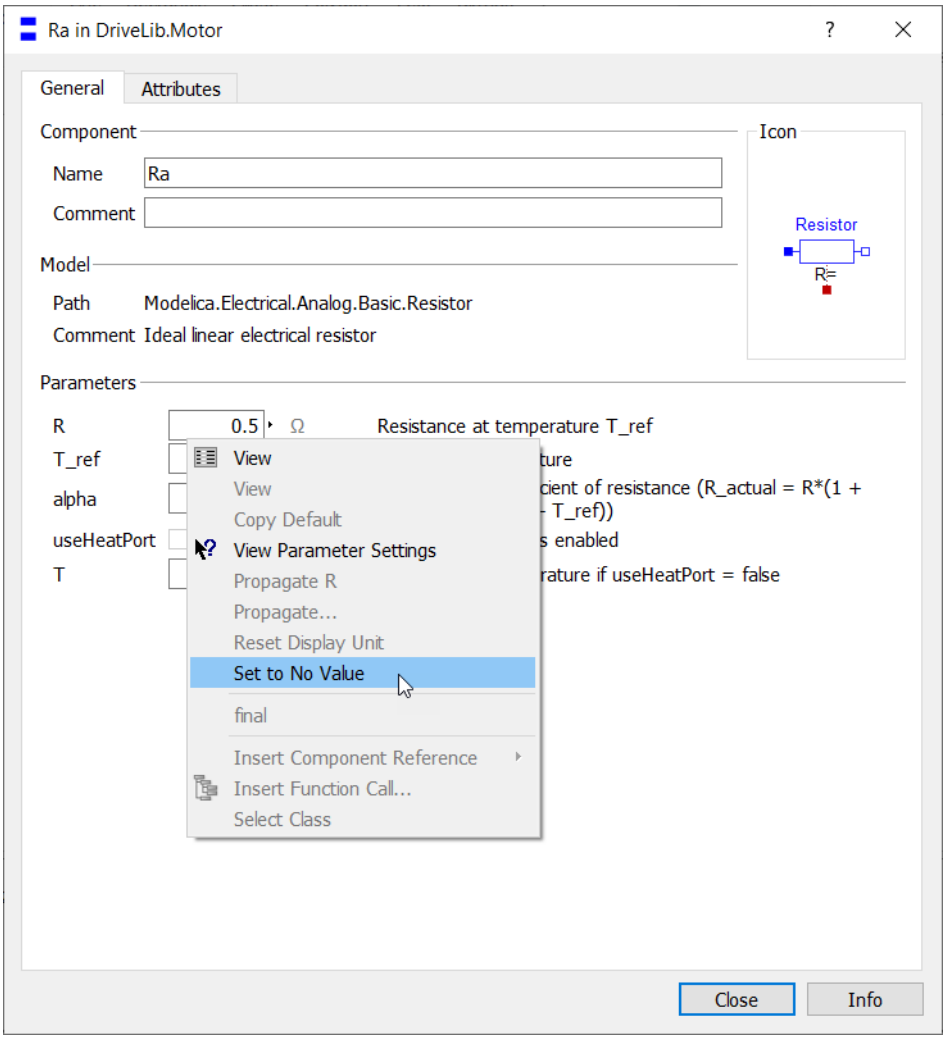
```
Resistor R1(R(start=break)=break);
```

To make it easier to create such constructions, you can right-click a parameter in the parameter dialog and select **Set to No Value**.

Important! The flag `Advanced.Modelica.SupportUndefinedModification` must be set to true to be able to use this command. (The flag is by default `false`.)

To test this:

- Open the demo Motor Drives by the command **File > Demos > Motor Drives**.
- In the package browser, right-click DriveLib and select **New > Duplicate Class…** and click **OK** to create an editable copy of the library.
- Open the Motor by double-clicking it in the package browser.
- Right-click the resistor Ra and select **Parameters** to display the parameter dialog.
- Right-click the value box after R in the Parameters section. You can now select **Set to No Value**. (Don´t forget that a flag has to be set, see above.)



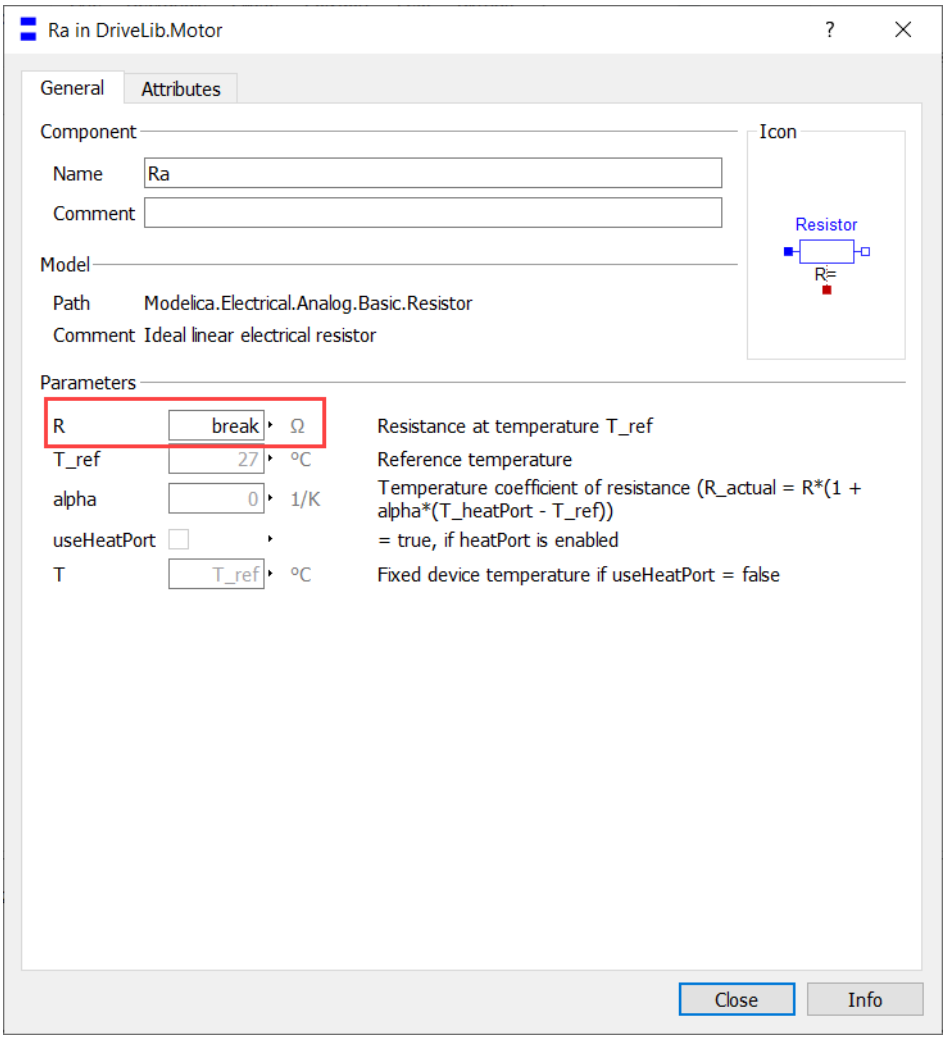
The Modelica Text before applying the command is:

```
model Motor "A basic model of an electrical dc motor."

  Modelica.Electrical.Analog.Sources.SignalVoltage Vs ;
  Modelica.Electrical.Analog.Basic.Ground G ;
  Modelica.Electrical.Analog.Basic.Resistor Ra(R=0.5) ;
  Modelica.Electrical.Analog.Basic.Inductor La(L=0.05, i(fixed=true))
                                                      ;
  Modelica.Electrical.Analog.Basic.RotationalEMF emf(k=1) ;
  Modelica.Blocks.Interfaces.RealInput inPort         ;
  Modelica.Mechanics.Rotational.Components.Inertia Jm(
                                          J=0.001) ;
  Modelica.Mechanics.Rotational.Interfaces.Flange_b flange_b ;
equation

  
end Motor;
```

Applying the command **Set to No Value** gives:



And the corresponding Modelica Text:

```
model Motor "A basic model of an electrical dc motor."

  Modelica.Electrical.Analog.Sources.SignalVoltage Vs ;
  Modelica.Electrical.Analog.Basic.Ground G ;
  Modelica.Electrical.Analog.Basic.Resistor Ra(R(start=break) = break)
                                                    ;
  Modelica.Electrical.Analog.Basic.Inductor La(L=0.05, i(fixed=true))
                                                    ;
  Modelica.Electrical.Analog.Basic.RotationalEMF emf(k=1) ;
  Modelica.Blocks.Interfaces.RealInput inPort          ;
  Modelica.Mechanics.Rotational.Components.Inertia Jm(
                                          J=0.001) ;
  Modelica.Mechanics.Rotational.Interfaces.Flange_b flange_b ;
equation

  
end Motor;
```

13.7.3 Support for Modelica Language version 3.4

Dymola is compliant with the Modelica Language Specification version 3.4. Support for the following features was added in Dymola 2018:

- Explicitly casting a model record, provided you set the flag `Advanced.RecordModelConstructor=true;` (the default value of the flag is `false`).
- Differentiation of functions handles records mixing Real and non-Reals.
- Conversion from Integer to enumeration, e.g. Modelica.Blocks.Types.Smoothness(2) gives Modelica.Blocks.Types.Smoothness.ContinuousDerivative.
- Ellipse segments can set whether to also draw arc, chord, or nothing.

Note that the other major features added in Modelica 3.4 were already supported in previous releases of Dymola – e.g. automatic conversion of models.

13.7.4 Synchronous Modelica

The synchronous features of Modelica are supported. References to papers describing these features are available using the command **Tools > Help Documents**. A tutorial is also available using the command **File > Demos > Modelica Synchronous Tutorial**. The tutorial relates to the papers and presentations mentioned.

13.7.5 State Machines

The State Machines of Modelica are supported. References to papers describing these features are available using the command **Tools > Help Documents**. A tutorial is also available using the command **File > Demos > Modelica Synchronous Tutorial**. The tutorial relates to the papers and presentations mentioned.

13.7.6 Operator overloading

Dymola supports operator overloading.

- The special class `operator record` is supported. Overloaded operators can only be defined inside such a class.
- This allows user-defined specification of operations such as +, -, /, *.
- Overloaded element "0" is supported. This element enables operator record classes to be used as flow variables in connectors.
- Inheritance of `operator record` is allowed if defined via a short class definition.

13.7.7 Homotopy operator

The Modelica homotopy operator `homotopy` is supported.

The operator is a powerful tool in improving the convergence of iterative solvers by providing an alternative, simplified version of the model that is not so dependent of accurate initial guess values for the unknown variables, and then continuously transforming that simplified model to the original more complicated model.

In other words, the operator allows the user to make it easier to solve the initialization problem by formulating a simpler initialization problem. This is handled by symbolically processing the system of equations, resulting in e.g. replacing non-linear flow-characteristics by a linear approximation valid at the operating point. The simpler initialization problem is first solved, and then continuously changed to finally solve the actual initialization problem.

The step-size is chosen adaptively allowing the initialization to converge for more complicated problems.

The homotopy method is activated automatically if an initialization problem cannot be solved and the homotopy operator is used.

To debug a failing homotopy process, you can use the flag `Advanced.DebugHomotopy`. By setting this flag to true (the default is false), a file continuation.csv is created during simulation. It contains the values of the iteration variables at the different stages during the transformation from the simplified problem to the actual problem.

In previous versions, the file was only created during homotopy recursions where the homotopy operator is used. From Dymola 2019 FD01 the file is also created during homotopy recursions where the homotopy operator is not used, during both initialization and even iteration

For more information about homotopy, please see the papers:

- Robust Initialization of Differential-Algebraic Equations Using Homotopy.
- Steady-state initialization of object-oriented thermo-fluid models by homotopy methods.

These papers are available at www.Modelica.org. **Note** that the first paper includes application examples from different physical domains.

Note. It is possible to debug a failed homotopy process. This is not included in the papers above.

To debug a failed homotopy process, set the flag

```
Advanced.DebugHomotopy = true
```

(the flag is by default false). With this flag enabled, a file `continuation.csv` is created during simulation. It contains the values of the iteration variables at the different stages during the transformation from the simplified problem to the actual problem.

13.7.8 Arrays

The support for array of records makes it possible to check the entire Modelica library.

- The size of arrays of records can depend on the inputs.
- Arrays in functions declared using the size : can be resized by assigning to the entire array. (This applies to both arrays of records and arrays of simple types.):

```
function f
  input Integer n;
  output Real x[:];
algorithm
   for j in 1:n loop
      x:=cat(1, x, {j});
   end for;
end f;
```

Array of records (with literal size) is supported in compiled functions, and in all cases for non-compiled functions.

Flexible array sizes and resizing of arrays in functions

Dymola fully supports functions with variable sized arrays (declared using [:]) in compiled functions, described in section 12.4.5 in Modelica Language Specification, Version 3.4.

Note that changing the size of such arrays uses more memory and time.

For efficiency a hint is to first assign zeros(maxSize) to the array, assign elements using an auxiliary size-indicator, and then shrink the array at the end to the auxiliary size-indicator.

As example, consider the following function:

```
function
…
protected Real x[:];
algorithm
   for i in 1:n
     if (…) then
       x:=cat(1,x,{f(i)});
     end if;
   end ..;
```

This function can be rewritten, to be more efficient, as (changes in italics):

```
function
…
protected Real x[:];
Integer x_size;
algorithm
   x:=zeros(n);
   x_size:=0;
```

```
for i in 1:n
  if (…) then
    x_size:=x_size+1;
    x[x_size]:=f(i);
  end if;
end for ..;
x:=x[1:x_size];
```

Note that dynamic sizing of arrays in models is not supported.

13.7.9 Enumerations

General

Enumerations are supported. You can:

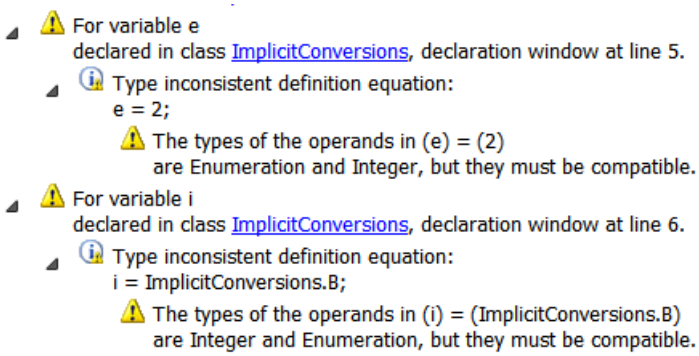
- Declare enumeration types – with description for each member. The enumeration types will automatically get proper choices in the parameter dialog (using the description).
- Declare variables of enumeration types (the min and max attribute is automatically set to first and last enumeration members).
- Have algorithms, equations and bindings for enumerations.
- Use enumeration literals.
- Declare arrays indexed by enumeration types.
- Convert enumeration values to strings.
- Convert from Integer to enumeration, e.g. Modelica.Blocks.Types.Smoothness(2) gives Modelica.Blocks.Types.Smoothness.ContinuousDerivative.

Enumeration value conversion

Implicit conversion of integers to enumeration values and vice-versa are unsafe. For example, the following model will result in translation warnings:

```
model ImplicitConversions
type E = enumeration(
A,
B);
output E e = 2;
output Integer i = E.B
end ImplicitConversions;
```

The warnings:



To safely convert an enumeration value to an integer or vice-versa, the integer or respective enumeration type constructors can be used:

```
model ExplicitConversions
type E = enumeration(
A,
B);
output E e = E(2);
output Integer i = Integer(E.B)
end ExplicitConversions;
```

The usage of enumeration constructors to avoid warnings is highly recommended. In future Dymola versions implicit integer to enumeration values might be errors, not just warnings.

13.7.10 Support of String variables in models

String variables can be used in models, not only in functions used in models and for scripting. String variables in models are allocated with a maximum string length. It is defined by the scripting variable `Advanced.MaxStringLength` with default=500. If assignment to a model string variable fails due to too short string length, truncation is done and a warning is given in the simulation log window.

13.7.11 Support of inner/outer components

In order to support inner/outer components some new annotations are supported (can also be useful for other cases):

```
annotation (
  defaultComponentName="world",
  defaultComponentPrefixes="inner replaceable",
  missingInnerMessage="No \"world\" component is defined. A
default world
component with the default gravity field will be used
(g=9.81 in negative y-axis). If this is not desired,
drag MultiBody.World into the top level of your model.",
```

13.7.12 Functions as formal input to functions

Functional input arguments to functions in order to e.g. pass criteria functions to generic optimization functions are supported. This is useful for e.g. Design Optimization.

The following functionality is currently supported:

- Sending a function as an input argument to another function, e.g. `function sin()` as argument.
- Calls through a functional input argument, e.g. `integrand(x)`.
- Propagating a functional input argument, either directly, e.g. `function integrand(x)`, or with partial binding, e.g. `function integrand(x=1)`.
- Partial interface functions.
- Functional input arguments to functions can be used interactive or in models.

13.7.13 Assert

In Modelica, an assertion that in short looks the following (for more information, see the Modelica Language Specification version 3.4, section 8.3.7):

```
assert(condition, message, level=AssertionLevel.error);
```

where `condition` is a Boolean expression, `message` is a string expression and `level` is a built-in enumeration with a default value. It can be used in equations or algorithms.

If the `condition` is true, message is not evaluated and the procedure call is ignored. If `condition` evaluates to false, different actions is taken depending on the level input:

- `Level=AssertionLevel.error`. The current evaluation is aborted; message indicates the cause of the error.
- `Level=AssertionLevel.warning`: The current evaluation is not aborted; message indicates the cause of the warning.

13.7.14 Identifiers starting with underscore and vendor-specific annotations

In general identifiers can start with underscore (although not recommended if not vendor-specific annotations).

Modelica allows vendor-specific annotations. All Dymola specific annotations can be preceded by `__Dymola_` as specified in the Modelica language specification. The next sections are examples of this.

13.7.15 Quoted identifiers containing dot supported

Quoted identifiers containing dot are now supported, including

- Class and component names
- Hierarchical modifiers

- Variable browser
- Import
- Derivative names

It is recommended to avoid quoted identifiers (with or without dot) for top-level classes; the file names will not be nice.

13.7.16 Running a function before check/translation/simulation

A model may specify that a function should be run before a component of the model is checked, translated or simulated. This could be used to verify structural parameters or external data before using them. The function to be called is specified in an annotation in the following form:

```
model M
  parameter String s="";
  annotation(__Dymola_preInstantiate=MyPackage.foo(s));
end M;
```

Notes:

- This is only intended as a help and there is no absolute guarantee that this function will be called. It is not intended for e.g. license-checking.
- Any parameters used must be possible to evaluate to literals.
- Any parameters used must be top-level parameters in the model, e.g. you cannot use `component.s`.
- Conditional components first have their condition evaluated. If the condition is false the component is not present and the `__Dymola_preInstantiate` function is not called.

13.7.17 Forcing translation of functions

By applying the annotation `annotation(__Dymola_translate=true)` to a function, a translation for that function is forced. This speeds up the translation. The annotation can be applied to both external and non-external functions.

13.7.18 Support for predefined plots

Dymola supports **Annotations for Predefined Plots**, which are proposed as an extension of the Modelica Specification (MCP-0033). For details, see index entry *plot : predefined plots* in this manual.

13.7.19 Deprecation warnings

In Dymola the following language constructs generate warnings indicating future deprecation:

- The usage of `extends` to constrain replaceable components; `constrainedby` should be used instead.
- The usage of `operator` as an identifier name; since Modelica 3, operator is a keyword.
- The usage of the `flow` and `stream` prefix within short class definitions.

The warnings are displayed in the messages window, in the **Syntax Error** tab.

13.7.20 Licensing

Licensing of packages is supported. Please see, in the chapter "Model Management", section "Encryption in Dymola", sub-section "Licensing Libraries" for more information.

13.8 Symbolic Processing of Modelica Models

(This section, except "Handling of implicit constraints between dynamic variables", "Alias elimination of parameters", and "Efficient code generation for large tables" is a reprint from Handbook of Dynamic System Modeling 2007, chapter 36.4, by M. Otter, H. Elmqvist and S. E. Mattson - courtesy of Taylor & Francis Group LCC – Books.)

The Modelica Language Specification (Modelica 2005) defines how a Modelica model shall be mapped into a mathematical description as a mixed system of *differential-algebraic equations* (DAE) and *discrete equations* with Real, Integer and Boolean variables as unknowns. There are no general-purpose solvers for such problems. There are numerical DAE solvers, which could be used to solve the continuous part. However, if a DAE solver is used directly to solve the original model equations, the simulation will be very slow and initialization might be not possible for higher index systems (see below). It is therefore assumed that Modelica models are first symbolically transformed into a form that is better suited for numerical solvers. In this section, the transformation techniques are sketched that have been initially designed for the Dymola modeling language (Elmqvist 1978), further developed in Omsim for the Omola language (Mattsson and Söderlind 1993) and in the commercial Modelica simulation environment Dymola (Mattsson et. al. 2000, Dynasim 2006):

Dymola converts the differential-algebraic system of equations symbolically to ordinary differential equations in state-space form, i.e. solves for the derivatives. Efficient graph-theoretical algorithms are used to determine which variables to solve for in each equation and to find minimal systems of equations to be solved simultaneously (algebraic loops). The equations are then, if possible, solved symbolically or code for efficient numeric solution is generated. Discontinuous equations are properly handled by translation to state or time events as required by numerical integration routines.

13.8.1 Sorting and algebraic loops

The behavior of a Modelica model is defined in terms of genuine equations and a Modelica translator must assign an equation for each variable as part of the sorting procedure, which also identifies algebraic loops. To be able to process problems with hundred thousand

unknowns, the idea is to focus on the structural properties, i.e., which variables that appear in each equation rather than how they appear. This information can be represented by a "structure" Jacobian, where for a system of equations, $\mathbf{h}(\boldsymbol{x}) = \mathbf{0}$, each element i, j, is zero if x_j does not appear in the expression h_i, otherwise it is one. The sorting procedure is to order unknowns and equations to make the structure Jacobian become Block Lower Triangular, BLT. A BLT partitioning reveals the structure of a problem. It decomposes a problem into sub-problems, which can be solved in sequence. There are efficient algorithms, see, e.g., (Duff et.al. 1986), for constructing BLT partitions with diagonal blocks of minimum size (with respect to permutation of equations and variables). Each non-scalar block on the diagonal constitutes an algebraic loop. This sorting procedure identifies all algebraic loops in their minimal form that is unique. The sorting procedure is done in two steps. The first step is to assign each variable, x_j, to a unique equation, $h_i = 0$ such that x_j appears in this equation. It can be viewed as permuting the equations to make all diagonal elements of the structure Jacobian non-zero. If it is *impossible* to pair variables and equations in this way then the problem is structurally singular. The second step of the BLT partition procedure is to find the loops in a directed graph that has the variable/equation pairs of the first step as nodes. The basic algorithm was given by Tarjan (Tarjan 1972).

13.8.2 Reduction of size and complexity

A Modelica model has typically many simple equations, $v_1 = v_2$ or $v_1 = -v_2$ being the result of connections. These are easy to exploit for elimination.

From the BLT partition it is rather straightforward to find unknowns that actually are constant and can be calculated and substituted at translation. This may have considerable impact on the complexity of the problem that has to be solved numerically. For example, a multi-body component is developed for free motion in a 3-dimensional space. When using it we connect it to other components and set parameters implying restrictions on its motion. For example, it may be restricted to move in a plane. It means that coefficients in the equations become zero and terms disappear. This in turn may make algebraic loops to decompose into smaller loops or even disappear.

A linear small algebraic loop is solved symbolically. Otherwise code for efficient numeric solution is generated. In order to obtain efficient simulation, it is very important to reduce the size of the problem sent to a numerical solver. The work to solve a system of equations increases rapidly with the number of unknowns, because the number of operations is proportional to the cube of n, i.e. $O(n^3)$, where n is the number of unknowns. One approach to reduce size is called tearing (Elmqvist and Otter 1994). Let $\mathbf{z}$ represent the unknowns to be solved from the system of equations. Let $\mathbf{z}$ be partitioned as $\mathbf{z}_1$ and $\mathbf{z}_2$ such that

$$\begin{aligned}\mathbf{L} \cdot \mathbf{z}_1 &= \mathbf{f}_1(\mathbf{z}_2) \\ \mathbf{0} &= \mathbf{f}_2(\mathbf{z}_1, \mathbf{z}_2)\end{aligned}$$

where $\mathbf{L}$ is lower triangular with non-zero diagonal elements. A numerical solver needs then only consider $\mathbf{z}_2$ as unknown. A numerical solver provides guesses for $\mathbf{z}_2$ and would like to have the $\mathbf{f}_2$ residuals calculated for these guesses. When having a value for $\mathbf{z}_2$, it is simple to calculate $\mathbf{z}_1$ from the first set of equations. Note, that it is very important to avoid divisions by zero. The assumption that the diagonal elements are non-zero guarantees this. It is then straightforward to calculate the $\mathbf{f}_2$ residuals. The $\mathbf{z}_1$ variables are in fact hidden from the

numerical solver. The general idea of tearing is to decompose a problem into two sets, where it is easy to solve for the first set when the solution to the second set is known and to iterate over the second set. The aim is of course to make the number of components of $\mathbf{z}_2$ as small as possible. It is a hard (NP-complete) problem to find the minimum. However, there are fast heuristic approaches to find good partitions of $\mathbf{z}$. If the equations are linear, they can be written as

$$\mathbf{L}\mathbf{z}_1 = \mathbf{A}\mathbf{z}_2 + \mathbf{b}_1$$
$$\mathbf{0} = \mathbf{B}\mathbf{z}_1 + \mathbf{C}\mathbf{z}_2 + \mathbf{b}_2$$

and it is possible to eliminate $\mathbf{z}_1$ to get $\mathbf{J}\mathbf{z}_2 = \mathbf{b}$, where

$$\mathbf{J} = \mathbf{C} + \mathbf{B}\mathbf{L}^{-1}\mathbf{A}$$
$$\mathbf{b} = \mathbf{b}_2 + \mathbf{B}\mathbf{L}^{-1}\mathbf{b}_1$$

This may be interpreted as Gauss elimination of $\mathbf{z}_1$. The procedure may be iterated. Note, since $\mathbf{L}$ is a lower triangular matrix, the determination of $\mathbf{J}$ and $\mathbf{b}$ is at most $O(n^2)$.

When solving a linear equation system, a major effort is to calculate an LU or QR factorization of the Jacobian, $\mathbf{J}$. Back substitutions are much less computationally demanding. In some cases the elements of the Jacobian does not vary continuously with time. The Jacobian may for example only change at events and it is then only necessary to calculate and factorize it during event iterations and not during continuous simulation. In other cases, it may depend only on parameters and constants and then it needs only to be calculated once, at the start of a simulation.

When using Newton methods for non-linear equation systems, it is necessary to calculate the Jacobian. If this is made numerically from residuals, then n residual calculations are needed. Dymola provides analytic Jacobians. These are more accurate and much less computationally demanding, because there are many common subexpressions to exploit. Modelica provides facilities to provide derivatives also for external functions.

13.8.3 Index reduction

When solving an ordinary differential equation (ODE) the problem is to integrate, i.e. to calculate the states when the derivatives are given. Solving a DAE may also include differentiation, i.e. to calculate the derivatives of given variables. Such a DAE is said to have high index. It means that the number of states needed for a model is less than the number of variables appearing differentiated. The number of states is equal to the number of independent initial conditions that can be imposed. Higher index DAEs are typically obtained because of constraints between models. To support reuse, model components are developed to be "general". Their behavior is restricted when they are used to build a model and connected to other components. Take as a very simple example two rotating bodies with inertia J_1 and J_2 connected rigidly to each other. The angles and the velocities of the two bodies should be equal. Not all four differentiated variables can be state variables with their own independent start values. The connection equation for the angles, $\varphi_1 = \varphi_2$, must be differentiated twice to get a relation for the accelerations to allow calculation of the reaction torque.

The reliability of a direct numerical solution is related to the number of differentiations needed to transform the system algebraically into ODE form. Modern numerical integration algorithms for DAEs, such as used by most simulators, can handle systems where equations needed to be at most differentiated once. However, reliable direct numerical solutions for non-linear systems are not known if two or more differentiations are required. Furthermore, if mixed continuous and discrete systems are solved, the hybrid DAE must be initialized at every event instant. In this case, it is in general not sufficient to just fulfill the original DAE. Instead, also some differentiated equations have to be fulfilled, in order that the initialization is consistent. Direct numerical methods have problems at events to determine consistent restart conditions of higher index systems.

Higher index DAEs can be avoided by restricting how components may be connected together and/or include manually differentiated equations in the components for the most common connection structures. The drawback is (1) physically meaningful component connections may no longer be allowed in the model or (2) unnecessary "stiff" elements have to be introduced in order that a connection becomes possible. For example, if a stiff spring is introduced between the two rotating bodies discussed above, the problem has no longer a higher index.

Since most Modelica libraries are designed in a truly object-oriented way, i.e., every meaningful physical connection can also be performed with the corresponding Modelica components, this leads often to higher index systems, especially in the mechanical and thermo-fluid field. Also modern controllers based on non-linear inverse plant models lead to higher index DAEs (Looye et.al. 2005) and can be conveniently treated with Dymola.

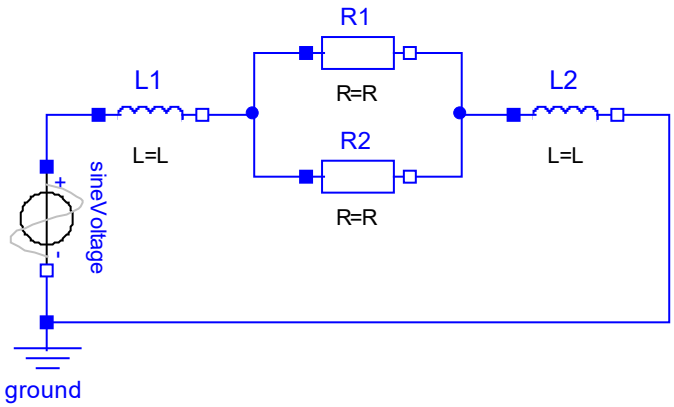
Dymola transforms higher index problems by differentiating equations analytically. The standard algorithm by Pantelides (Pandelides 1988) is used to determine how many times each equation has to be differentiated. The algorithm by Pantelides is based on the structure of the equations. It means that there are examples where it does not give the optimal result (Reissig, Martinsson and Barton, 1999). However, the practical experience is very good. Moreover, for large problems a structural analysis is the only feasible approach. Selection of which variables to use as state variables is done statically during translation or in more complicated cases during simulation with the dummy derivative method (Mattsson and Söderlind 1993, Mattsson et.al. 2000). Let us make the example above a bit more realistic and put a gearbox with fixed gear ratio n between the two bodies. Dymola differentiates the position constraint twice to calculate the reaction torque in the coupling, and it is sufficient to select the angle and velocity of either body as state variables. The constraint leads to a linear system of simultaneous equations involving angular accelerations and torques. The symbolic solution contains a determinant of the form "$J_1 + n^2 J_2$". Dymola thus automatically deduces how inertia is transformed through a gearbox.

Handling of implicit constraints between potential state variables

(This section is not included in the original article; it is later improvement in Dymola 7.1 and later.)

The index reduction algorithm basically focuses on the structure of the equations. The practical experience of this approach is good. However, there are examples were the results is not satisfactory. Such examples appear typically in models of electrical circuits.

Consider the example



The model has two potential state variables, namely the currents through the inductors: L1.i and L2.i. However, they cannot be selected as states simultaneously because these currents must be equal, L1.i = L2.i. In this example it is easy to see for a human being. Unfortunately, the original Pantelides's algorithm will not detect this constraint, because it is too implicit in the model equations. After alias elimination of the connector variables, the zero sum current equations for the connections between the inductors and the resistors are

```
L1.i =  R1.i + R2.i
L2.i = -R1.i - R2.i
```

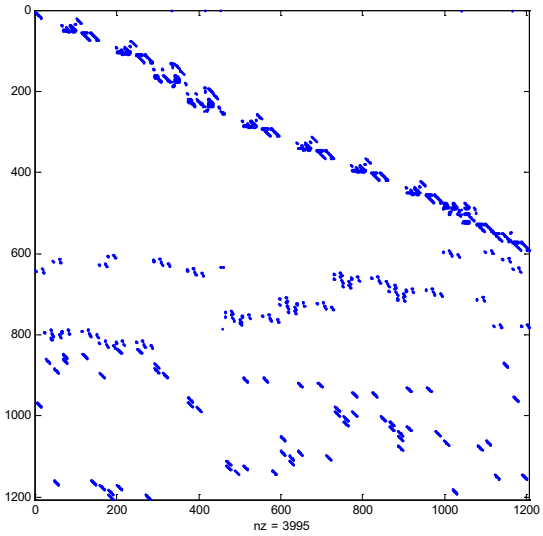
Assuming L1.i and L2.i to be states, the structural analysis of Pantelides's algorithm indicates that R1.i and R2.i can be solved from these two equations. However, this is not true and the simulation will fail because the system actually is singular from that respect. It is easy to see by just adding the two equations giving L1.i = L2.i.

The improved support of index reduction looks for such kinds of implicit constraints between potential state variables and manipulates the equations to make them explicit.

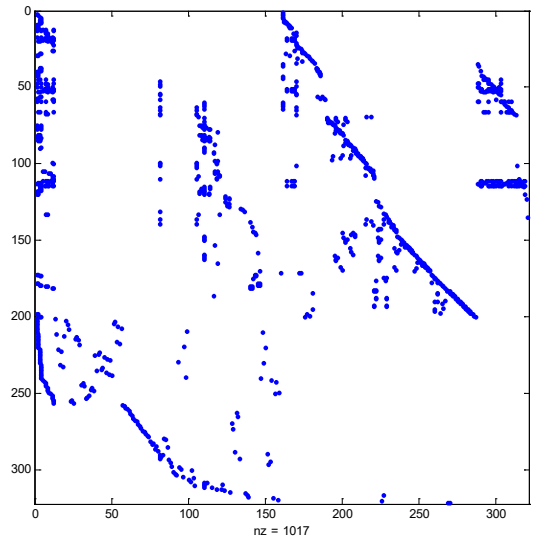
13.8.4 Example

To illustrate how Dymola's symbolic processing reduces the size and complexity we will show the structure Jacobian at different stages when translating a mechanical model with a kinematic loop.

The structure Jacobian of the original model.



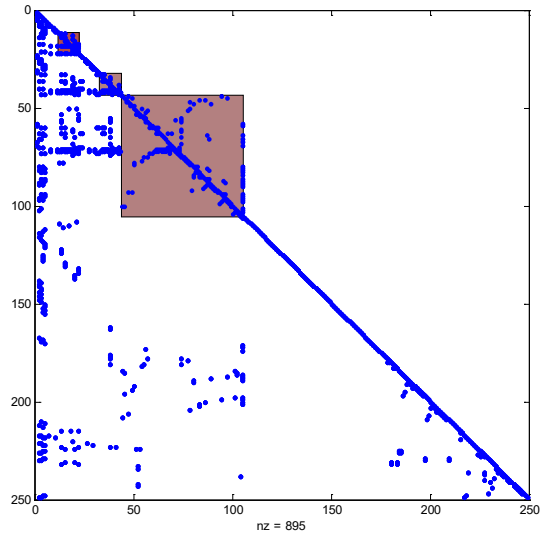
The structure Jacobian after elimination of alias variables.



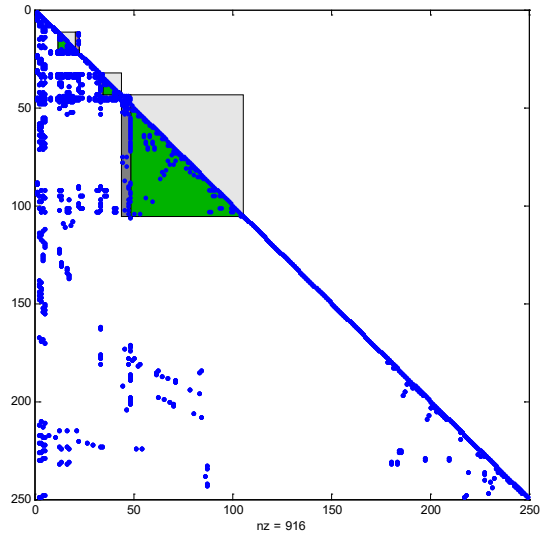
The upper figure above shows the structure Jacobian of the original model. There are about 1200 unknown variables and equations. Each row corresponds to an equation and each column corresponds to a variable. A blue marker indicates that the variable appears in the equation. There are 3995 markers. The upper half of the matrix has a banded structure. These equations are the equations appearing in the component models and such equations refer typically only to the local variables of the component. The equations in the lower part are equations deduced from the connections, which includes references to variables of two or more components.

The lower figure above shows the structure of the problem after exploitation of simple equations to eliminate alias variables and utilizing zero constants. The number of unknowns is reduced from about 1200 to 330.

The BLT partitioning.



The structure after tearing.



Then equations are differentiated to reduce the DAE index and states are selected. After some further simplifications the number of unknowns is reduced to 250. A BLT partitioning reveals that there are 3 algebraic loops as indicated by the upper figure above.

The lower figure above shows the structure after tearing. The first algebraic loop is a nonlinear loop with 12 unknowns. This loop includes the positional constraints of the kinematics loop.

The tearing procedure reduces the number of iteration variables to 2. This is illustrated by turning the eliminated part from grey to green. The second loop includes the velocity constraints due to the kinematic loop. It means that this loop includes the equations of the positional constraints differentiated. This loop has also 11 unknowns, but it is linear. The remaining two by two system can be solved symbolically or numerically. The third loop includes acceleration and force/torque as unknown variables. The loop is linear and has 62 unknowns and the tearing procedure eliminates 57 so a linear 5 by 5 system remains to be solved numerically.

13.8.5 Alias elimination of parameters

Alias elimination of parameters can be obtained by setting the flag

```
Advanced.AllowParameterAlias=true
```

This eliminates parameters equal to other parameters –in the same way as alias elimination of time-varying variables. This is primarily intended to reduce the size of the generated C-code, and the only down-side is that such parameters are treated a bit specially when exporting the model to FMI.

13.8.6 Efficient code generation for large tables

Large tables in Modelica models will by default yield a large amount of extra C code to be compiled and fills up the variable browser with table data which may be of limited interest. It is possible to avoid this overhead by setting the annotations `Evaluate=true` and `HideArray=true` on the table, combined with the global setting `Advanced.AllowParameterAlias=true` (see previous section).

An example:

```
model HideArray
  parameter Real myTable[:,2]=[linspace(0, 30, 50), sqrt(linspace(0.0, 100, 50))]
    annotation (Evaluate=true, HideArray=true);

  Modelica.Blocks.Tables.CombiTable1D combiTable1D(table=myTable) ∃ ;
  Modelica.Blocks.Sources.Clock clock ∃ ;
equation
  connect(combiTable1D.u[1], clock.y) ∃ ;
  annotation (__Dymola_experimentFlags(Advanced(AllowParameterAlias=true)));
end HideArray;
```

13.8.7 References

Duff, I.S, A.M. Erisman, and J.K. Reid. 1986. *Direct Methods for Sparse Matrices*, Clarendon Press, Oxford.

Dynasim. 2006. Dymola Version 6.0. Dynasim AB, Lund, Sweden. Homepage: http://www.dynasim.se/.

Elmqvist Hilding. 1978. *A Structured Model Language for Large Continuous Systems*. Dissertation. Report CODEN:LUTFD2/(TFRT--1015), Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, 1978.

Elmqvist, H., and M. Otter. 1994. *Methods for Tearing Systems of Equations in Object-Oriented Modeling*. Proceedings ESM'94, European Simulation Multiconference, Barcelona, Spain, June 1-3, pp.326--332.

Elmqvist, H., S. E. Mattsson, and M. Otter. 2001. *Object-Oriented and Hybrid Modeling in Modelica*. Journal Europeen des Systemes Automatises, 35, pp. 1 a X.

Looye, G., M. Thümmel, M. Kurze, M. Otter, and J. Bals. 2005. Nonlinear Inverse Models for Control. Proceedings of the 4th International Modelica Conference, Hamburg, ed. G. Schmitz,. http://www.modelica.org/events/Conference2005/online_proceedings/Session3/Session3c3.pdf

Mattsson, S. E., and G. Söderlind. 1993. *Index reduction in differential-algebraic equations using dummy derivatives*. SIAM Journal of Scientific and Statistical Computing, Vol. 14 pp. 677-692.

Mattsson, S. E., H. Olsson and H. Elmqvist. 2000. Dynamic Selection of States in Dymola. Proceedings of the Modelica Workshop 2000. pp. 61-67. http://www.modelica.org/Workshop2000/papers/Mattsson.pdf.

Modelica. 2005. *Modelica® - A Unified Object-Oriented Language for Physical Systems Modeling - Language Specification, Version 2.2*. http://www.Modelica.org/Documents/ModelicaSpec22.pdf.

Pantelides C.. 1988. *The consistent initialization of differential-algebraic systems*. SIAM Journal of Scientific and Statistical Computing, pp. 213-231.

Reissig, G., W. S. Martinsson, and P. I. Barton . 1999. Differential-Algebraic equations of index 1 may have an arbitrarily high structural index. SIAM J. Sci. Comp.

Tarjan, R.E. 1972. *Depth First Search and Linear Graph Algorithms*, SIAM J. Comput. 1, pp. 146-160.

13.9 Symbolic solution of nonlinear equations in Dymola

13.9.1 Introduction

In the general case it is not possible to solve a nonlinear equation analytically. However, Dymola supports some special cases:

1. Solving a nonlinear equation with single appearance of the unknown by applying function inverses
2. Solving a nonlinear equation with special patterns for the unknown such as abs(w)*w

3. Partitioning of a system of equations into a linear and a nonlinear (one variable) part
4. Using min and max values to evaluate if-conditions

These items will be discussed in turn.

13.9.2 Solving a nonlinear equation with single appearance of the unknown by applying function inverses

A single appearance of the unknown allows the technique of successive inverses to be applied. It requires the inverses of the appearing functions to be known. The basic idea is simple. However, the issues of nonexistent or multiple solutions need to be considered.

When translating a model, Dymola sorts the equations to determine which variable to solve for in each equation. The result of the sorting procedure of the initialization problem and the simulation problem is a sequence of sub-problems which can be solved in turn. The feature to solve a nonlinear equation symbolically is useful when such a sub-problem has one unknown, x, that appears nonlinearly in the equation. You may say that the computational causality of the initialization or simulation problem requires "inversion" of the equation.

The feature to solve a nonlinear equation symbolically is activated when Dymola finds that solving a sub-problem with one unknown is a nonlinear problem. If Dymola is not able to solve the equation symbolically, then code is generated for numerical solution as previously.

Solving nested functions

Consider the equation

```
f1(f2(x,v), v) = f3(v);
```

Let x denote the scalar real variable to be solved for and let the vector v denote the remaining appearing variables which are considered to be known variables. When solving nested functions, the approach is to introduce an auxiliary variable, w, to decompose the problem as

```
f1(w, v) = f3(v);
f2(x, v) = w;
```

where the first equation is used to solve for w and if successful, then the solution procedure is applied recursively on the second equation to solve for x. This includes also the functions +, -, * and /. For example, let f1 in the example above be +, implying the problem

```
f2(x,v) + v = f3(v);
```

The decomposition becomes

```
w + v = f3(v)
f2(x, v) = w;
```

Below we will focus on solving f(x) = v.

No solution exists

When solving an equation

```
f(x) = v;
```

it is important to check that the equation has a solution. It is easy to produce false solutions. Consider the equation

```
sqrt(x) = v;
```

It has the unique solution x = v*v when v ≥ 0. However, when v < 0, there is no solution. Note, that the right hand side of the equation x = v*v is well-defined also for v < 0, but the result is not a solution to the original equation. Thus Dymola transforms the equation as

```
assert(v >= 0, "Cannot solve sqrt(x)=v when v < 0");
x = v*v;
```

If f(x) is not onto the real axis (the range of f(x) is not the whole real axis), it is necessary to add asserts to check if v belongs to the range of f. However, sometimes the check can be deferred, because calculating the "inverse" will fail. Consider the equation

```
exp(x) = v;
```

It has the unique solution x = ln(v) when v > 0, but no solution when v <= 0, so Dymola solves it as

```
x = ln(v)
```

If v <= 0, the failure to calculate ln(v) will trig an assertion and stop the simulation.

Multiple solutions

The equation

```
f(x) = v;
```

may have multiple solutions. As an example, consider the equation

```
abs(x) = v;
```

First, it has no solution if v < 0, so it is necessary to generate an assert for that. Second, when v > 0, there are two solutions:

```
x =  v;
x = -v;
```

A numerical solver needs a start or guess value and it then hopefully returns the solution closest to the guess value. This is most critical during initialization and during event iteration, because v may then jump. During continuous time integration, v shall change continuously and the resulting solution x shall be smooth. Dymola's approach is to mimic this behavior. The solution looks as

```
assert(v >= 0, "Cannot solve abs(x)=v when v < 0");
x := if x>= 0 then v else -v;
```

Supported functions

Modelica and Modelica.Math functions

Dymola currently support symbolic inversion of the following Modelica functions

- abs
- noEvent, smooth
- x^a, a^x, sqrt

and the following Modelica.Math functions

- sin, cos, tan
- exp, ln, log10
- asin, acos, atan

Annotations

Functions may have an additional annotation to define an inverse of the function:

```
function f1
   input  Real x;
   input  Real y;
   input  Boolean b[4,3];
   output Real z;
   annotation(__Dymola_inverse(y=f2(z,x,b));
algorithm
   ..
end f1;
```

The meaning is that function "f2" is an inverse to the function "f1" where the previous output "z" is now an input and the previous input "y" is now an output.

The inverse requires that for all valid values of the input arguments of f2(z,x,b) and y being calculated as y := f2(z,x,b) implies the equality z = f1(x,y,b).

Function "f1" can have any number and types of arguments. The current restriction is that both "f1" and "f2" must have exactly one scalar Real output argument and that "f2" must have exactly the same arguments as "f1", but the order of the arguments may be permuted.

Inlining of inverse functions

Functions with annotations that indicate that they have inverses or can be differentiated are by default inlined after that information is used.

This behavior can be disabled by setting

```
Advanced.Translation.DelayInliningForInverseDerivatives = false;
```

Disabling inlining using the annotation `Inline=false` or specifying it using any of the annotations `LateInline=true` or `InlineAfterIndexReduction=true` takes precedence over this behavior.

Deactivation of supported functions

The feature is by default activated. It can be disabled by setting the flag

```
Advanced.SolveNonlinearEquationSymbolically = false;
```

13.9.3 Solving a nonlinear equation with special patterns for the unknown

Above it was assumed that the unknown variable to solve for only appeared once. However, the feature to solve nonlinear equations also supports some special patterns

```
x*x = v;
x*abs(x) = v; abs(x)*x = v
sign(x)*sqrt(abs(x)) = v; sqrt(abs(x))*sign(x)
```

The two last equations may appear in modeling of the pressure drop, dp, characteristics with respect to the mass flow rate w:

```
dp = c*w*abs(w);
w = sign(dp/c)*sqrt(abs(dp/c);
```

The recursive scheme discussed above also supports the case when the unknown parts are expressions depending on x, for example

```
f(x, v)*f(x, v) = v;
```

as long as Dymola can establish that the two factors are equal expressions.

The equation x*x = v; is solved in a similar way as abs(x) = v:

```
assert(v>=0);
x = if x >= 0 then sqrt(v) else -sqrt(v)
```

The equations x*abs(x) = v; and abs(x)*x = v have the solution

```
x = sign(v)*sqrt(abs(v))
```

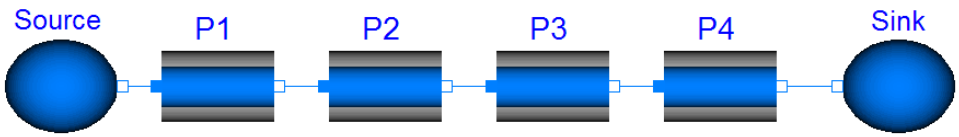
The equations sign(x)*sqrt(abs(x)) = v; and sqrt(abs(x))*sign(x) = v have the solution

```
x = v*abs(v)
```

13.9.4 Partitioning of a system of equations into a linear and nonlinear (one variable) part

When modeling flow systems with a number of flow elements (pipes, ducts, valves etc.) in series there will typically be a nonlinear system of equations to solve for flow rate. Introduction of storage elements will break such loops. However, there is the cost of extra dynamics. Moreover, making the volumes small makes the problem stiff.

Consider the example below with four flow elements connected in series between a source and a sink with given pressures.



The pressure drop, dp, of a flow element is modeled by quadratic characteristics with respect to the mass flow rate m_flow as

```
dp = noEvent(c*m_flow*abs(m_flow));
```

where c is independent of dp and m_flow, but it may depend on physical dimensions of the component. After elimination of simple equations the simulation problem includes a nonlinear system of equations with 8 unknowns

```
P1.dp, P1.m_flow, P2.dp, P2.port_a.p, P3.dp, P3.port_a.p,
P4.dp, P4.port_a.p
```

and the equations

```
P4.dp = P4.c*P1.m_flow*abs(P1.m_flow);
P4.dp = P4.port_a.p-Sink.port.p;
P3.dp = P3.port_a.p-P4.port_a.p;
P2.dp = P2.c*P1.m_flow*abs(P1.m_flow);
P2.dp = P2.port_a.p-P3.port_a.p;
P1.dp = Source.port.p-P2.port_a.p;
P1.dp = P1.c*P1.m_flow*abs(P1.m_flow);
P3.dp = P3.c*P1.m_flow*abs(P1.m_flow);
```

The algebraic loop is nonlinear only because of the expression

```
P1.m_flow*abs(P1.m_flow);
```

Moreover, P1.m_flow appears in no other ways in the equations of the algebraic loops. By substituting all appearances of the expression P1.m_flow*abs(P1.m_flow) by w_expr, the problem decomposes into two sub-problems

- A linear system of equations that is independent of P1.m_flow.

```
P4.dp = P4.c* w_expr;
P4.dp = P4.port_a.p-Sink.port.p;
P3.dp = P3.port_a.p-P4.port_a.p;
P2.dp = P2.c* w_expr;
P2.dp = P2.port_a.p-P3.port_a.p;
P1.dp = Source.port.p-P2.port_a.p;
P1.dp = P1.c*P1 w_expr;
P3.dp = P3.c* w_expr
```

- A nonlinear equation for P1.m_flow

```
P1.m_flow*abs(P1.m_flow) = w_expr;
```

This decomposition is valuable, because the nonlinearity has been reduced to one equation; to solve a scalar nonlinear equation. Dymola manipulates the two subsystems in the usual way and solves the linear system symbolically.

The feature to solve nonlinear equations also supports support solving a nonlinear equation with single appearance of the unknown by applying function inverses. It is also able to handle some special patterns

```
x*x = v;
x*abs(x) = v; abs(x)*x = v
sign(x)*sqrt(abs(x)) = v; sqrt(abs(x))*sign(x)
```

The scheme discussed supports the case when the unknown parts are expressions depending on x, for example

```
f(x, v)*f(x, v) = v;
```

as long as Dymola can establish that the two factors are equal expressions.

The nonlinear equation for P1.m_flow

```
P1.m_flow*abs(P1.m_flow) = w_expr;
```

is solved symbolically as

```
P1.m_flow :=
       noEvent(sign(w_expr)*sqrt(abs(w_expr)));
```

13.9.5 Using min and max values to evaluate if-conditions

An if-then-else expression may cause the appearances of algebraic loops. A typical case is flow systems. The model components are built to handle either flow direction as well as reversing flows. It means that the equations include if-then-else expressions depending on the actual flow direction:

```
d = if m_flow >= 0 then medium_a.d else medium_b.d;
H_flow = semiLinear(m_flow, medium_a.h, medium_b.h);
```

According to definition of the Modelica function semiLinear, the last equation means

```
H_flow = if m_flow >= 0 then m_flow*medium_a.h
                        else m_flow*medium_b.h;
```

If we are simulating a flow system at normal operation, the flow direction shall not change and it is known, say m_flow >= 0. By using this information it is possible to significantly simplify the equations to

```
d = medium_a.d;
H_flow = m_flow*medium_a.h;
```

Please, note that neither medium_b.d nor medium_b.h appear in the resulting equations. The elimination of them may make an algebraic loop decrease in size or even disappear.

At translation Dymola of course evaluates all constant conditions and simplifies the equations accordingly. In this case Dymola is dealing with relations where the expressions of the relations may be time varying to see if it nevertheless is possible to evaluate the condition by exploiting the values of the min and max attributes of variables. To specify that the flow direction is known, m_flow >= 0, is done by setting the min value of m_flow to 0, i.e., m_flow(min=0).

If the assumption that m_flow >= 0 turned out to be false, the normal assert that the value must not be less than min will be triggered.

This feature may be disabled by setting the flag

```
Advanced.UseMinMaxToSimplify = false
```

14 VISUALIZE 3D

14 Visualize 3D

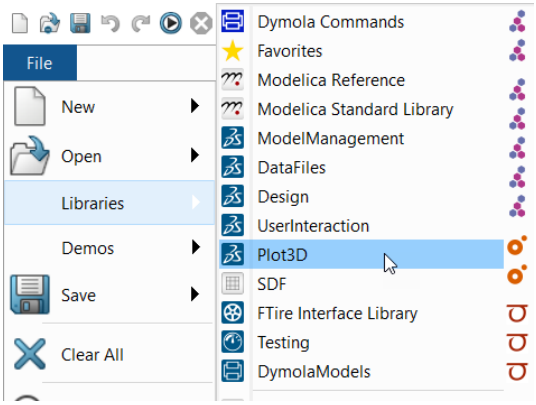
14.1 Introduction

Data visualization in 3D is an important way of representation, and it is adequate for understanding and comprehending model behavior. Dymola includes a 3D graphical tool: Visualize 3D.

Visualize 3D renders 3D scenes and has an associated Modelica package named Plot3D. This package manipulates and sends the graphical data representation of the scene to Visualize 3D. This guide describes how to use Plot3D to obtain graphs and figures with the Visualize 3D tool in Dymola.

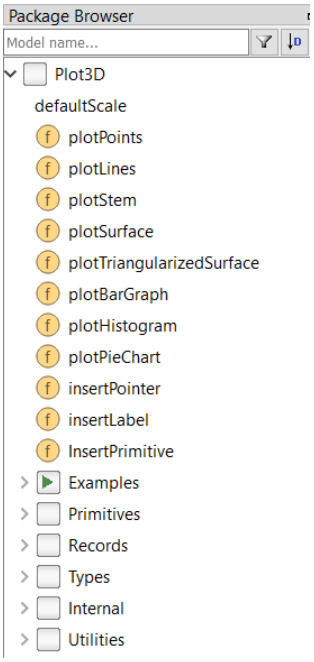
To open the Plot3D package, use the command **File > Libraries > Plot 3D**:

Opening the Plot3D package.



The package browser of Plot3D will look the following:

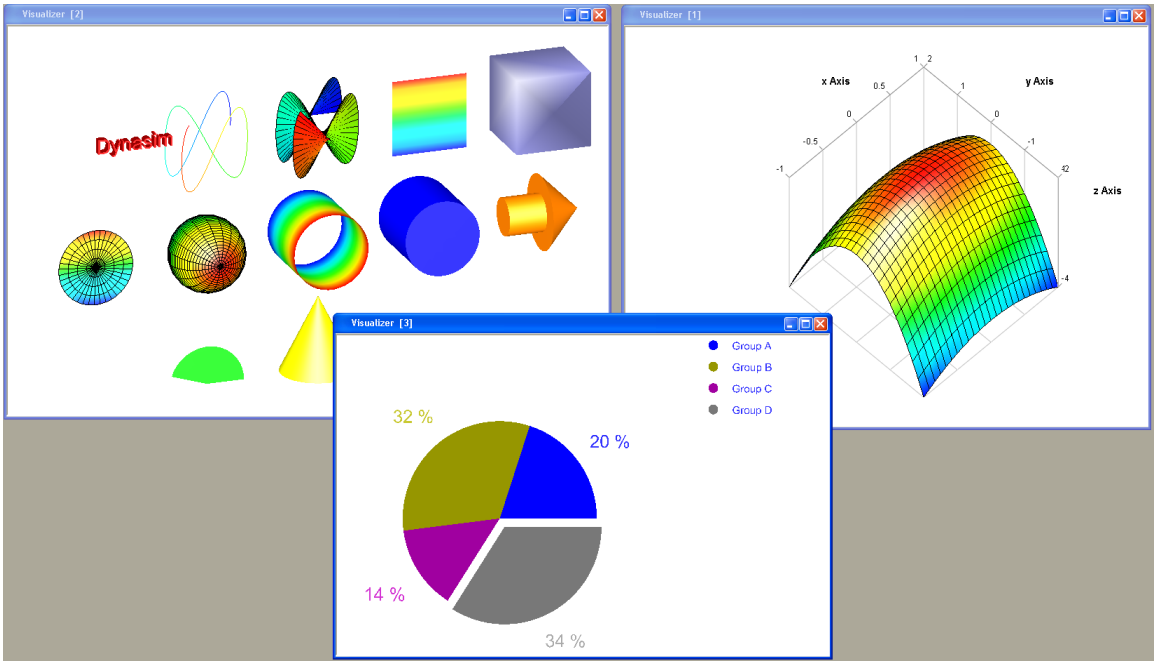
Plot3D packages.



The main functions are at top level of the package: plotPoints, plotLines, plotStem, plotSurface, plotTriangularizedSurface, plotBarGraph, plotHistogram, plotPieChart, insertPointer, insertLabel and insertPrimitive. We recommend strongly using these high-level functions instead of trying to use the low-level ones in Plot3D.Internal.

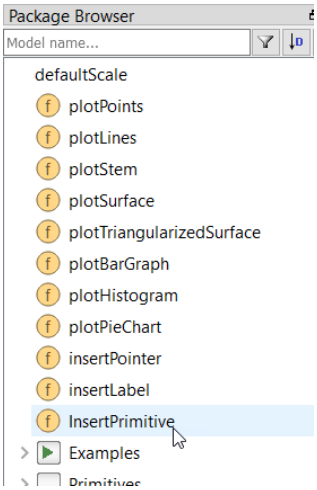
The sub-package Plot3D.Primitives contains the basic primitives preset. The sub-packages Records and Types also contain information about the internal representation of a 3D scene. The sub-package Examples contains in its turn some of the examples presented here and we will refer to them later on.

Visualize 3D supports several different types of plots and can be presented separately in their own windows if desired, all integrated in the Simulation tab.

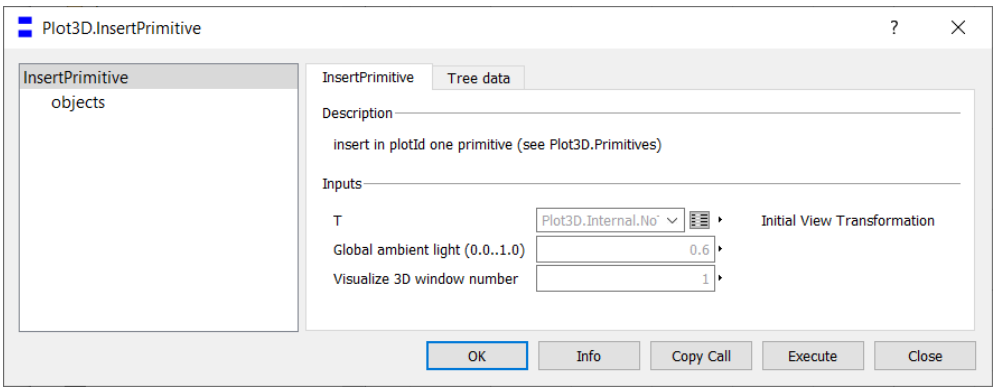


14.2 Inserting and removing graphical objects

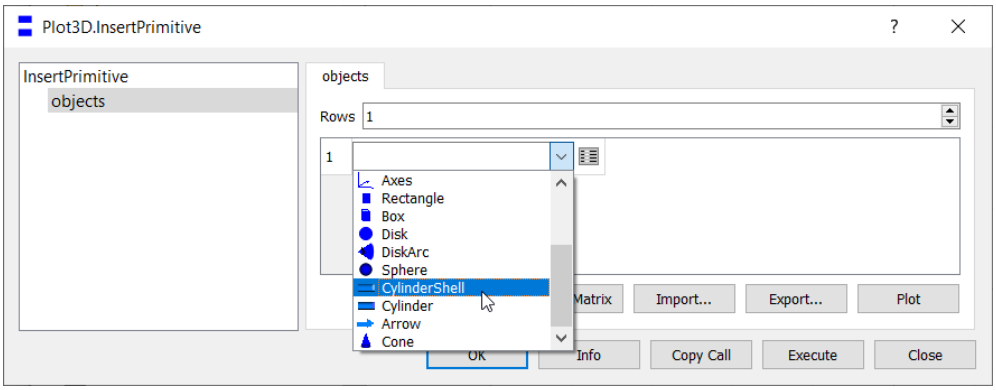
Visualize 3D has several basic primitives that can be combined to construct more complicated scenes. We will start by constructing a simple solid cylinder by combining a cylinder shell with two disks. We start by using the function `Plot3D.InsertPrimitive`



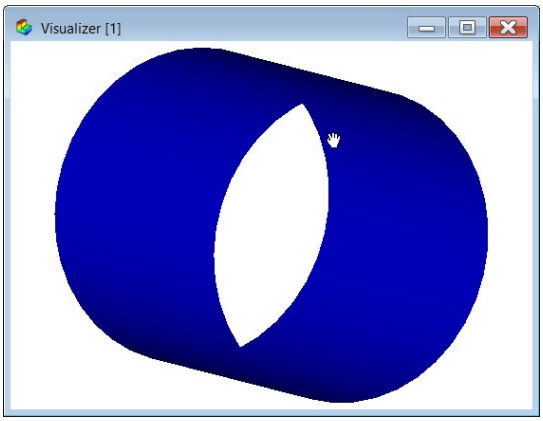
Right-click on it and select **Call Function**. The following dialog will pop up:



The first elements we observe are the View transform matrix T, the global ambient light and Visualize 3D window number. This number identifies the window we want to add some primitive to. We keep the default values now and click on **objects** field in the tree. There we are to select the primitive forms to be added. Click on the arrow of the combo box and scroll down until **CylinderShell**.



We have now selected a cylinder shell and we can plot it with default values. Press **Execute**. At first sight there is just an empty Visualize 3D window. Actually, we are looking at the shell with zero thickness along its main axis. In order to see the figure, press the **Ctrl** key and move the mouse to rotate along the axes x and y. The figure below shows one possible view of the new created cylinder shell.

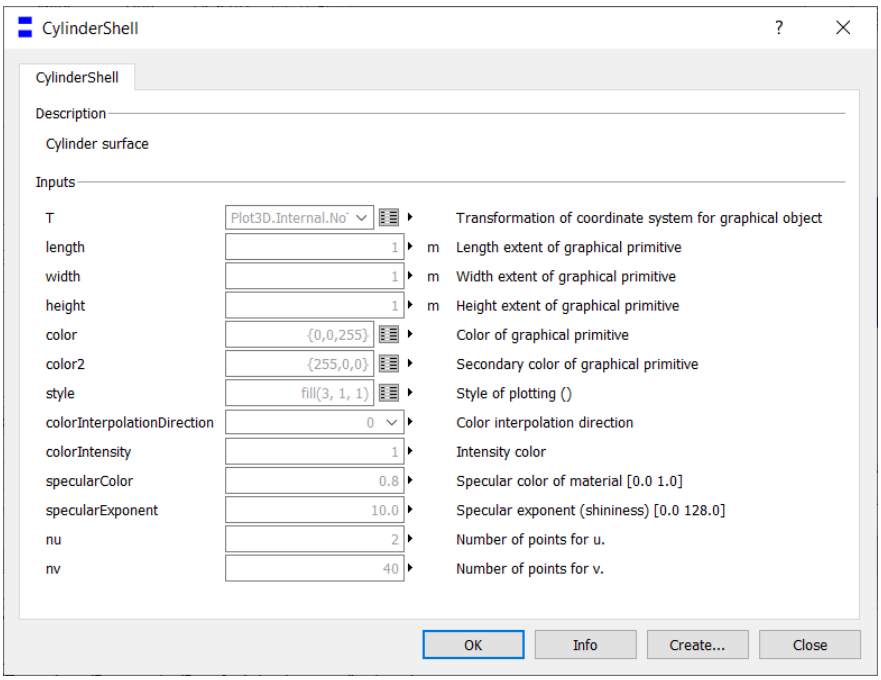


The operation of Visualize 3D can be summarized in the following table:

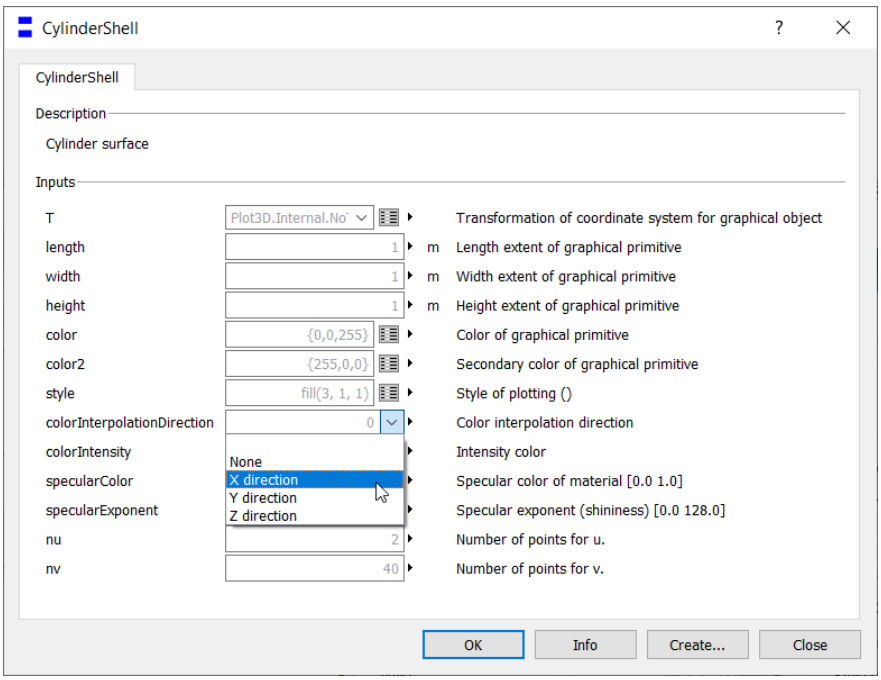
Operation	Meta key	Mouse move (dragging)	Arrow keys
Selecting object	Alt+Select		
Moving up/down/left/right	none	Up/Down/Left/Right	Up/Down/Left/Right
Tilt (Rotate around x-axis)	Ctrl	Up/Down	Up/Down
Pan (Rotate around y-axis)	Ctrl	Left/Right	Left/Right
Roll (rotate around z-axis)	Ctrl+Shift	Clockwise/Counter-clockwise	Right/Left
Zoom in/out	Shift	Up/Down	Up/Down
Zoom in/out	none	Wheel	
Zoom in/out	Ctrl	Wheel	
Zoom in/out	Ctrl	Right mouse button Up/Down	



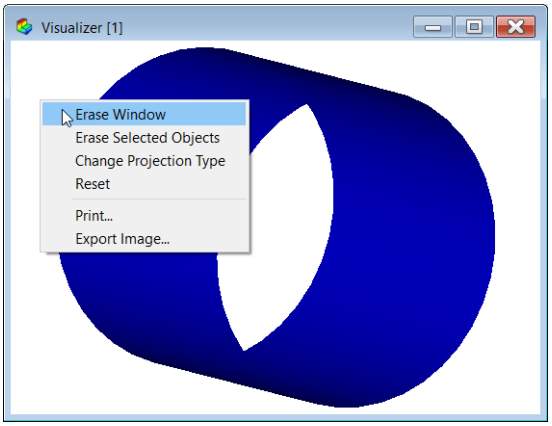
We can also perform other operations on the cylinder shell. Going back to the dialog window and clicking on the **Edit** icon, we get the following dialog window. (Note that it might seem that the dialog has disappeared, but it is behind the main Dymola window. You can hover over the Dymola symbol in the status bar in the bottom of the Windows window to be able to select the dialog to be shown on top of the Dymola window.)



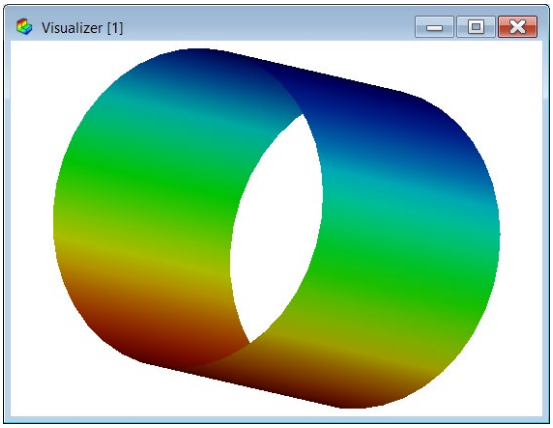
We observe different graphical properties of the cylinder shell primitive. We are now interested in a few: matrix T, length, color, style, colorInterpolationDirection and colorIntensity. Change colorInterpolationDirection to “x direction”, press **OK** in the edit menu and press **Execute** once more.



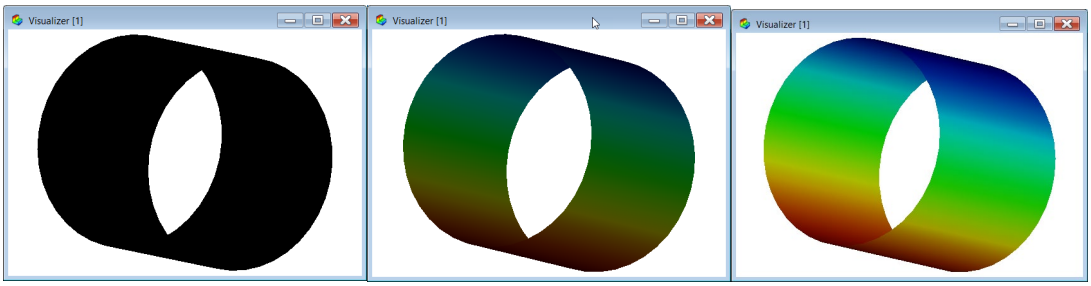
You don´t see any change, but remember that we are adding primitives; this means that if the intention is to change and paint again, the Visualize 3D window has to be erased. This can be done by right-clicking in the window and selecting **Erase window** in the context menu, as below. This operation will clean the window object list.



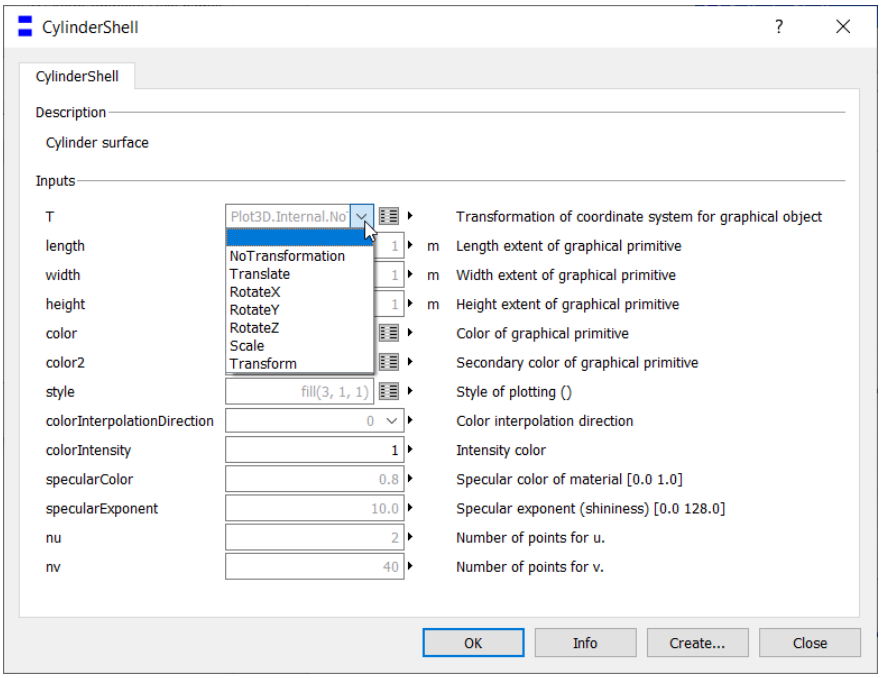
Now, perform this cleaning operation and then select **Execute** again in the dialog. The change is that Visualize3D interpolates the color using the range of the x coordinate of the primitive.



Changing the colorIntensity parameter it is possible to set the brightness of the color scheme applied. This factor is to be in the interval [0,1]. Below we find depicted the cylinder shell for intensityColor=0, 0.5 and 1.

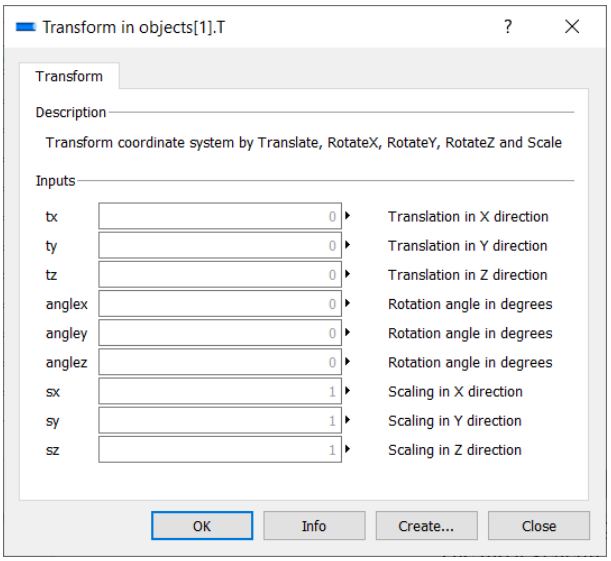


The matrix T is used to perform transforms on just the associated graphical object. Operations like translation, scaling and rotation of the body respect to the global coordinate system are described with this T matrix. These transforms are independent of the global view, and are used to construct the 3D scene. Clicking on the combo box arrow shows the predefined possibilities.



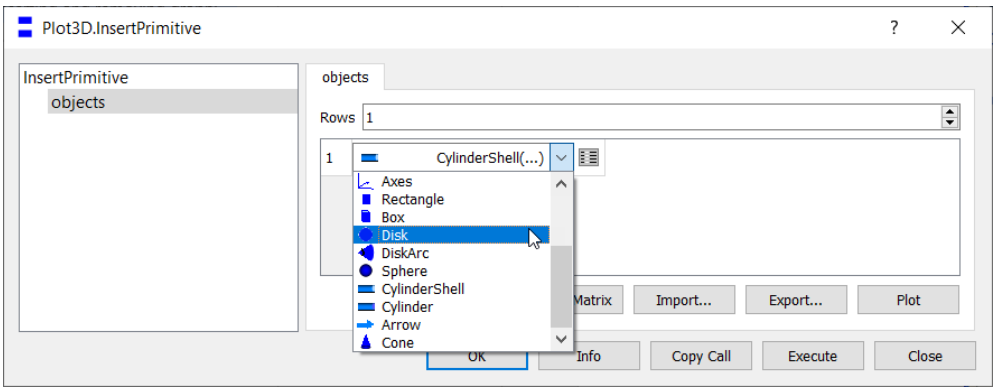
We can select NoTransform, Translate, RotateX, RotateY, RotateZ, Scale and Transform. The most general of the operations is Transform that involves a combination of all the others.

The dialog window for “Transform” is the following:



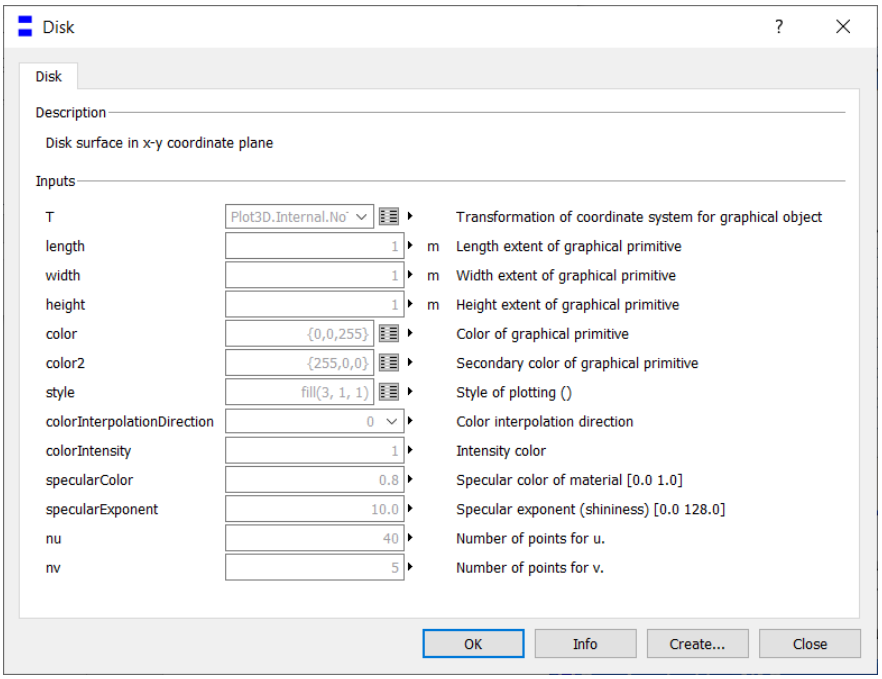
Here we can describe what we want to do with the graphical object. The order is important, since all these operations are not commutative, for instance, it is not the same to rotate and translate as to translate and then rotate. The order preset is scale first, rotate around Z, rotate around Y, rotate around X and then translate.

Let us now add the top and bottom of the cylinder. Again in the dialog window, we change the “CylinderShell” primitive to the “Disk” primitive.

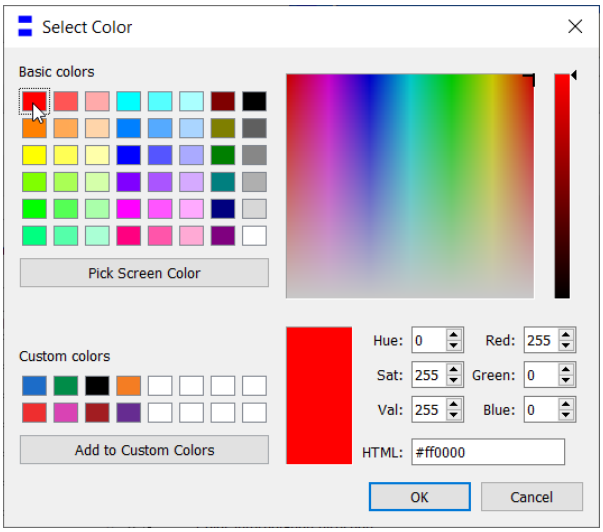




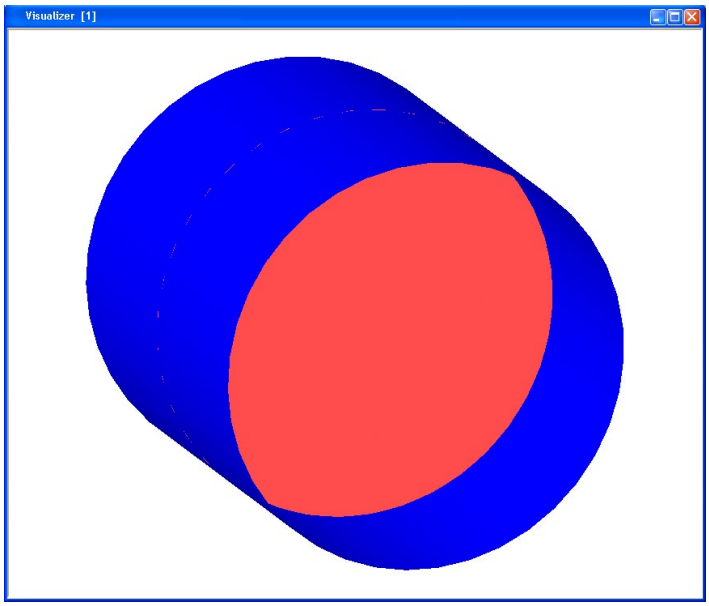
Click on the **Edit** button to pop the menu for the disk.



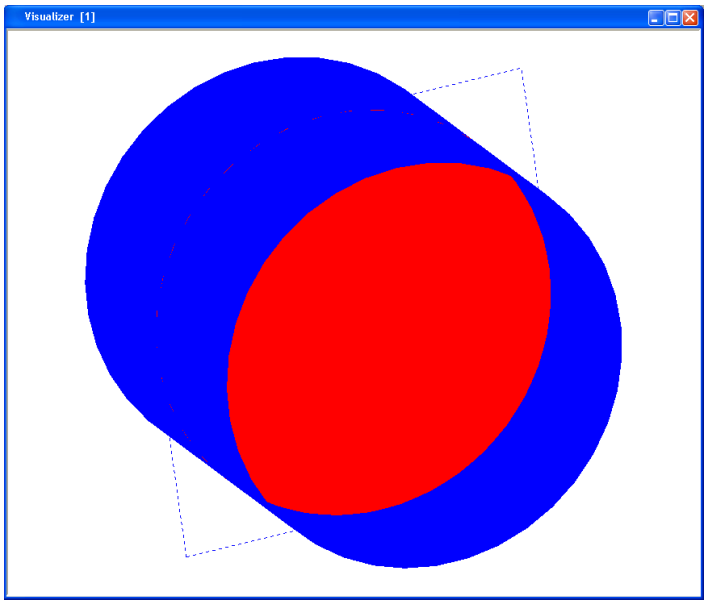
Change the color of the “Disk” by pressing the **Edit** icon of the field “color”. We choose in this case the red color to get a good contrast.



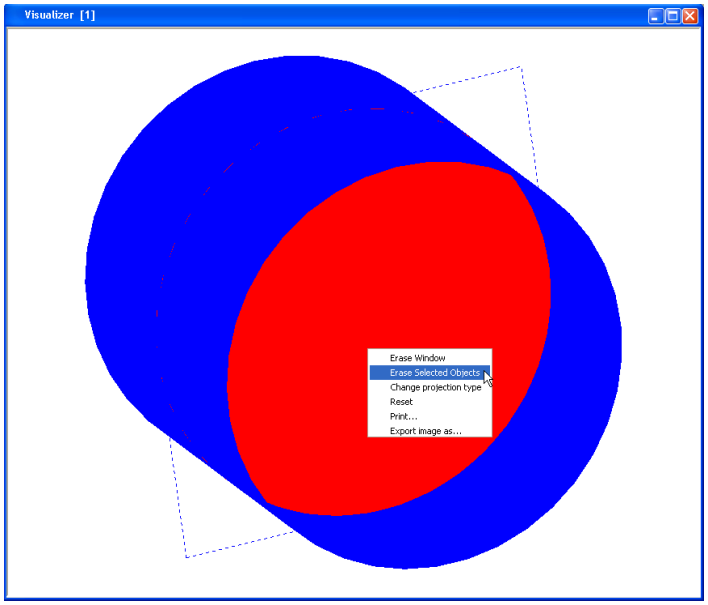
Click **OK** in the color window, click **OK** in the edit window and press **Execute** and finally rotate once more using the **Ctrl** key and moving the mouse. We observe the following result:



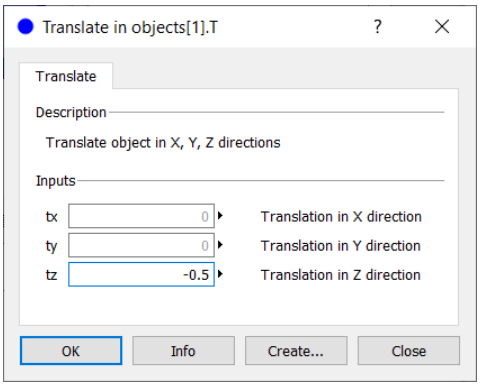
The disk is in the middle by default. We want to place the disks on top and bottom of the cylinder shell. We will therefore erase the disk and place it correctly using the translate transform. To erase a graphical object, we have to select it first by clicking on it while pressing the **Alt** key. The selected object will then be delimited by a dotted box.



Now we select from the context menu **Erase Selected Objects**, and the disk is erased from the actual view.

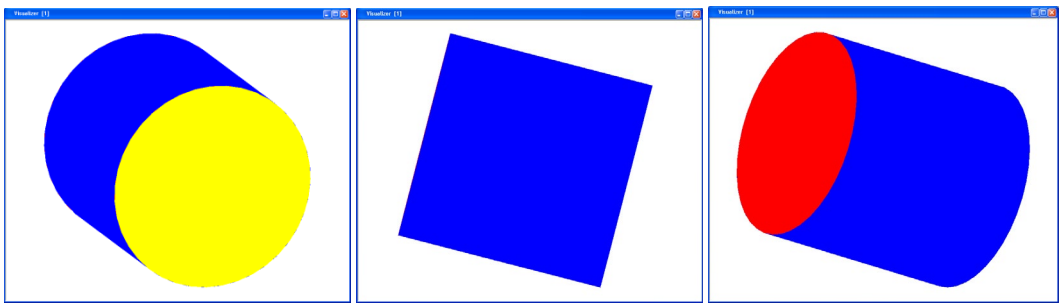


To close the cylinder shell, we have then to set the bottom disk at the point $(0, 0, -0.5)$ and the top disk at $(0, 0, 0.5)$, since the cylinder has length 1. In the window "Disk", click on the edit icon of "T" and change "NoTransformation" to "Translate" using the arrow in the combo box. A menu will appear. Set "tz" to -0.5 in this menu.



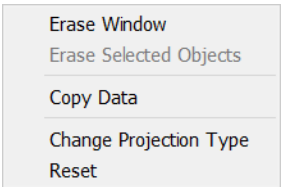
Press **OK** in this menu. It will disappear. Press **Execute**. A bottom disk will appear on the cylinder.

To create a top disk for the cylinder, go back to the "Disk" window; change the color to yellow (to be able to see all components clearly). Then click on the edit icon of "T" change tz to 0.5 and press **OK** and **Execute** again. We obtain now a closed cylinder as below.



Here we see top, side and bottom of the newly created cylinder. The primitive Plot3D.Primitives.Cylinder is constructed with this technique, encapsulating all necessary steps to get a uniform color, size and other properties.

The context menu of the visualizer window can be summarized as:



Erase Window will erase all objects in the window.

Erase Selected Objects will erase the selected objects.

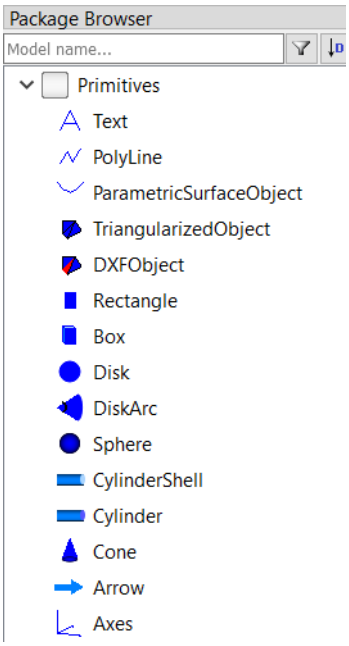
Copy Data will copy 3D plots of a visualizer window to the clipboard, for further use in for example MS Excel. The data is copied in matrix form, x, y, and z. If you have more than one 3D plot in the window, the matrices are named x1,y1,z1, x2,y2,z2 etc.

Change Projection Type will toggle between orthogonal projection (angles are preserved) and perspective projection.

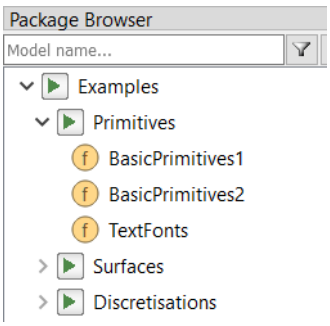
Reset will reset the objects in the window.

14.3 Basic primitives

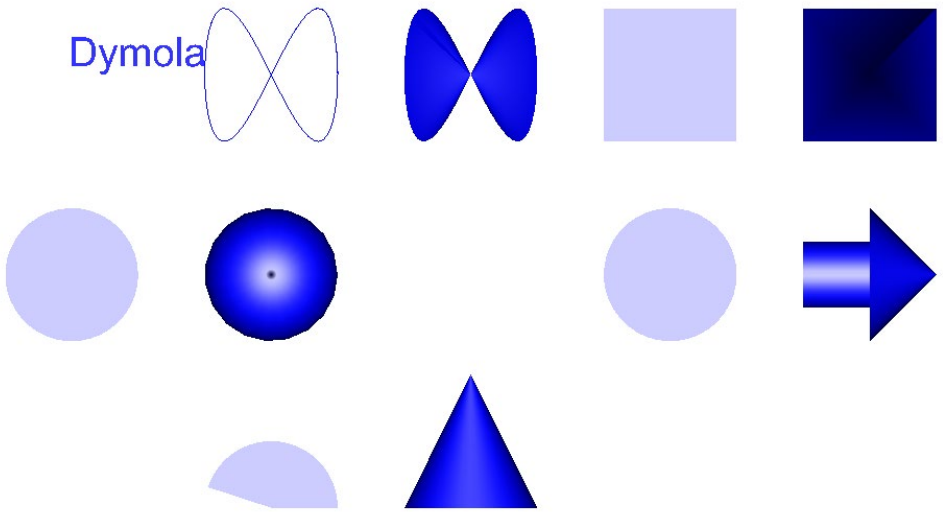
The basic predefined primitives included in Plot3D are presented in the figure below.



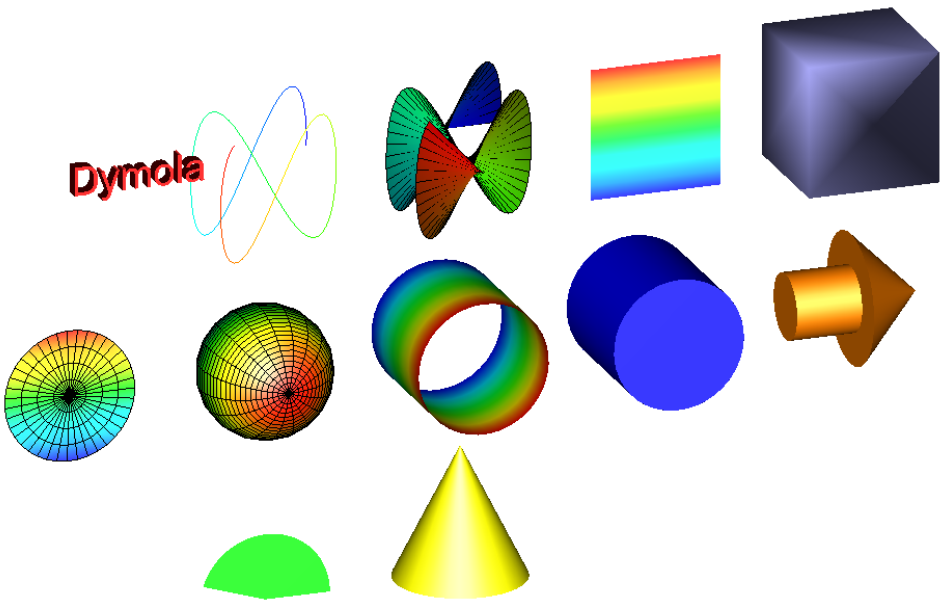
In the package Plot3D.Examples.Primitives, the functions BasicPrimitives1 and BasicPrimitives2 produce 3D scenes with the primitives.



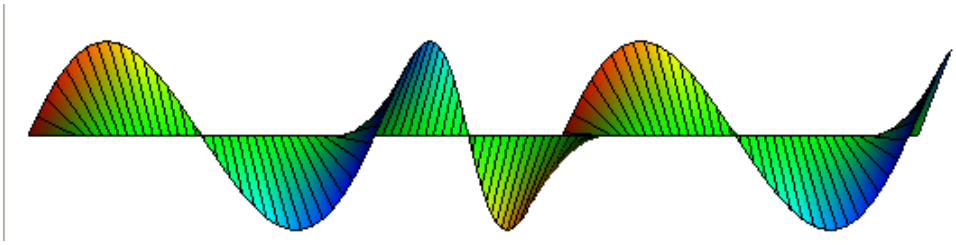
Right-click on BasicPrimitives1 and then **Call Function...**. Then, press **Execute**. The resulting 3D scene is the following.



Notice that the cylinder shell has no thickness. BasicPrimitives2 is another example showing some of the features of Visualize 3D. Repeat for BasicPrimitives2 as before to get the following 3D scene.



In particular, the third curve at the top line is a Lissajous curve, typically used in electronics and electrotechnique to find frequency and phase of an unknown sine curve, using a known one as reference. If we observe now this curve along its z-axis, the result is the following.



The dialog windows of all primitives are very similar. Each one of them have inherent fields, for instance, Plot3D.Primitives.Text has a String field called textString. In this case, the label we want to render. The primitive Plot3D.Primtives.Axes is a very particular one, since it produces a reference coordinate system. We will use it in the next section.

14.4 Surface Plots

Other important feature of Plot3D is the easy user interface and the inclusion of high level help functions that will render surfaces, contour lines, water fall plots and bar graphs from matrix data. In the following, the notation we use is as follows

1. The matrices x,y,z describing a parametric surface of the form $x=f(t,s), y=g(t,s), z=h(t,s)$.
2. The matrices nx,ny,nz describing a vector field (η_x,η_y,η_z) on the point (x,y,z).

14.4.1 Using the generateMeshGrid function to generate parametric surface matrices

To help generate the parametric surface matrices x, y, and z, you can use the utility function `Plot3D.Utilities.generateMeshGrid`. For example, consider five values of the independent variable `xvec={1.0, 2.3, 3.5, 5.2, 7.0}`, three values of the other independent variable `yvec={0.4, 1.4, 3.4}`, and the 15 corresponding dependent values `zmat=[2.7, 6.0, 5.5; 7.0, 8.1, 10.7; 10.2, 11.7, 13.9; 18.2, 20.3, 22.1; 13.4, 16.3, 17.0]`. Increased row index in the `zmat` matrix corresponds to increased index in `xvec` and similarly increased column index in `zmat` corresponds to increased index in `yvec`. (To perform a test, you can type in the variables in the command input line of the command window.)

Now apply the call (to test, you can type it in the command input line of the command window):

```
(x, y, z)=Plot3D.Utilities.generateMeshGrid(xvec, yvec, zmat);
```

The result of the call is three matrices. To see the matrix `x`, type `x` and press Enter. You can display `y` and `z` the same way:

```
x
=
[1.0 1.0, 1.0;
2.3, 2.3, 2.3;
3.5, 3.5, 3.5;
5.2, 5.2, 5.2;
7.0, 7.0, 7.0]

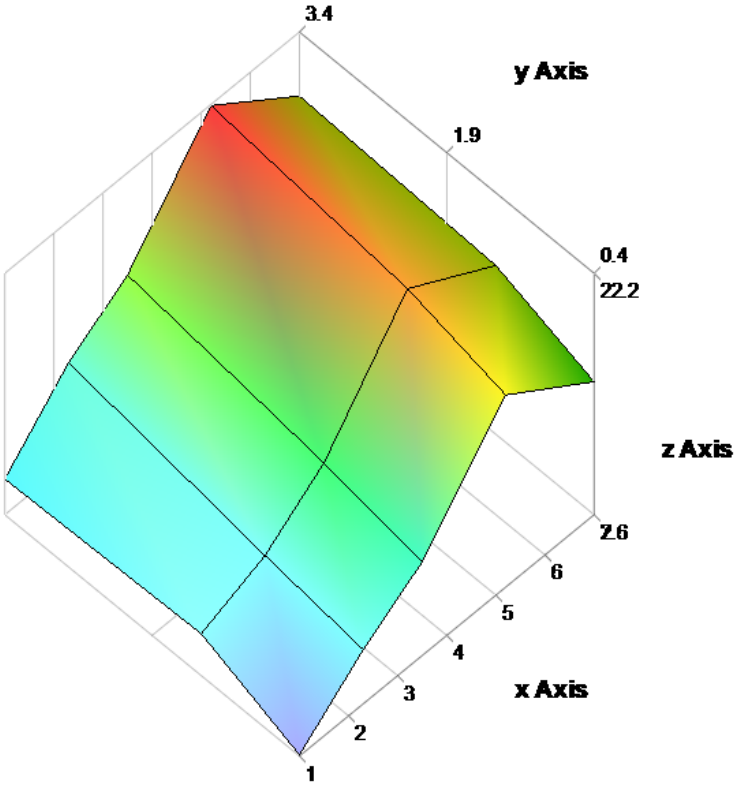
y
=
[0.4, 1.4, 3.4;
0.4, 1.4, 3.4;
0.4, 1.4, 3.4;
0.4, 1.4, 3.4;
0.4, 1.4, 3.4]

z
=
[2.7, 6.0, 5.5;
7.0, 8.1, 10.7;
10.2, 11.7, 13.9;
18.2, 20.3, 22.1;
13.4, 16.3, 17.0]
```

To create the surface plot, apply the call:

```
Plot3D.plotSurface({Plot3D.Records.ParametricSurfaceData(plotData=
Plot3D.Records.PlotData(x=x, y=y, z=z))});
```

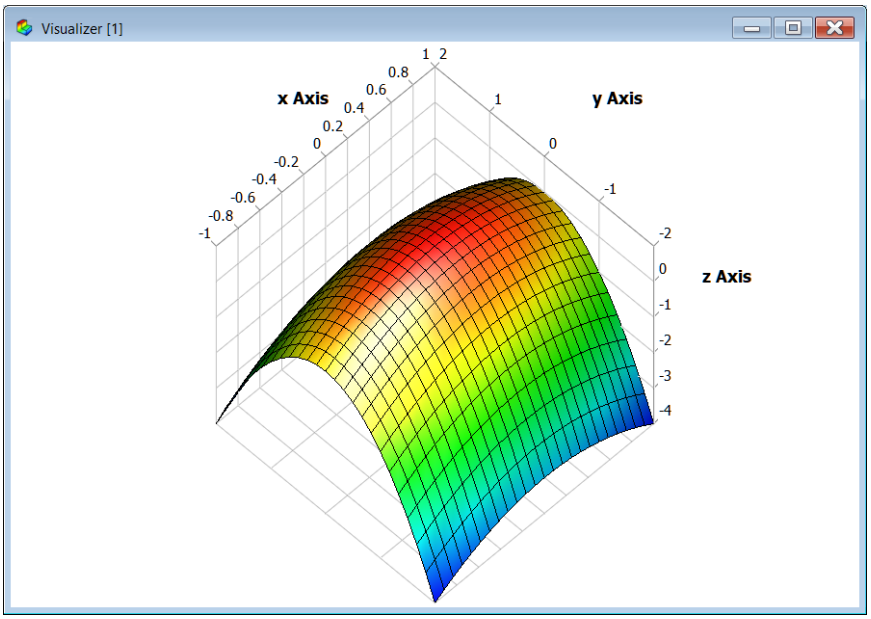
The result is:


3.4
y Axis
1.9
0.4
22.2
z Axis
2.6
6
5
4
3
x Axis
2
1

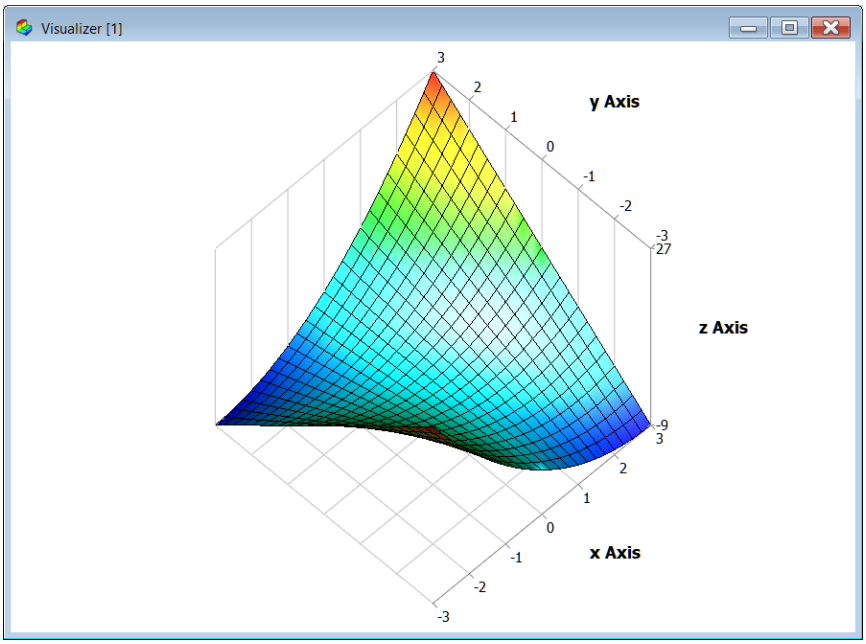
14.4.2 Three examples using the SurfaceDemo

We will consider three test cases with their respective plots:

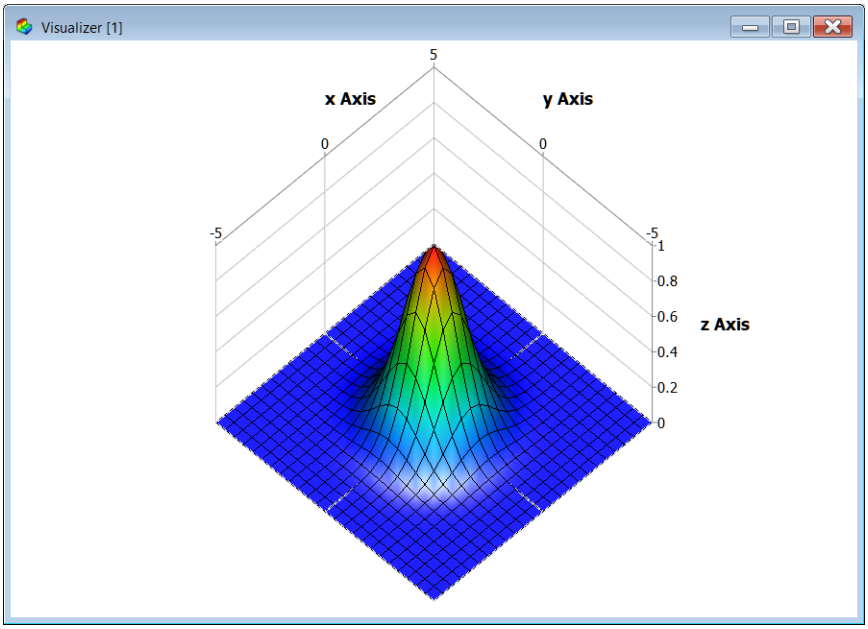
- parabolic function $z=1-x^2-y^2$ on the interval $[-1,1]\times[-2,2]$



- hyperbolic function $z = x^2 + 2xy$ on the interval $[-3,3]\times[-3,3]$

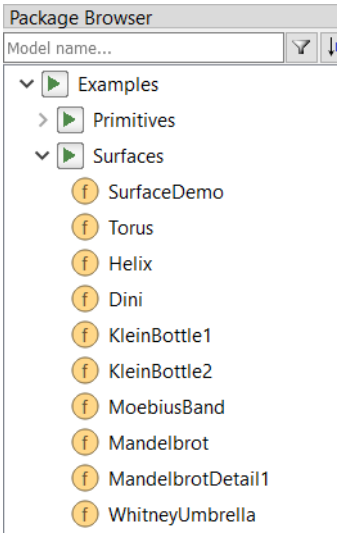


- bivariate non-normalized Gaussian distribution $z = e^{-\frac{x^2+y^2}{2}}$ on the interval $[-5,5]\times[-5,5]$.



The function Plot3D.Examples.Surfaces.SurfaceDemo runs all test cases. We will consider two of them here and just show the rest.

There are a number of more functions available, as can be seen in the package browser.

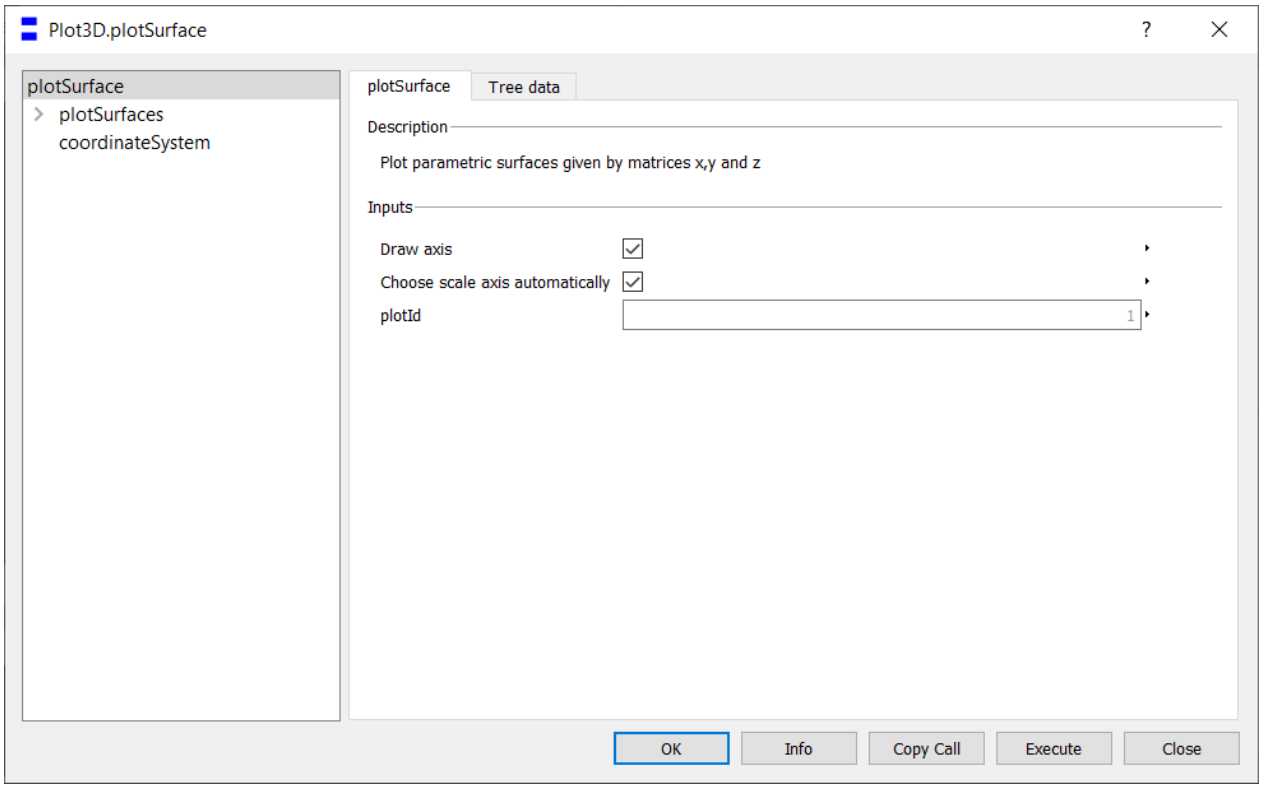


Let us plot the first test function. Enter the following command (followed by Return) in the command input line to create the matrices x,y,z,nx,ny and nz

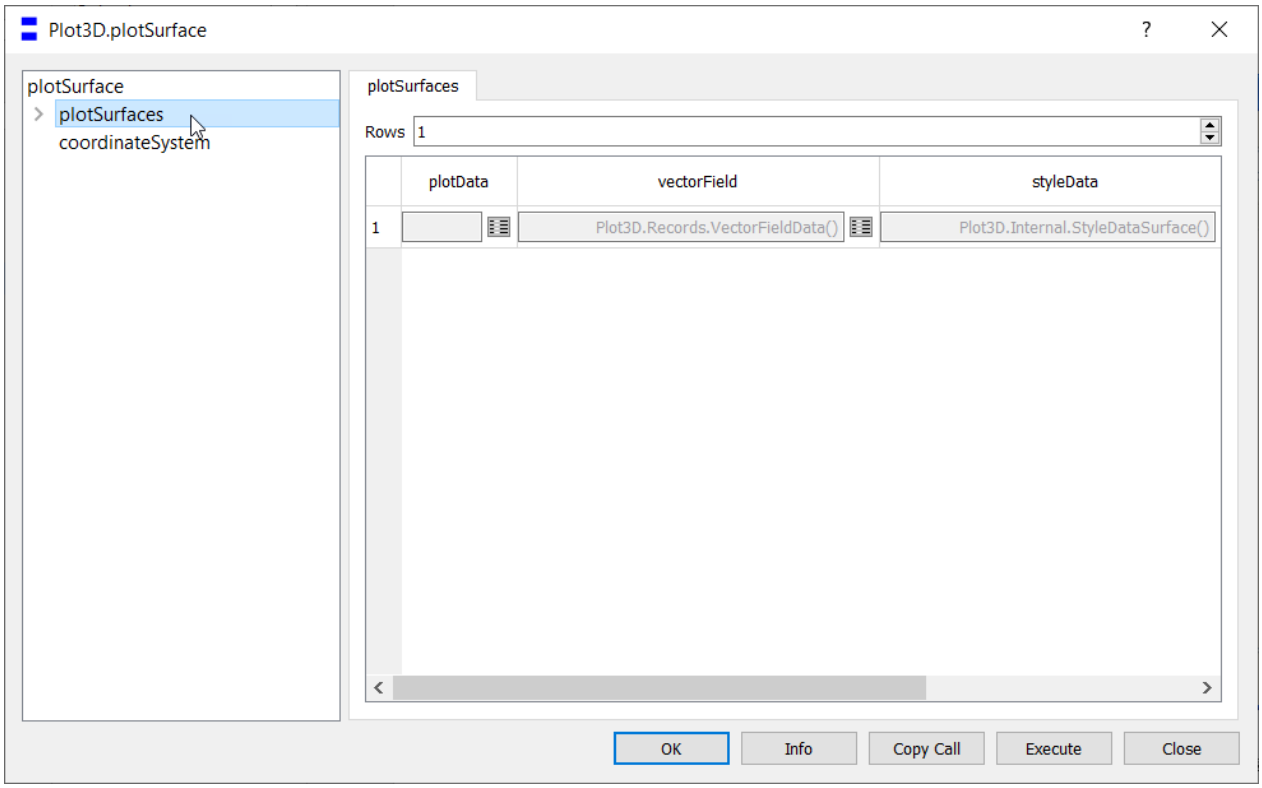
```
(x,y,z,nx,ny,nz):=Plot3D.Utilities.SurfaceTest1(25);
```

(You can get the command window and command input line displayed in the **Grahics** tab without having to go to the **Simulation** tab, by using the command **Windows > Docked Windows > Commands**.)

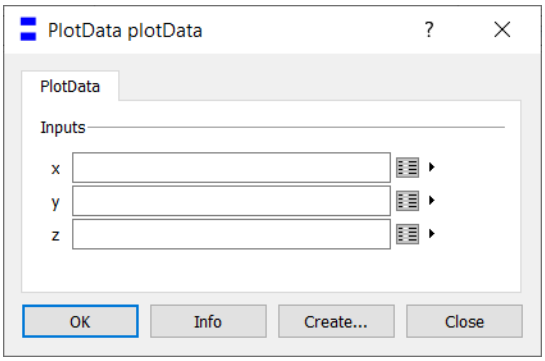
Now, right-click on the function Plot3D.plotSurface. Select **Call Function...** . The following dialog window appears



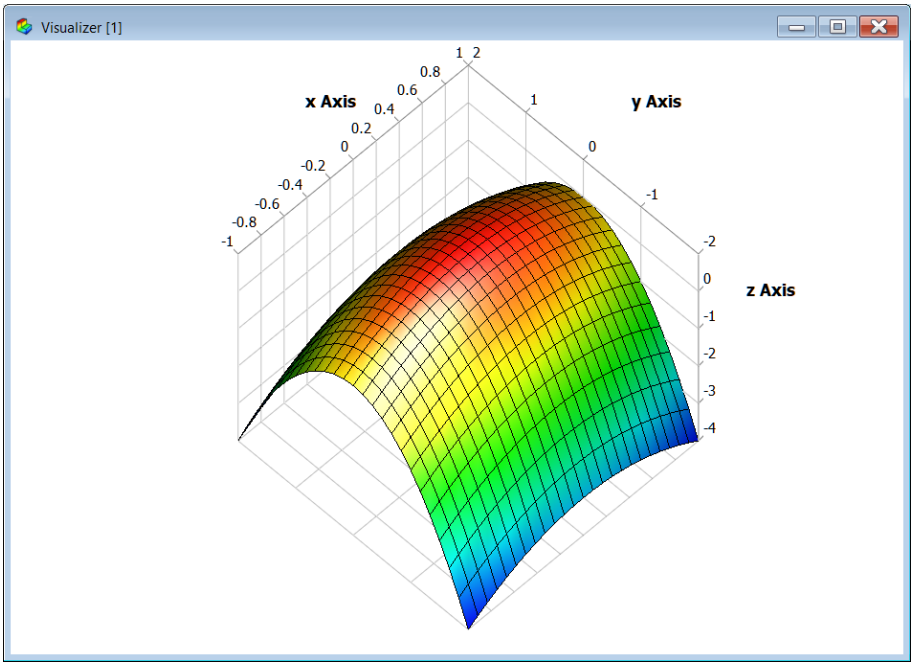
In this dialog we can set whether we want the axes automatically constructed or not. Furthermore, we can indicate a Visualize window number in “plotId”. Click now on **plotSurfaces** in the tree to the left. The following dialog pops



To set the parametric surface matrices x, y and z we click on the edit icon of "plotData". The following window pops



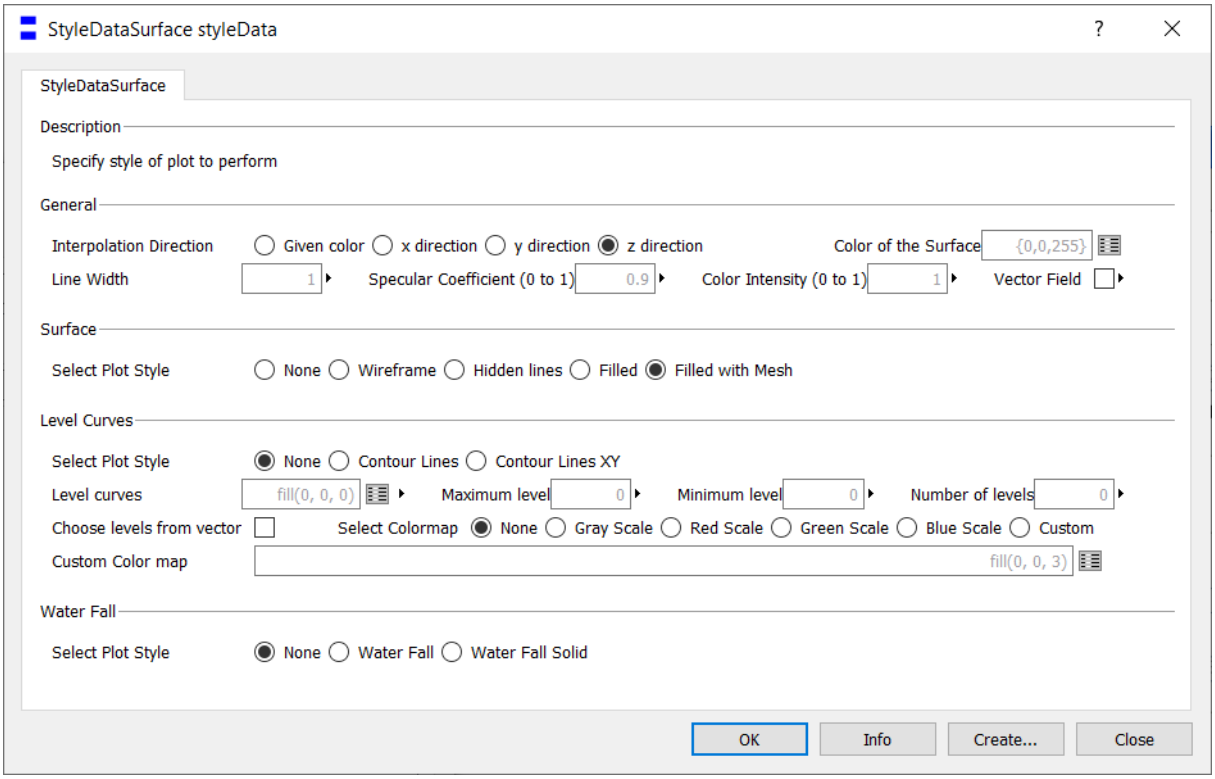
The only information needed to create a plot is to fill in the matrices. We write x, y and z in their corresponding places and click **OK**. Then, back in the main Dialog, we click on **Execute** and obtain the following plot:



This is the default plot style (Filled with Mesh), and with default names for the X, Y and Z axes.

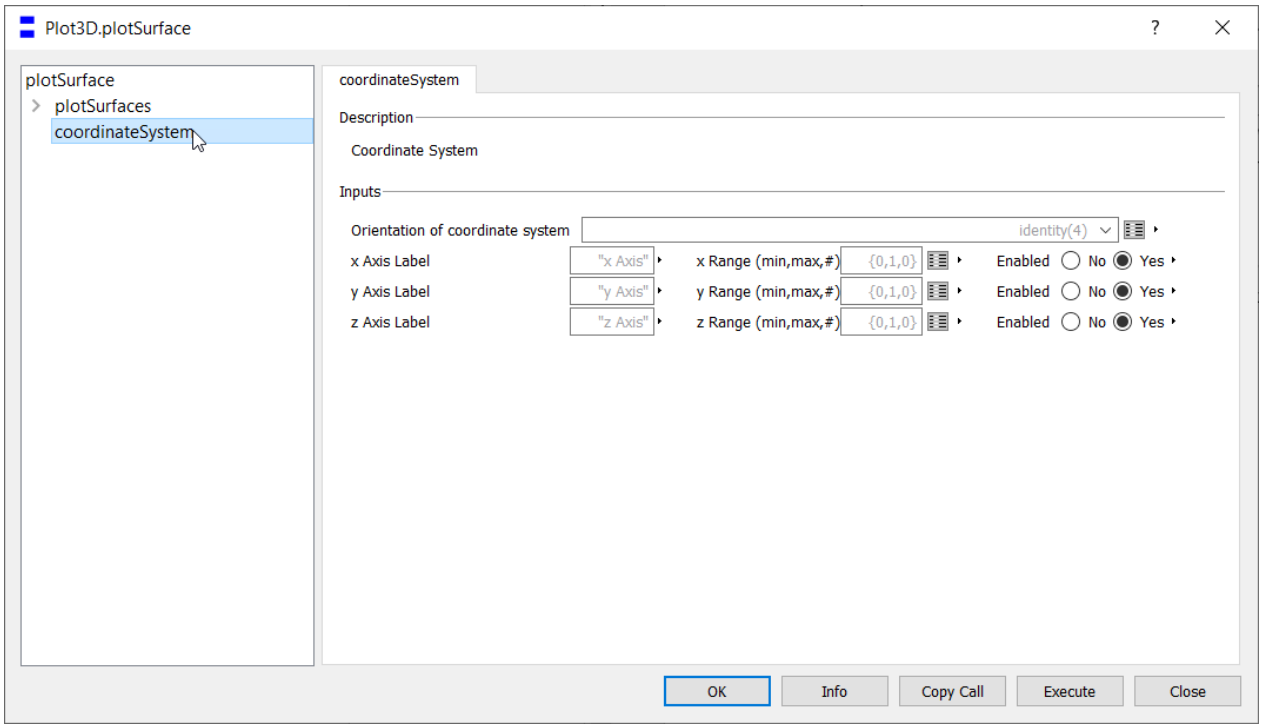
Note that you can create the plots corresponding to the second and the third test case the same way, if you first create the corresponding matrices; for example, to create the matrices of the second test case (hyperbolic function), use, the same way as for the first test case: `(x,y,z,nx,ny,nz):=Plot3D.Utilities.SurfaceTest1(25);`

Now, if we want to change the style of the plots, the data has to be filled in the “styleData” field, using its **Edit** icon. The dialog follows



We observe the different styleData alternatives. We can combine independently four groups of data: Surface (Wireframe, Hidden Lines, Filled and Filled with Mesh), Level Curves (Contour Lines and Contour lines XY), Water Fall (Normal and Solid disks) and Vector Field (check box in General group).

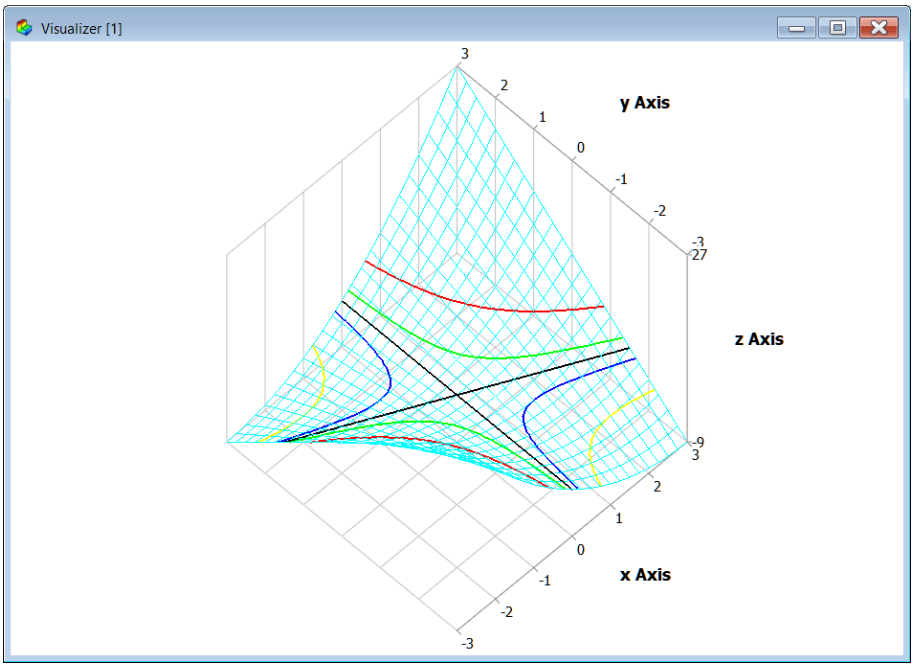
To change the axes properties, we click on “coordinateSystem” in the tree to the left in the Plot3D.plotSurface window. The following dialog window pops up:



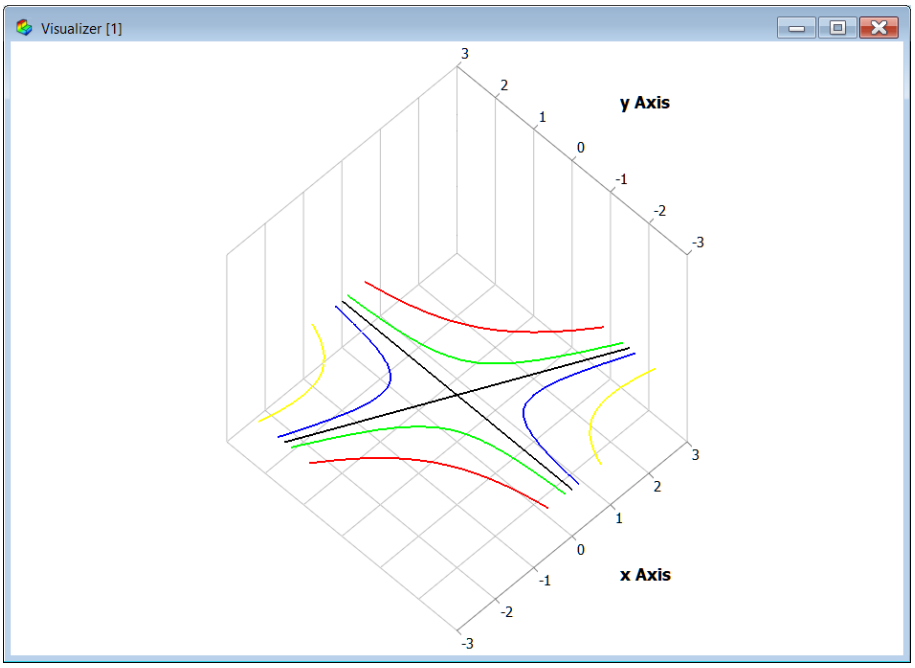
We observe here the fields “Axis label”, “Range” and “Enabled” for X, Y and Z axis.

Using the functions plotPoints, plotLines, plotStem, plotSurface and plotBarGraph follows the same lines, with particular variations.

We want to emphasize the combination of contour plots with Wireframe. This combination is particularly interesting to show features of a surface. For instance, the contour lines of the hyperbolic function $z = x^2 + 2xy$ for $z = 0$ yield two straight lines and constitute a degenerated transition case between the hyperbolic lines in two quadrants (above of $z = 0$) and hyperbolic lines in the other two quadrants (below of $z = 0$). The resulting plot follows (you can get the plot by right-clicking the function surfaceDemo and select the demo type as **Surface with Contour lines plane chosen with custom color**).

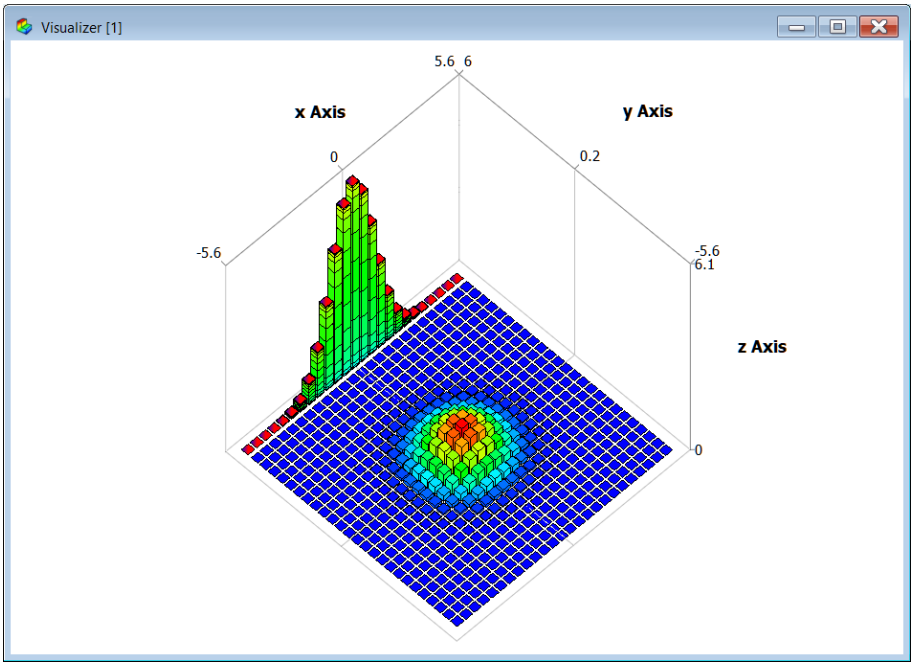


Or easily viewed, projected on the XY plane without Z axis ticks (you can get the plot by right-clicking the function surfaceDemo and select the demo type as **Contour lines projected at XY plane chosen with custom color**).

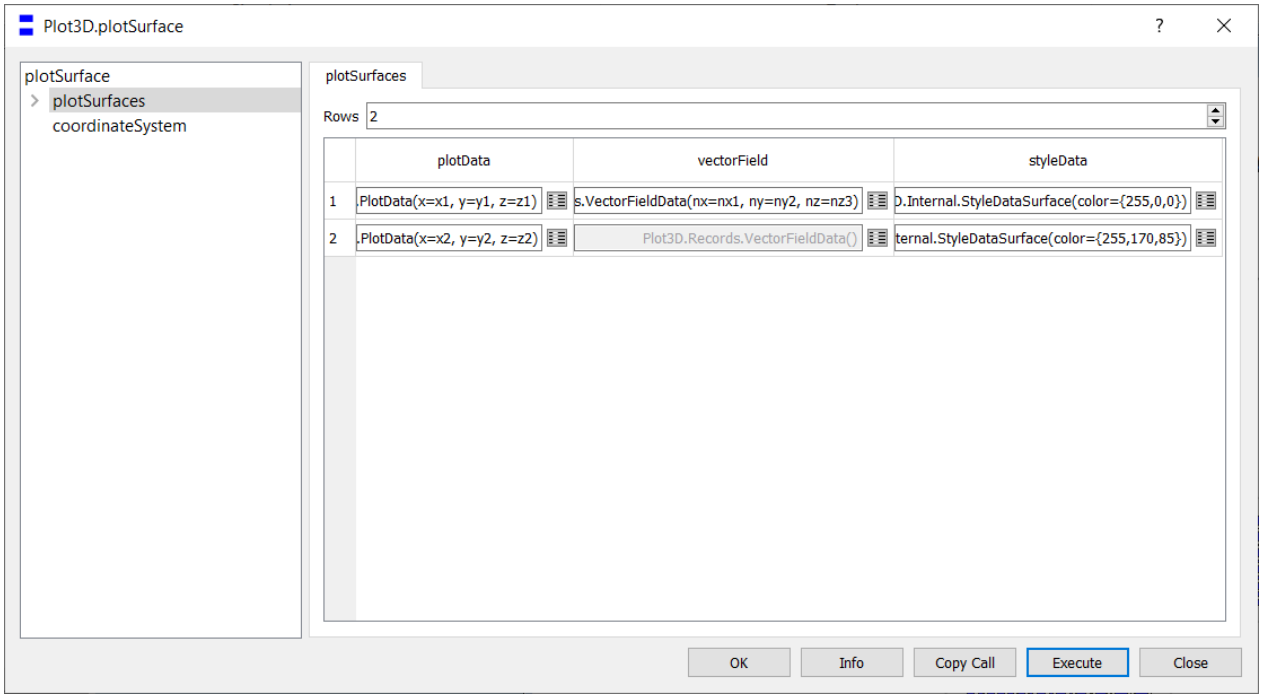


The black lines are the asymptotes of both sets of hyperbolic contour lines (red means $z = 4$, green $z = 1$, blue $z = -1$ and yellow $z = 4$).

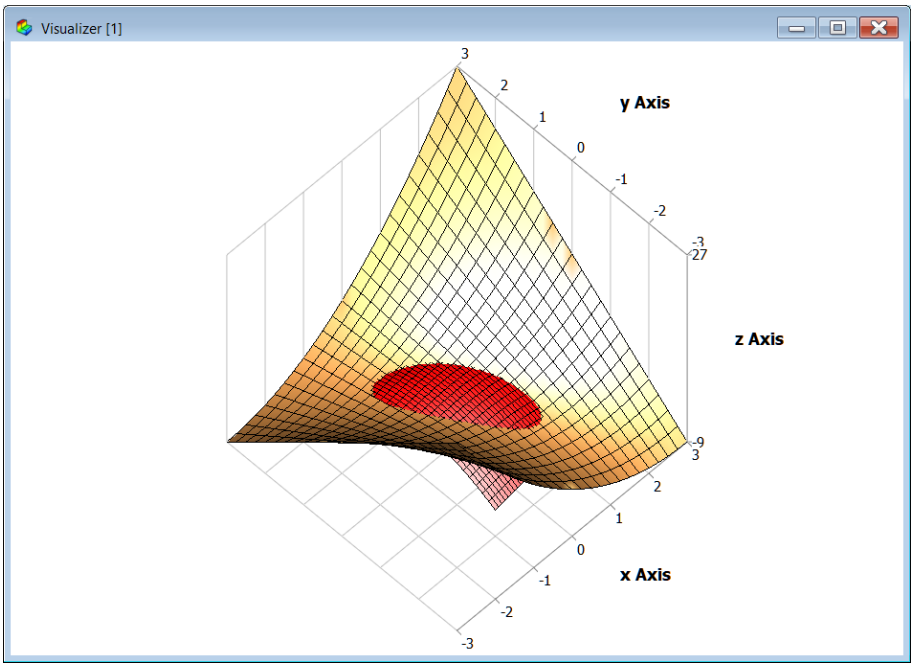
Other combinations can be useful to explain features too. For instance, when considering the Gaussian bivariate probability distribution. If we integrate one of the variables (let us integrate the y variable in this case) the resulting univariate function is also a Gaussian distributed variable. Combining the "Rectangle on Top" with the "Rectangle" plots of Plot3D.plotBarGraph function we can illustrate just that. The result follows (you can get the plot by right-clicking the function surfaceDemo and select the demo type as **Surface with Contour lines plane chosen with custom color**).



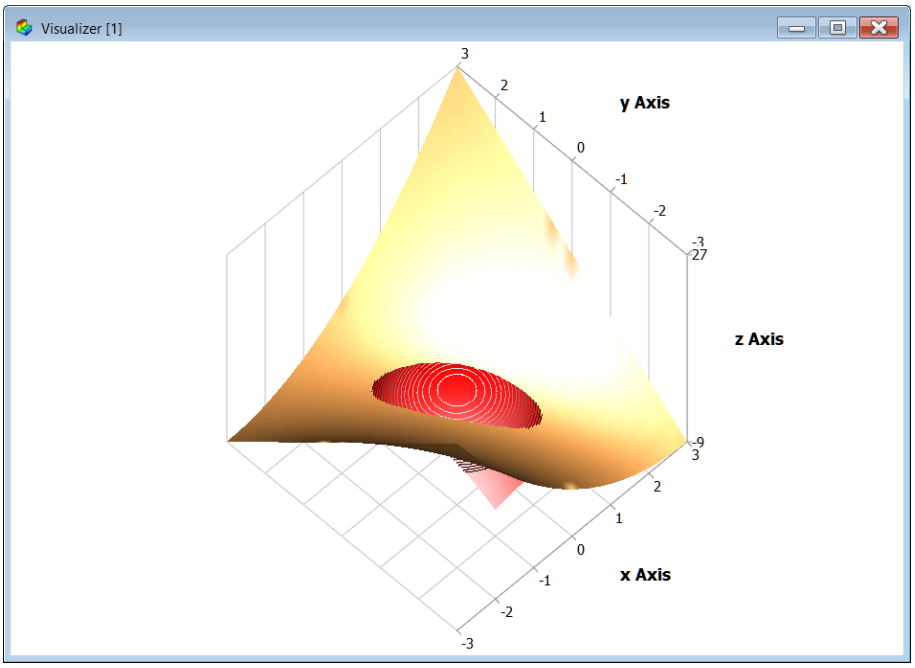
The intersection of surfaces using Plot3D.plotSurface is also possible. The only thing we have to do is to increment the number of elements to plot in the dialog. We can also set color and style to identify easily the functions and delimit the intersection area.



One possibility is to have different colors for the surfaces. The intersection of the parabolic surface and the hyperbolic surface with different colors will look as follows (you can get the plot by right-clicking the function surfaceDemo and select the demo type as **Surface Intersection 1**).

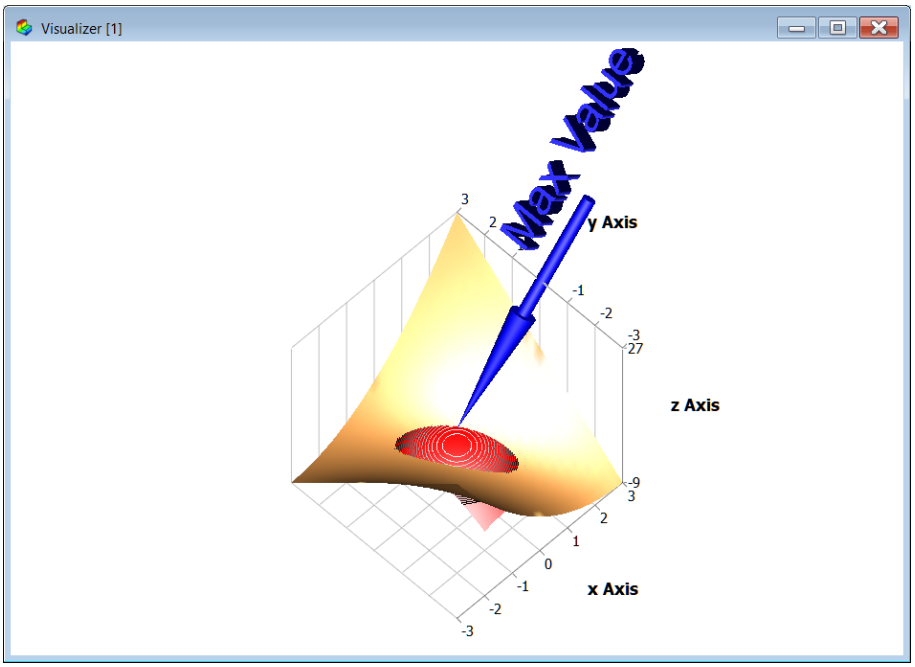


Combining different styles, we can obtain the following graph (you can get the plot by right-clicking the function surfaceDemo and select the demo type as **Surface with Contour lines plane chosen with custom color**).

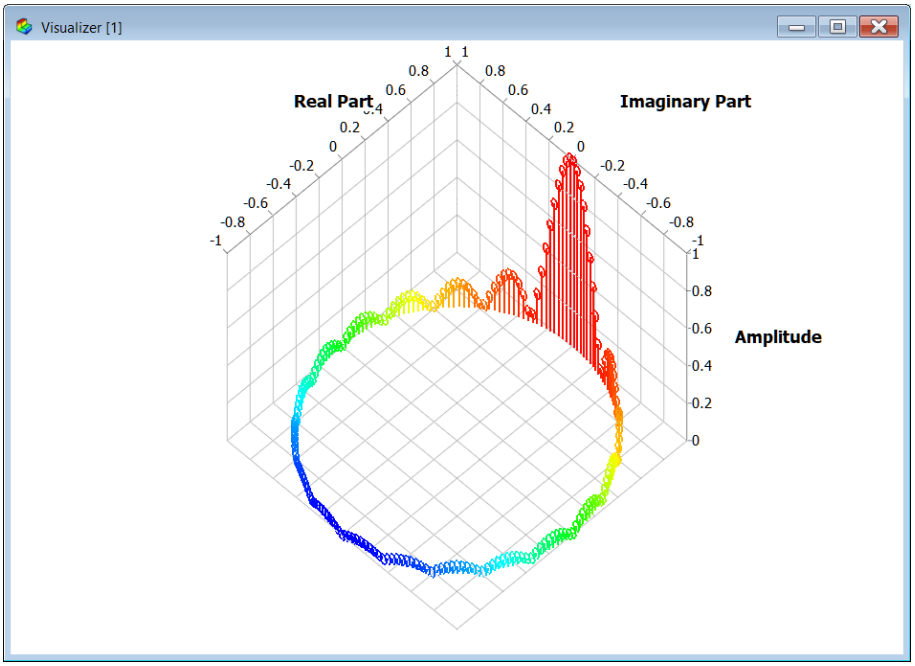


The contour lines in the parabolic surface (red) are used to illustrate that the intersection does not happen on a plane. The black color corresponds to the level z=-1.5 and the white color corresponds to the level z=1.

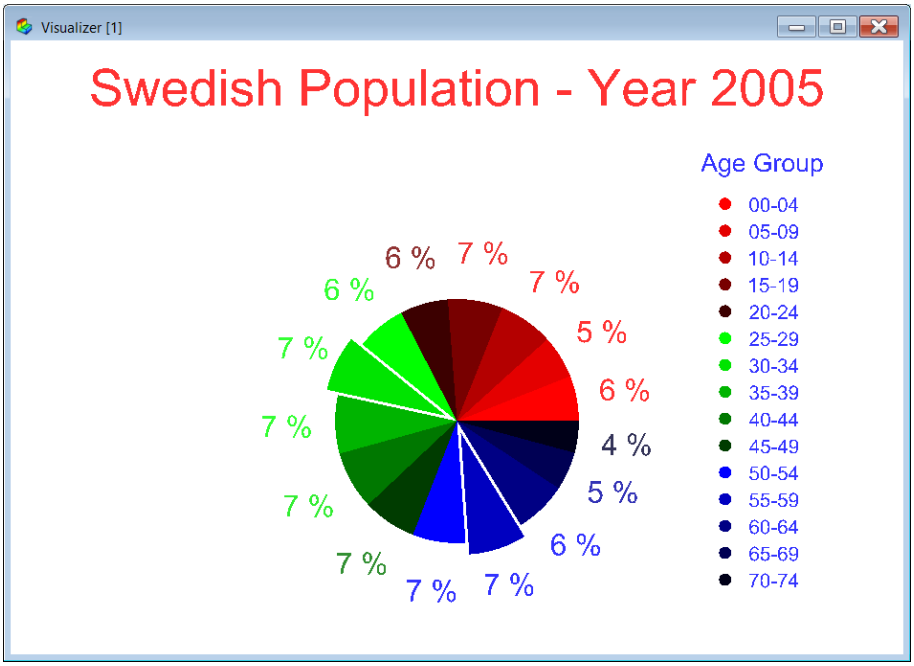
We can add a pointer to show where the maximum of the red surface occurs. Using the function Plot3D.insertPointer directly on the last image we can add text and an arrow. The resulting figure follows (you can get the plot by right-clicking the function surfaceDemo and select the demo type as **Pointing to red surface´s Maximum Value**).



To plot discrete data, the alternative is to use plotStem function. This function considers each point by itself and puts a triangle, square or circle at the data point and adds a line from the point to the plane XY. The following example shows the amplitude or absolute value of the discrete Fourier Transform of a pulse. Putting the values in the unit circle of the complex plane relates the Z-transform to the discrete Fourier transform. The color is interpolated in the x direction (you can get the plot by right-clicking the function surfaceDemo and select the demo type as **Stem test**).



Other alternative that Plot3D provides is to make pie charts. Statistiska Centralbyrån (Central statistics office) in Sweden reports the following population distribution by age in 2005. Two age groups are separated (30-34 and 55-59) to distinguish them (you can get the plot by right-clicking the function surfaceDemo and select the demo type as **Swedish Population Distribution (Pie Chart)**).


Visualizer [1]
Swedish Population - Year 2005
Age Group
00-04
05-09
10-14
15-19
20-24
25-29
30-34
35-39
40-44
45-49
50-54
55-59
60-64
65-69
70-74
6 %
7 %
7 %
5 %
6 %
4 %
5 %
6 %
7 %
7 %
7 %
7 %
7 %
6 %
7 %

15 USER-DEFINED GUI

15 User-defined GUI

This chapter has two main sections. The first describes building user-defined input dialogs for models and functions that corresponds to records and arrays. The second describes the extendable user interface of Dymola, that is, extending the user interface by introducing own menus and toolbars from which Modelica functions can be called, and the possibility to define packages with users own collection of favorite models.

15.1 Building user-defined dialogs

In addition to primitive data types, Real, Integer, Boolean and String and from them derived types, Modelica has records and arrays. We will in this section show how to build graphical user interfaces for models and functions that correspond to these data structuring mechanisms, by using annotations

15.1.1 Ways of working with annotations

When it comes to working with annotations for variables, there are two ways of working:

- By direct typing in the Modelica Text layer.
- By using the Declare variable dialog.

Depending on what should be done, one way might be more convenient to use than the other. As an example consider the case when variables should be divided in tabs and groups in a

parameter dialog. The Declare variable dialog gives, in the Annotations tab, a very convenient way to use input fields to handle this.

On the other hand, all annotations are not available as settings in the Declare variable dialog. In that case it might be as easy to use the Modelica Text layer directly, even though it of course still is possible to work with the Declare variable dialog (typing in the annotations; the full Modelica text for that variable is visible and editable in the dialog).

Certain annotations are not at all connected to variables; in that case there is no alternative than the Modelica Text layer.

Please note that to see the annotations in the Modelica Text layer of the **Text** tab, right-click in the window and use the context command **Expand > Show entire text**.

How to work with the Declare variable dialog is introduced in section "Tabs and Groups" starting on page 1483 below.

For a full description of the Declare variable dialog, please see the chapter "Developing a model", section "Advanced model editing", sub-section "Parameters, variables and constants".

15.1.2 Records and dialogs

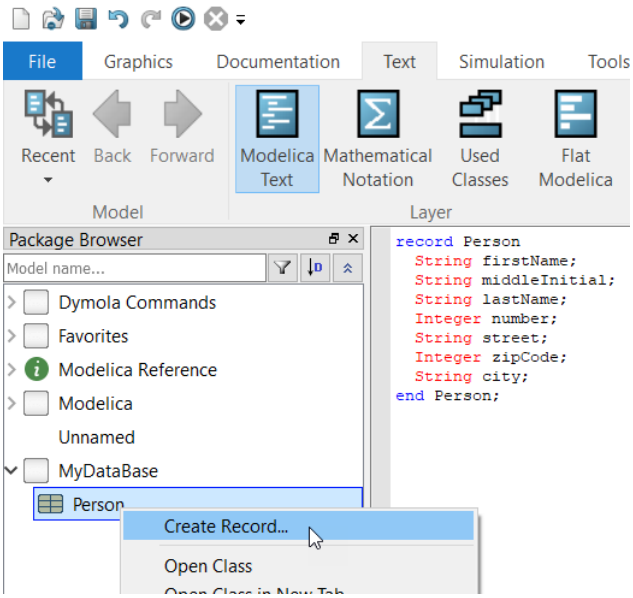
As an introductory example, we will consider making a small data base of personal data. Assume that each person is described by the following information:

```
record Person
  String firstName;
  String middleInitial;
  String lastName;
  Integer number;
  String street;
  Integer zipCode;
  String city;
end Person;
```

(The record can be declared by directly typing in the Modelica Text layer of the **Text** tab, or by using the command **File > New > Record**. The variables in it can be typed in directly or can be added using **Text > Insert > New Variable…**.)

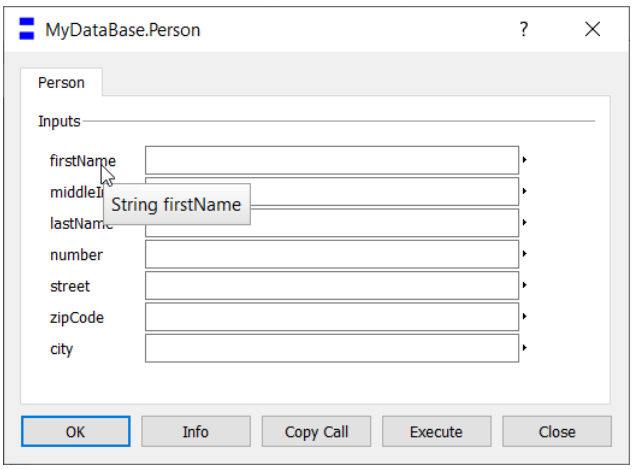
The corresponding automatically constructed GUI dialog for entering data (when the record is declared) is accessed by right-clicking on the record in the package browser, taking up the context menu and selecting **Create Record…**:

Accessing the menu for entering data.



(In this example the record has been created in a package MyDataBase.) The menu for entering data that pops looks as follows:

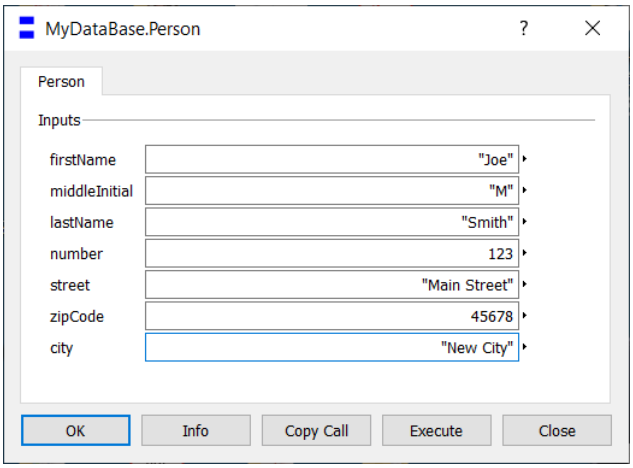
Automatically constructed dialog.



The tool tip shows the data type of the input field.

Entering the following data:

Filled-in dialog.

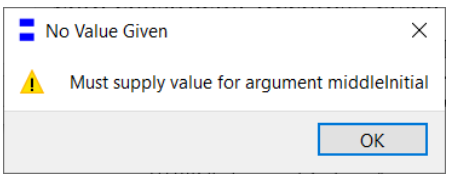


and pressing the **OK** or **Execute** buttons gives the result in the log window as a call to the record constructor Records.Person with the name-value pairs for the entered data.

```
= MyDataBase.Person(
  firstName = "Joe",
  middleInitial = "M",
  lastName = "Smith",
  number = 123,
  street = "Main Street",
  zipCode = 45678,
  city = "New City"
)
```

If we would not fill in any value for middleInitial, the following error message would be generated:

Missing data error.



To avoid having to give such data, a default value can be given in the declaration:

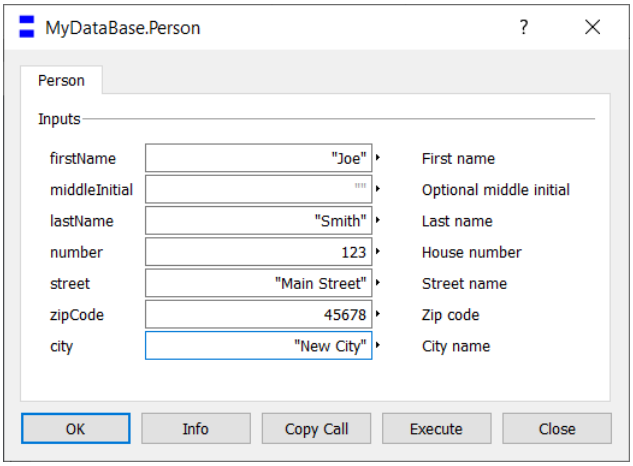
```
String middleInitial = "";
```

Modelica allows you to add description strings to all variables. We alter the record to:

```
record Person
  String firstName "First name";
  String middleInitial="" "Optional middle initial";
  String lastName "Last name";
  Integer number "House number";
  String street "Street name";
  Integer zipCode "Zip code";
  String city "City name";
end Person;
```

These are used to annotate the dialog as shown below.

Dialog with description strings.

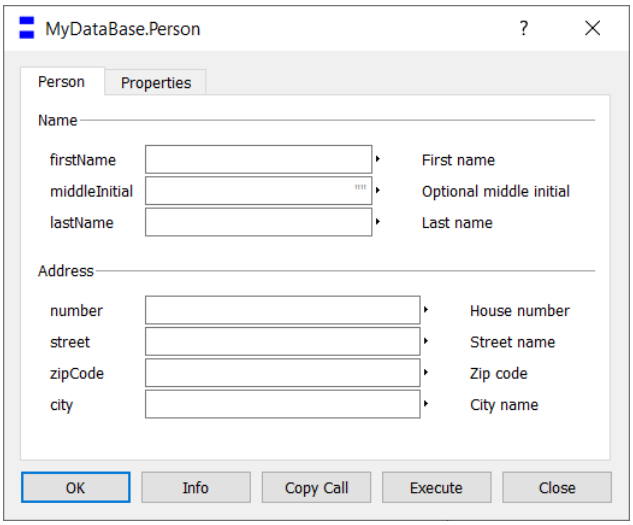


Tabs and Groups

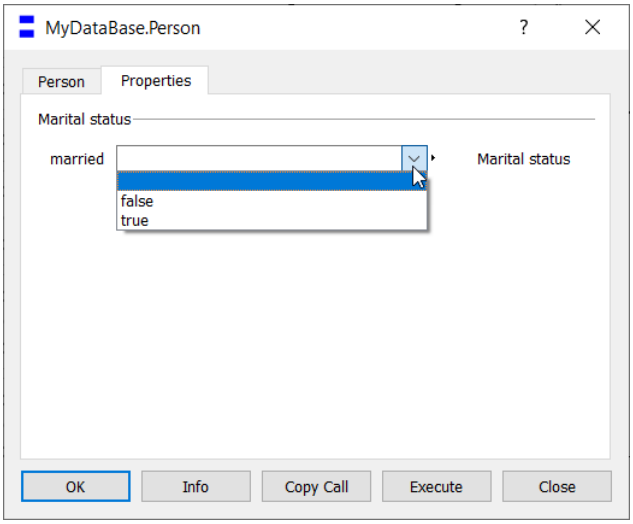
It is possible to annotate input fields in various ways in order to simplify for the user to enter data.

It is, for example, possible to group record fields together and introduce tabs in the dialog by means of annotations. This is very conveniently done for each variable using the Declare Variable dialog, please see below.

Dialog with groups and tabs.



These changes are made by adding the following annotations and extending the record with field married. Note that the field married is put in a tab "Properties":




```
record Person
  String firstName "First name"
    annotation (Dialog(group="Name"));
  String middleInitial="" "Optional middle initial"
    annotation (Dialog(group="Name"));
  String lastName "Last name"
    annotation (Dialog(group="Name"));
  Integer number "House number"
    annotation (Dialog(group="Address"));
```

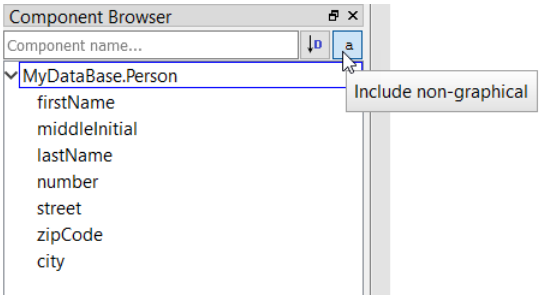
```
    String street "Street name"
      annotation (Dialog(group="Address"));
    Integer zipCode "Zip code"
      annotation (Dialog(group="Address"));
    String city "City name"
      annotation (Dialog(group="Address"));
    Boolean married "Marital status"
      annotation (Dialog(tab="Properties", group="Marital
        status"));
  end Person;
```

Note that for the Boolean field married the combobox with choices false and true appear automatically; see above image.

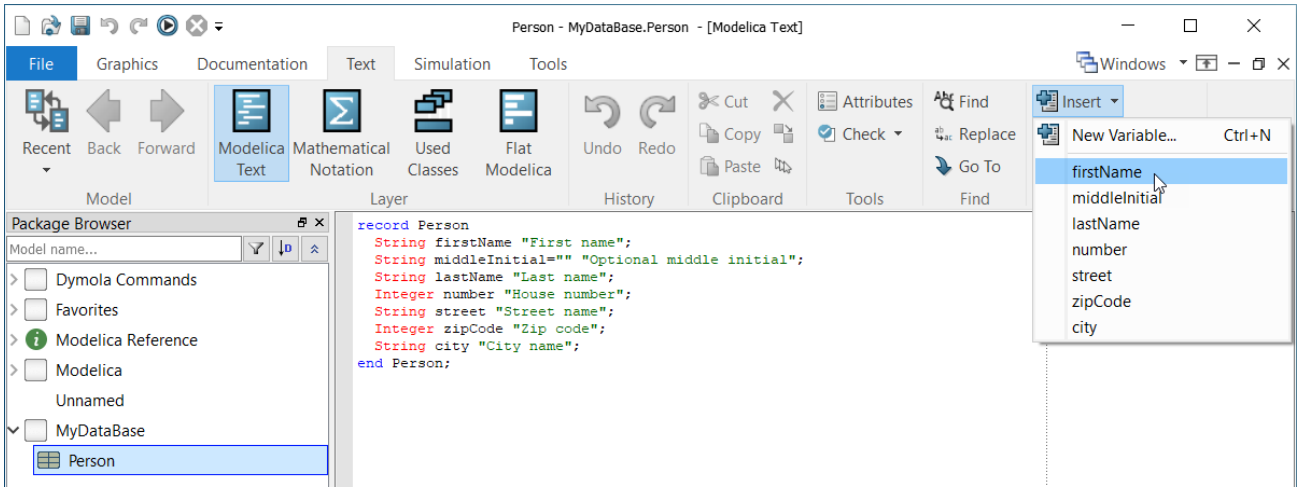
The changes above can be done by direct typing in the Modelica Text layer, but a convenient alternative way to work with the variables (and annotations) is to use the Declare variable dialog.

There are two to access the dialog when a variable has been declared:

- Use the command **Text > Insert > <variableName>**.
- If non-graphical components are enabled in the component browser variables will be shown. In that case a variable can be right-clicked; selecting **Variable…** will display the Declare variable dialog. (To enable display of non-graphical objects, click the button **Include non-graphical** as has been done in the image below):

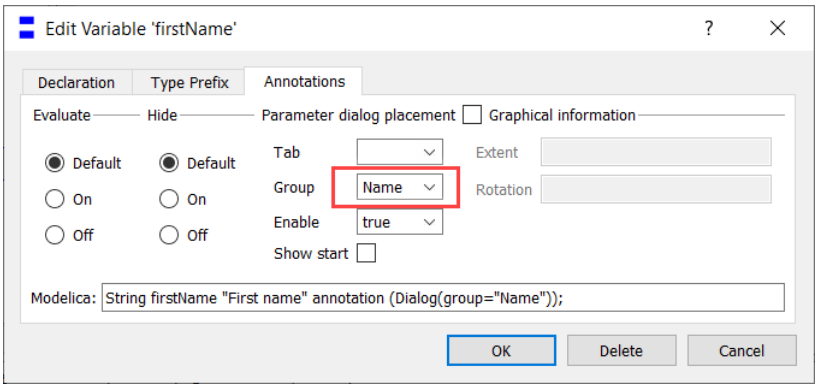


Say we want to add the annotations of firstName using the first alternative above. By using the command **Text > Insert** we get:



(It is also possible to right-click in the Modelica Text layer and select **Variables > firstName**.)

By popping the menu, selecting the **Annotations** tab, entering `Name` in the **Group** field we will get:



When clicking **OK**, we have created the completed the annotations for the variable firstName. In the same the way the other variables could be given annotations. New variables can be introduced using the **New variable…** alternative. This way the new variable `married` can be added, and corresponding annotations specified.

Hiding variables

A variable that should be internal (only accessible inside the model) should be declared as protected. This can be done using the **Type Prefix** tab of the Declare variable dialog, ticking **protected**. This variable will neither be seen in any parameter dialog, nor in the variable browser.

It is also possible to prevent the variable from being presented in the variable browser by declaring it as hidden using the **Annotations tab** of the Declare variable dialog (see previous

picture), ticking **On** in the Hide group. However, it will still be available in the parameter dialog. Usually the alternative to declare it as protected is better.

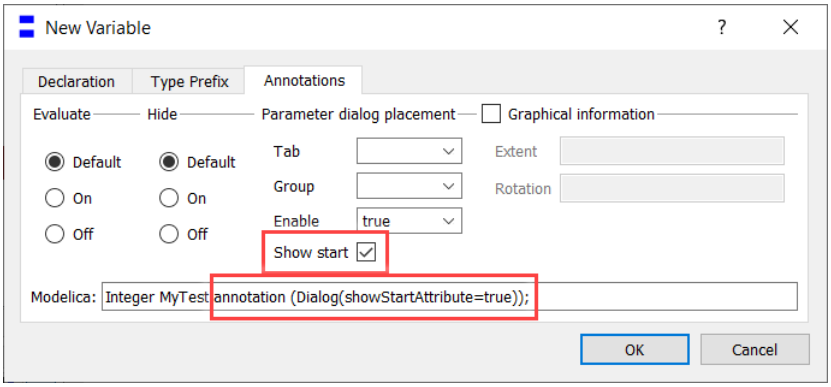
For more information about this, please see the chapter “Developing a model”, section “Advanced model editing”, sub-section “Parameters, variables and constants”.

Activation of the dialog entry for start values

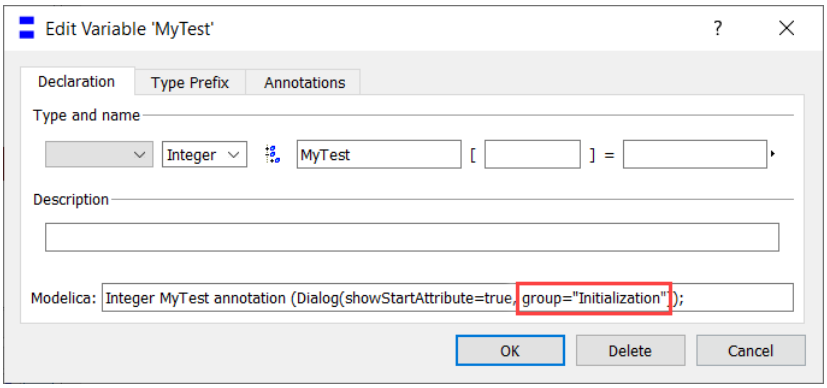
(This section is not part of the bigger example we are working with; it is put here since it is about another feature in the Annotations tab.)

The annotation for activating the dialog entry for start values can be set in the variable declaration dialog, in the **Annotations** tab. See the above figure; here it is also described how to display this tab.

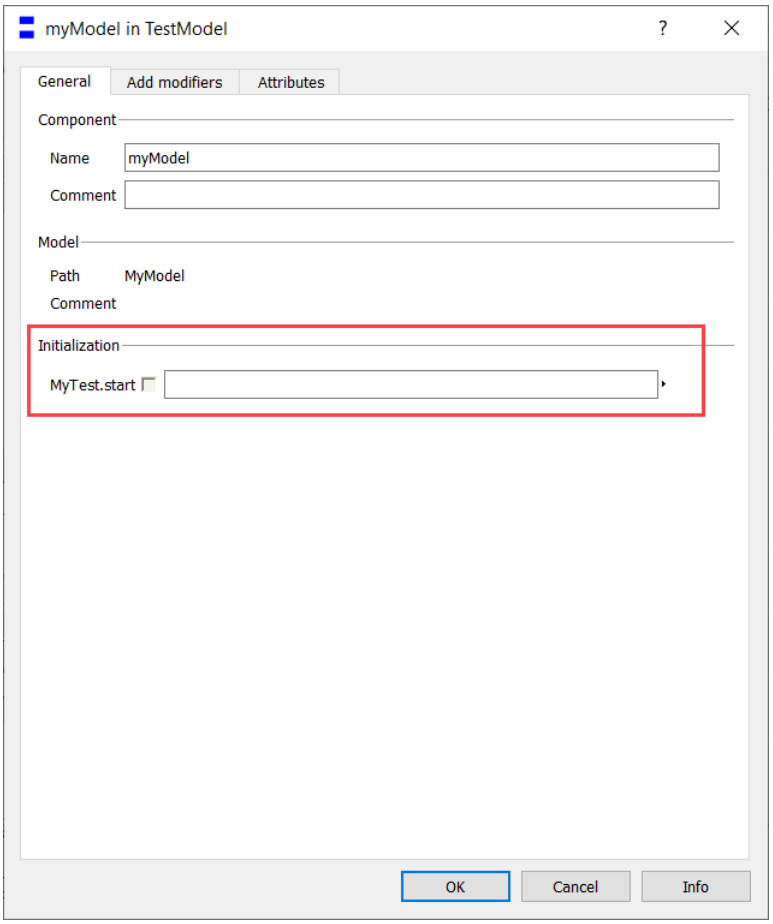
As an example, enabling this option **Show start** for an integer variable `MyTest` in the dialog gives the result:



Clicking **OK** will also add the annotation to display this start value in an “Initialization” group, taking up the menu again after having clicked **OK** will display:



The result of using the model containing this variable is, looking at the parameter dialog:



Labels and layout

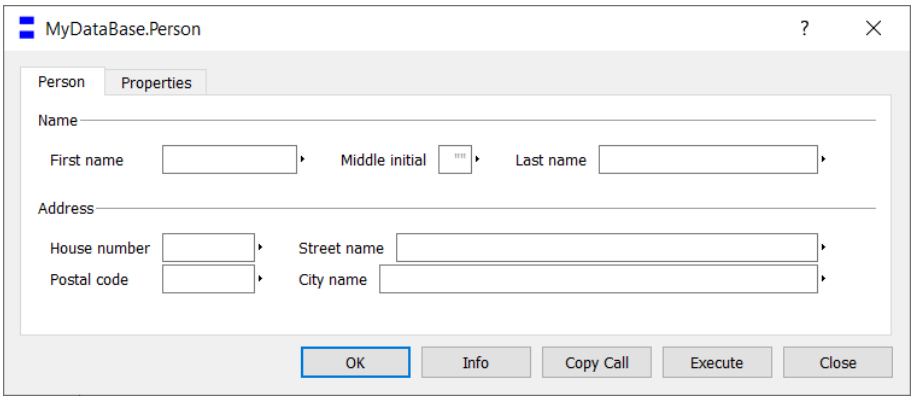
Returning to our bigger example; by annotating a field, such as firstName, with the attribute `__Dymola_joinNext=true`, the next field, middleInitial, is put on the same horizontal line as firstName.

Instead of having the variable name in front of the input field, the description string is used if the Dialog annotation: `__Dymola_descriptionLabel=true` is given. The description string is then not shown after the input field. A label with free text can be given by `__Dymola_label="free-text"`. The free text label has precedence over the description label.

The width of the inputs fields can be specified as, for example, `__Dymola_naturalWidth=10`. The width is given in the unit "en", the width of character '0'. The width can also be specified as `__Dymola_absoluteWidth=10`. The difference is that fields with `__Dymola_absoluteWidth` keep their size when the entire dialog is made wider. The fields with `__Dymola_naturalWidth` specification are made wider.

By use of these annotations we can make the dialog much nicer.

Flexible labeling and layout of input fields.



The details of the record declaration are given below:

```
record Person
  String firstName "First name"
    annotation (Dialog(group="Name", __Dymola_joinNext=true,
    __Dymola_naturalWidth=15, __Dymola_descriptionLabel=true));
  String middleInitial="" "Middle initial"
    annotation (Dialog(group="Name", __Dymola_joinNext=true,
      __Dymola_absoluteWidth=3, __Dymola_descriptionLabel =
      true));
String lastName "Last name"
    annotation (Dialog(group="Name", __Dymola_naturalWidth=25,
      __Dymola_descriptionLabel = true));

  Integer number "House number"
    annotation (Dialog(group="Address", __Dymola_joinNext=true,
      __Dymola_absoluteWidth = 10, __Dymola_descriptionLabel =
true));
  String street "Street name"
    annotation
(Dialog(group="Address",__Dymola_descriptionLabel
      = true));
  Integer zipCode "Zip code or postal code"
    annotation (Dialog(group="Address", __Dymola_joinNext=true,
      __Dymola_absoluteWidth = 10, __Dymola_descriptionLabel =
      true, __Dymola_label="Postal code"));
    String city "City name"
      annotation (Dialog(group="Address",
        __Dymola_descriptionLabel = true));

    Boolean married "Marital status"
      annotation (Dialog(tab="Properties",
        group="Marital status", __Dymola_absoluteWidth=10));
  end Person;
```

The annotations used above are not available as settings in the Declare variable dialog; it can however be entered in that dialog.

Alternative forms for input fields

Using combo boxes

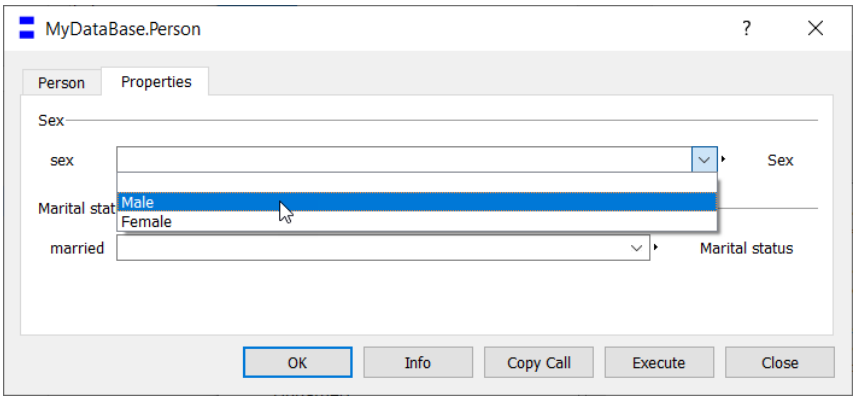
Sometimes there is a set of frequent input values (enumerations) and in addition free text should be possible. For such cases, it is possible to add a combo box for the frequent choices. This would, for example, be convenient for a sex field:

```
Integer sex "Sex"
  annotation (Dialog(tab="Properties", group="Sex"),
    choices(choice=1 "Male", choice=2 "Female"));
```

Associated with each value (1, 2), it's possible to give a description string ("Male", "Female").

The Properties tab has the following layout after this addition.

Input field with combo box.

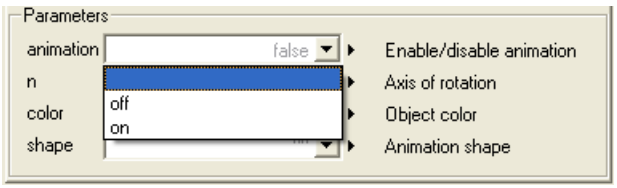


It should be noted that it's possible to enter any value without using the pull-down menu. This enables the use of expressions, for example.

Some additional examples

It is possible to annotate parameters or parameter types in order that it's possible to make a selection from a set of values from a pull down menu. For example, setting a parameter true or false can be made by selecting on or off as shown below.

Drop-down menu in parameter dialog.

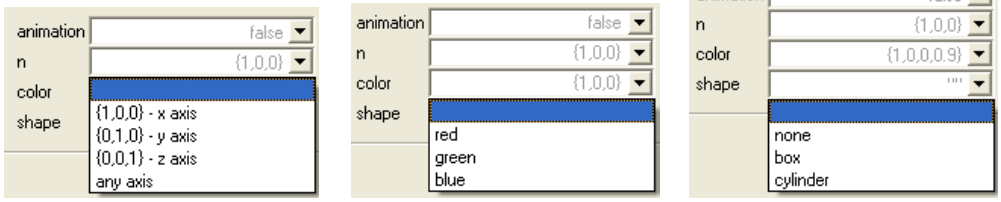


The needed declarations for this appearance are:

```
type OnOff = Boolean annotation (choices(
  choice=false "off",
  choice=true "on"));
parameter OnOff animation=false
```

```
  "Enable/disable animation";
```

The following examples show similar choices from a set of predefined vectors representing different common directional axes or commonly used colors. In the example to the right, a selection has been among a set of strings.



The corresponding declarations are:

```
type Axis = Real[3] annotation (choices(
  choice={1,0,0} "{1,0,0} - x axis",
  choice={0,1,0} "{0,1,0} - y axis",
  choice={0,0,1} "{0,0,1} - z axis",
  choice={0,0,0} "any axis"));
parameter Axis n={1,0,0} "Axis of rotation";
type Color = Real[3] annotation (choices(
  choice={1,0,0} "red",
  choice={0,1,0} "green",
  choice={0,0,1} "blue"));
parameter Color color={1,0,0} "Object color";
type Shape = String annotation (choices(
  choice="" "none",
  choice="box" "box",
  choice="cylinder" "cylinder"));
parameter Shape shape="" "Animation shape";
```

It should be noted that it's possible to enter any value without using the pull down menu. This enables the use of expressions, for example.

Using radio buttons

In the case of only a set of fixed choices, radio buttons are more appropriate. Specification of sex can, for example, be made by radio buttons by adding `__Dymola_radioButtons=true`, i.e. if the following declaration is given (__Dymola_ means that the annotation is Dymola vendor specific):

```
Integer sex "Sex"
  annotation (Dialog(tab="Properties", group="Sex"),
    choices(__Dymola_radioButtons=true, choice=1 "Male",
    choice=2 "Female"));
```

Boolean variables such as

```
Boolean married "Marital status"
  annotation (Dialog(tab="Properties",
    group="Marital status"));
```

give by default a combo box with choices false and true. However, in many cases a check box is more convenient. This is achieved by adding __Dymola_checkBox=true, i.e. be giving the declaration

```
Boolean married "Marital status"
  annotation (Dialog(tab="Properties",
    group="Marital status"), choices(__Dymola_checkBox=true));
```

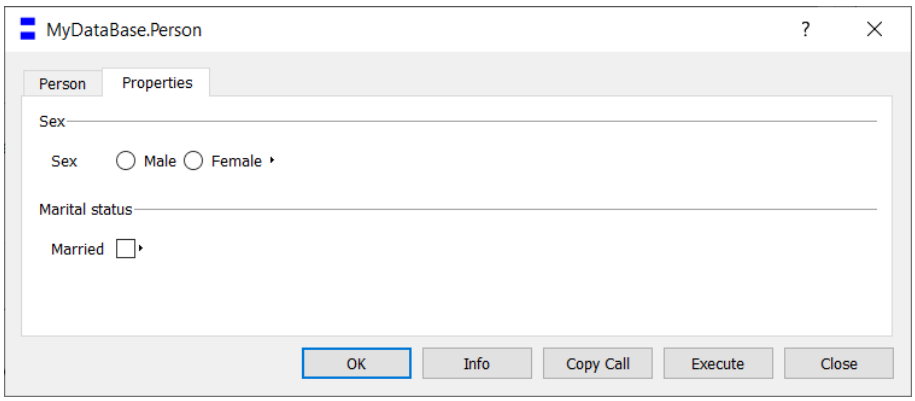
By adding these declarations for sex and married (removing the previous ones)

```
Integer sex "Sex"
  annotation (Dialog(tab="Properties", group="Sex",
    __Dymola_compact=true, __Dymola_descriptionLabel = true),
    choices(choice=1 "Male", choice=2 "Female",
    __Dymola_radioButtons=true));

Boolean married "Married"
  annotation (Dialog(tab="Properties",
    group="Marital status",
    __Dymola_compact=true, __Dymola_descriptionLabel = true),
  choices(__Dymola_checkBox=true));
```

including `__Dymola_compact=true` to move triangle closer to the input field, we obtain the following dialog layout:

Input field with radio buttons and check box.



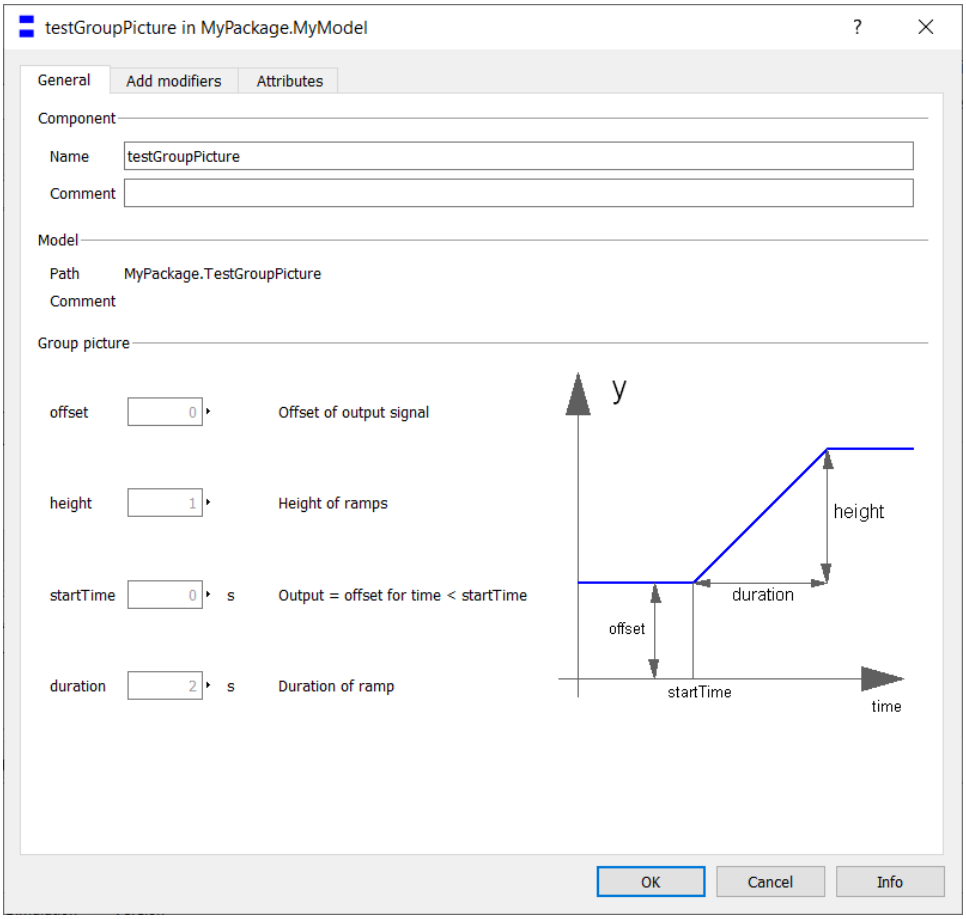
The annotations used above are not available as settings in the Declare variable dialog; it can however be entered in that dialog.

Illustrations and formatting in dialogs

Adding images to groups

To make it easier to understand the meaning of input data, it's possible to associate a picture with a Group:

Parameter dialog with illustrations.



The record declaration including the annotation to specify the file name of the picture is shown below. The last annotation creates the image. It is recommended to put pictures used in Dymola documentation in a folder "Images" in the same folder as the top package. It is also recommended to use the Modelica URI 'modelica://' scheme. If the top package is MyPackage, and an image "ramp.png" is located in a folder "Resources/Images" in the same folder as MyPackage, the annotation will be as below.

```
record TestGroupPicture
  parameter Real offset=0 "Offset of output signal"
    annotation(Dialog(group="Group picture"));
  parameter Real height=1 "Height of ramps"
    annotation(Dialog(group="Group picture"));
  parameter Modelica.SIunits.Time startTime=0
    "Output = offset for time < startTime"
    annotation(Dialog(group="Group picture"));
  parameter Modelica.SIunits.Time
    duration(min=Modelica.Constants.small) = 2
    "Duration of ramp"
    annotation(Dialog(group="Group picture"));

  annotation (__Dymola_Images(Parameters(group="Group picture",
    source="modelica://MyPackage/Resources/Images/ramp.png")));
end TestGroupPicture;
```

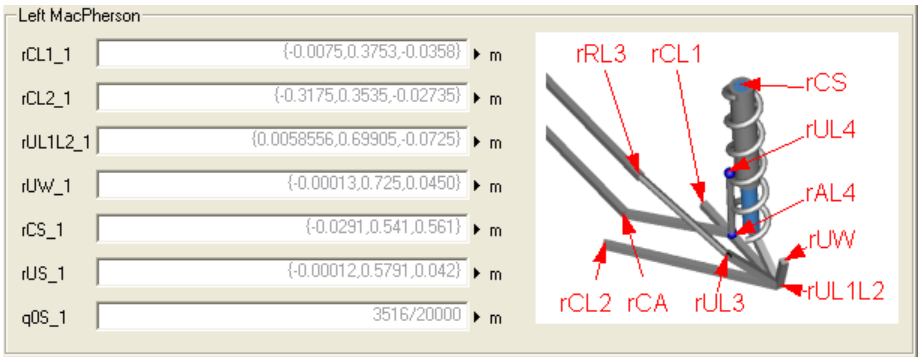
To test this functionality, please drag an instance of the record into the diagram layer and double-click it (or right-click it and select **Parameters**) to access the parameter dialog (the picture will not be shown if clicking on the record and selecting **Create Record...**).

The annotations used above are not available as settings in the Declare variable dialog; it can however be entered in that dialog.

An additional example

In this example for the Tab “Geometry” and the Group “Left MacPherson” we wanted to add an illustration showing the meaning of parameters.

Images in parameter dialog.



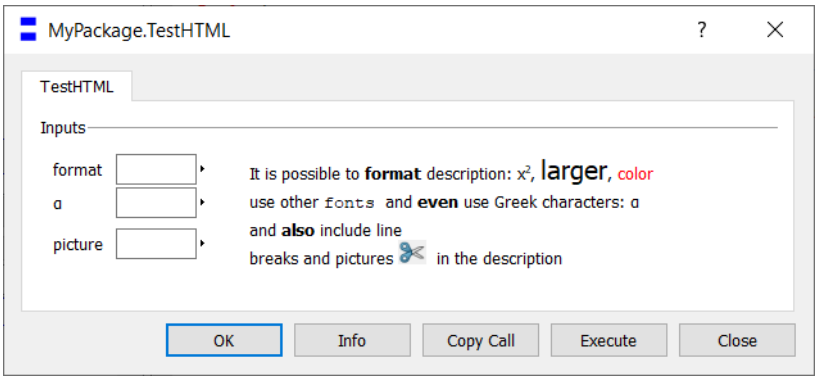
The syntax for adding the image to this group in the parameter dialog is:

```
annotation (__Dymola_Images(Parameters(tab="Geometry",
  group="Left MacPherson",
  source="modelica://MyPackage/Resources/Images/MacPherson_text.png")));
```

HTML formatting

The description texts and labels may contain HTML formatting tags if the text string is enclosed in <html> … </html>. The example below shows some of the possibilities.

Dialog with flexible formatting.



The corresponding record declaration is given below:

```
record TestHTML
  parameter Real format
    "<html>It is possible to <b>format</b> description:
      x<sup>2</sup>, <font size=\"+2\">larger</font>,
      <font color=\"#ff0000\">color</font> </html>";
  parameter Real alpha
    "<html>use other <font face=\"Courier New, Courier,
      monospace\">fonts </font> and <b>even</b>
      use Greek characters: &alpha;</html>"
    annotation(Dialog(__Dymola_label="<html>&alpha;</html>"));
  parameter Real picture
    "<html>and <b>also</b> include line <br> breaks and
      pictures
      <img src=\"E:/MyExperiments/Images/cut.png\" />  in
      the description</html>";
end TestHTML;
```

Greek symbols can, for example, be found at:

http://www.htmlhelp.com/reference/html40/entities/symbols.html

The annotations used above are not available as settings in the Declare variable dialog; it can however be entered in that dialog.

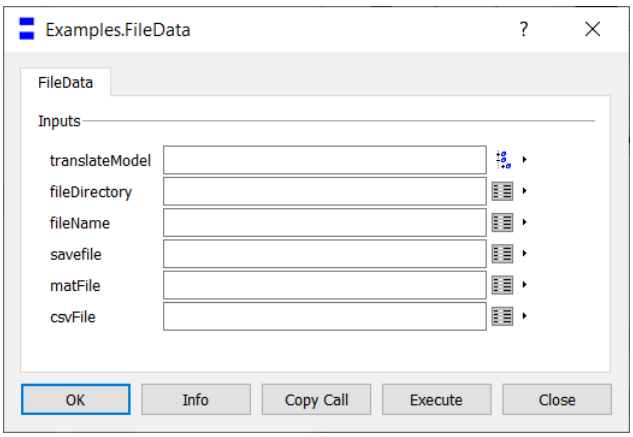
Specialized GUI widgets

Declarations of variables can be annotated to provide a convenient user interface, for example to select models, open files, input data in matrices or select color. These annotations are typically used to give inputs to functions or for creating records. The dialog is annotated with edit buttons (except when browsing files, then a browser symbol is automatically inserted).

File handling

The following dialog shows five examples of handling files. (The dialog pops when right-clicking on the record in the package browser and selecting **Create Record...**, but note that the dialog below assumes that all below types are defined also):

Input fields with associated specialized GUI widgets for file handling.



Given the appropriate type definitions (see below), such a record is very easy to declare. Let us do it in a package Examples.

```
record FileData
  Examples.TranslatedModel translateModel;
  Examples.FileName fileName;
  Examples.FileNameOut savefile;
  Examples.MatFileName matFile;
  Examples.CsvFileName csvFile;
end FileData;
```

The first example makes it possible to select a model. The type is declared (in the package Examples) as follows:

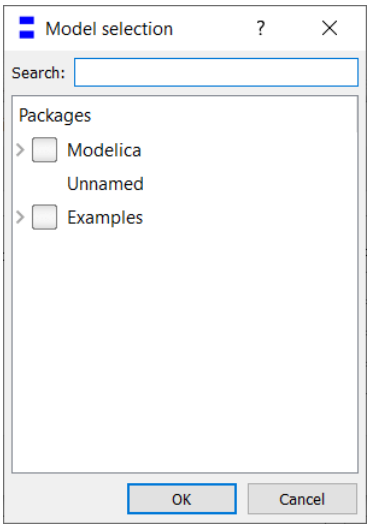
```
type TranslatedModel=String
annotation(Dialog(__Dymola_translatedModel(
   translate=true,caption="Model selection")));
```

If the selected model is not translated, it will be translated automatically. (Actually, translate=true can be omitted since it is true by default.) The caption will specify the header of the window that will pop. If omitted, the text will be “Select model”.



Pressing the **Edit** button for such function argument displays this dialog:

Translated model dialog.



The second example is a GUI widget for directory handling. It creates a button for selecting a directory:

```
type FileDirectory=String
  annotation(Dialog(__Dymola_directorySelector(caption"Select
Directory")));
```

The following declarations use annotations to display different kinds of file dialogs. The third example gets a filename for reading a file:

```
type FileName=String
  annotation(Dialog(__Dymola_loadSelector(filter="Matlab files
(*.mat);;CSV files (*.csv)",caption="Open experiment data
file")));
```

The fourth one gets a filename for writing a file:

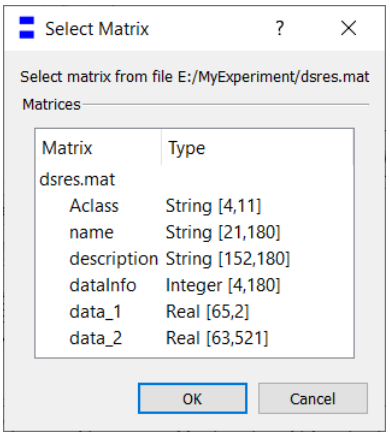
```
type FileNameOut = String
  annotation(Dialog(__Dymola_saveSelector(filter="Matlab files
(*.mat);;CSV files (*.csv)",caption="Save experiment data
file")));
```

The fifth one makes it possible to first select a .mat file, and then selecting a matrix in that file.

```
type MatFileName=String
annotation(Dialog(__Dymola_loadSelector(matrixAfter="|",filter=
"Matlab files (*.mat)",caption="Open experiment data file")));
```

As an example just to show how the annotation works a user might want to read a matrix in the file dsres.mat, which was created when simulating the CoupledClutches demo using the command **Simulation > Commands > Simulate and Plot** (after having selected E:/MyExperiment as working directory). Using the widget above to select the resulting `dsres.mat`, the following dialog for selection of matrix in that file will pop:

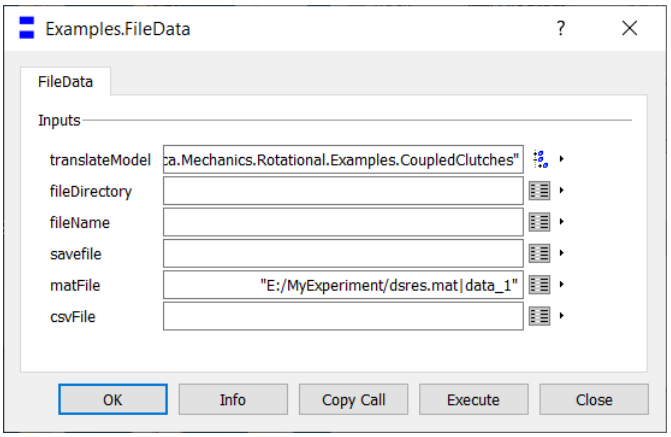
Selection of matrix in a .mat file.



(Note that if we had simulated with the command **Simulation > Simulate**, the result file name would have been CoupledClutches.mat instead of dsres.mat, due to different functions used when simulating.)

Selecting the alternative `data_1` and clicking **OK** will result in:

Selection of a matrix in a .mat file – the result.



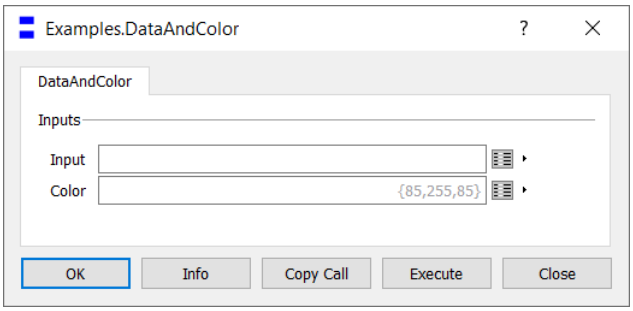
The sixth example makes it possible to read a csv file or a txt file (it is actually of the same type as the first one above).

```
type CsvFileName=String
  annotation(Dialog(__Dymola_loadSelector(filter="CSV files
(*.csv);;Text csv files (*.txt)",caption="Open experiment data
file")));
```

Data input and color handling

The following shows a dialog for data input in a matrix and color selection (with default value of the color) Note that the dialog below assumes that below types are defined also:

Input fields with associated specialized GUI widgets for data input and color.



The corresponding record is:

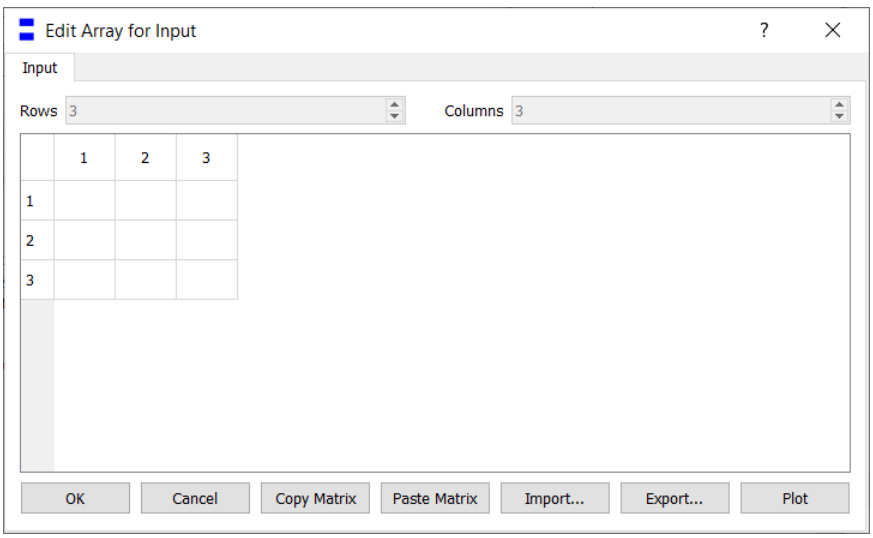
```
record DataAndColor
  Examples.MatrixData Input;
  Examples.Color Color={85,255,85};
end DataAndColor;
```

A type for entering the data of a 3 x 3 real matrix can look the following:

```
type MatrixData = Real[3,3];
```

Pressing the corresponding **Edit** button will display:

Input of matrix data.

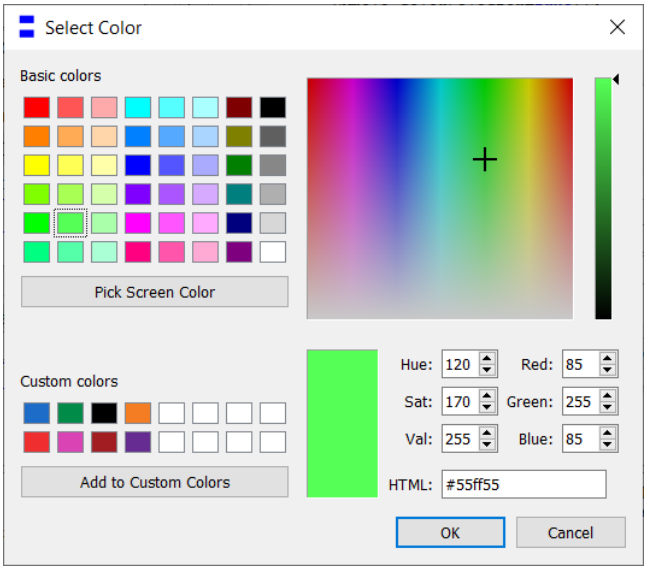


And the last type for color selection:

```
type Color=Real[3] annotation(Dialog(
    __Dymola_colorSelector=true));
```

Pressing the corresponding **Edit** button will give (note that default values have been given in the record definition):

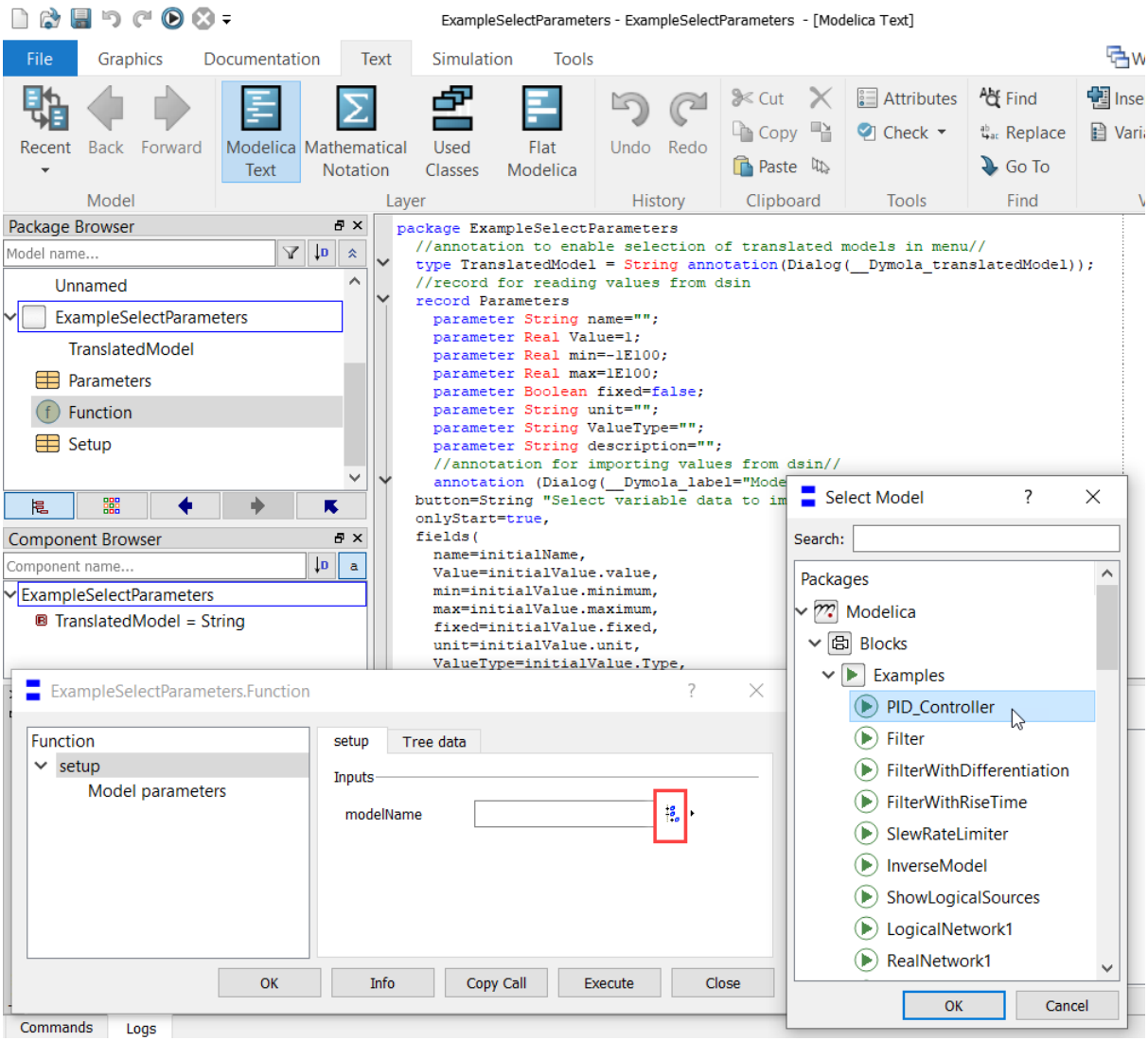
Color selection.



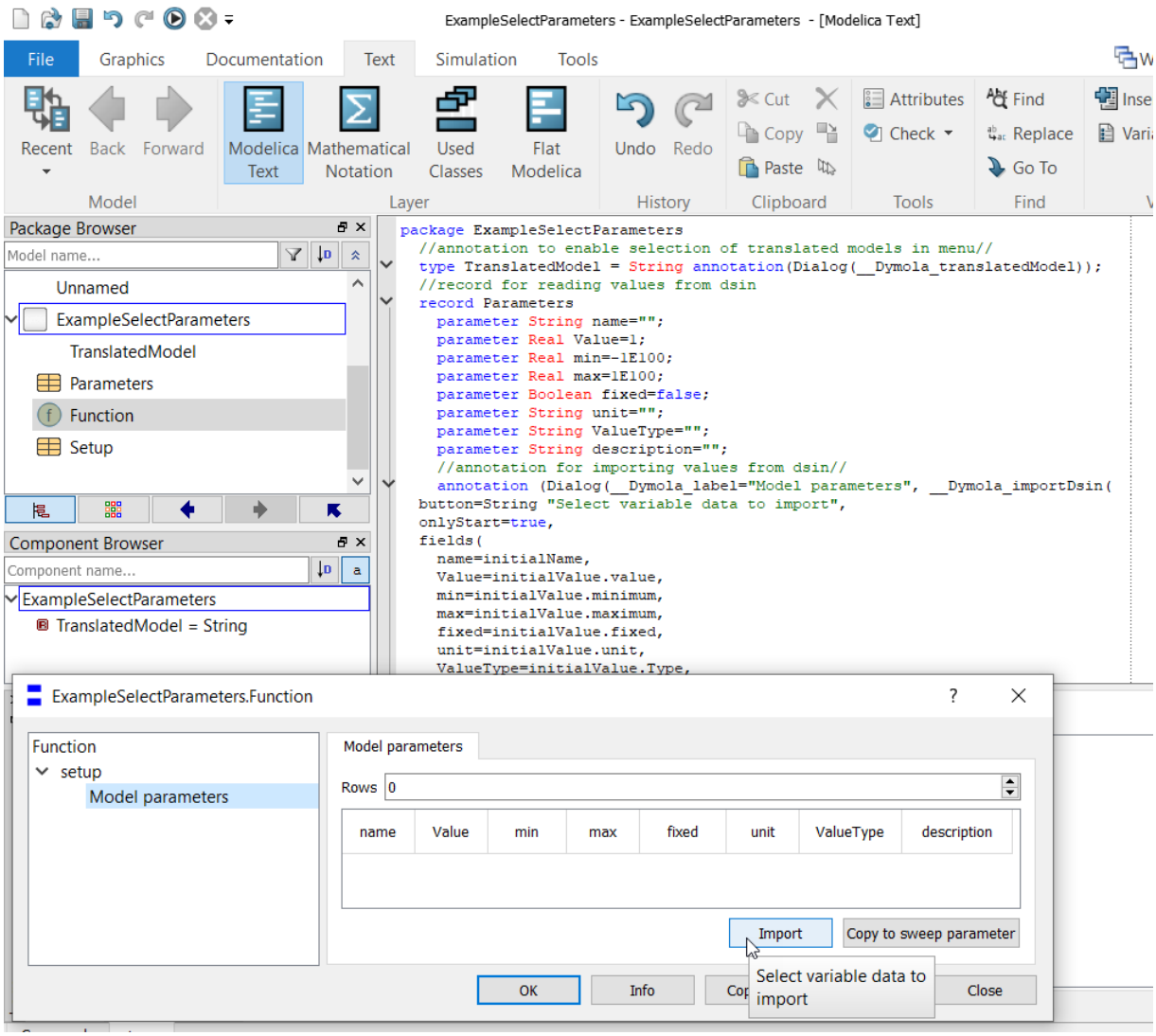
Import of model data in functions

The annotation `__Dymola_importDsin` makes it possible to define a function that enables the user to import selected data from a model.

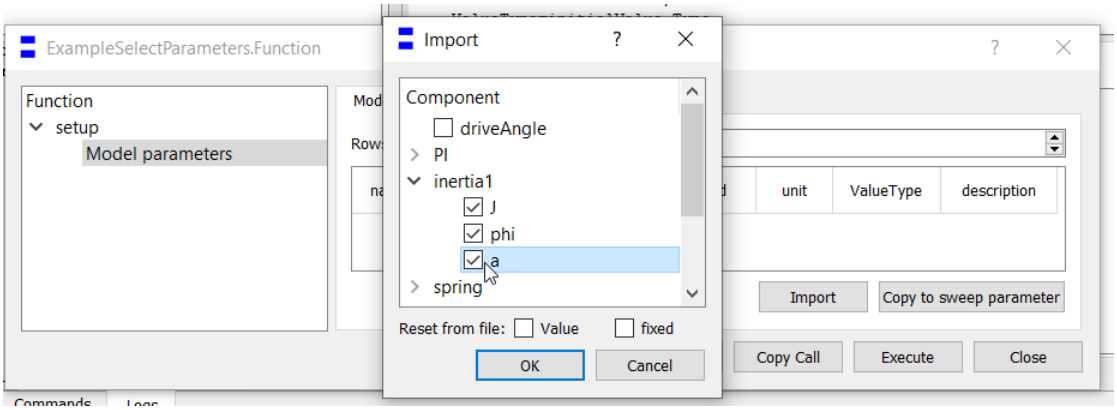
To first select the model to import from, right-click the function **Function** in the package browser and then select **Call Function...** and then select **setup** in the pane to the left in the dialog. You can now browse for the model, in this example, select Modelica.Blocks.Examples.PID_Controller:



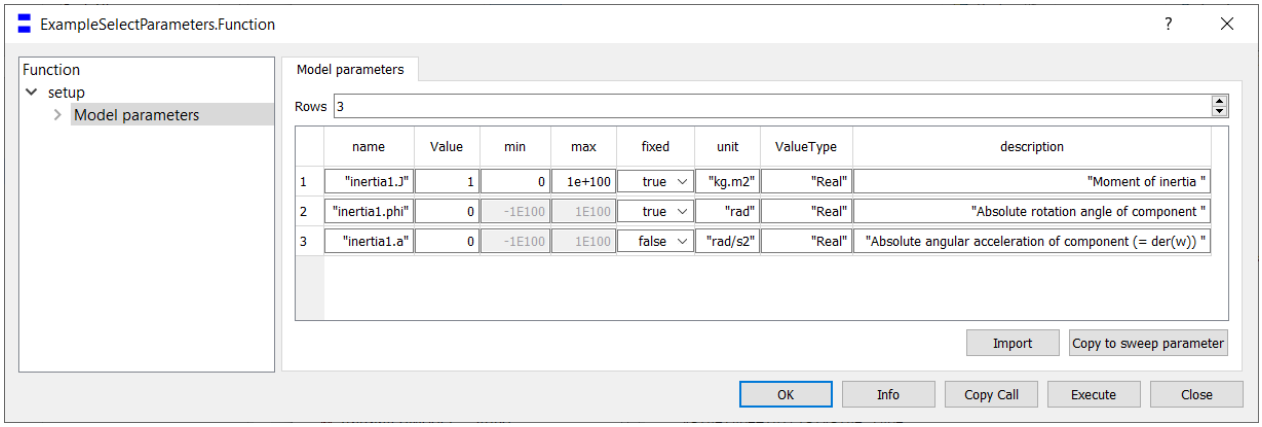
To import values from the selected model, select, in the dialog of the function call you just used, **Model parameters**. In the resulting dialog, a button **Import** is displayed:



Clicking on that button pops a tree of variables to select from.



When clicking **OK**, the imported data will be shown in the dialog (the columns have been somewhat adapted for nicer presentation):



It is possible to change e.g. the value of inertia1.J before importing the data into the function by clicking **OK**.

This feature is used in several menus in e.g. the Calibration package; please see the chapter “Model Calibration” for more examples of such menus and the use of those.

The code behind this example looks the following, some blank lines added for better readability (compare also with the package browser in the figure above when it comes to the structure of the code):

```
package ExampleSelectParameters

  //annotation to enable selection of translated models in menu//
  type TranslatedModel = String
annotation(Dialog(__Dymola_translatedModel));

  //record for reading values from dsin
  record Parameters
    parameter String name="";
    parameter Real Value=1;
    parameter Real min=-1E100;
    parameter Real max=1E100;
    parameter Boolean fixed=false;
    parameter String unit="";
    parameter String ValueType="";
    parameter String description="";
    //annotation for importing values from dsin//
    annotation (Dialog(__Dymola_label="Model parameters",
__Dymola_importDsin(
  button=String "Select variable data to import",
  onlyStart=true,
  fields(
    name=initialName,
    Value=initialValue.value,
    min=initialValue.minimum,
    max=initialValue.maximum,
    fixed=initialValue.fixed,
    unit=initialValue.unit,
    ValueType=initialValue.Type,
    description=initialValue.description))));
  end Parameters;

  //function that uses record Setup that in turn uses record
Parameters//
  function Function
    input ExampleSelectParameters.Setup setup;
    output Real out;

  algorithm
    //here code should be added to do something with the input//
  end Function;

  //record used in function//
  record Setup
    TranslatedModel modelName;
    Parameters parameters[:]=fill(Parameters(),0);
  end Setup;

end ExampleSelectParameters;
```

The code is shown with all annotations expanded. The annotation in the middle of the code is the one that makes it possible to import data from models. Extracting only the `__Dymola_importDsin` part of the annotation from the example we find:

```
__Dymola_importDsin(onlyStart=true,
  fields(
    name=initialName,
    Value=initialValue.value,
    min=initialValue.minimum,
    max=initialValue.maximum,
    fixed=initialValue.fixed,
    unit=initialValue.unit,
    ValueType=initialValue.Type,
    description=initialValue.description))
```

(For a full list of what can be handled in the annotation, see notes below.) In this example all possible attributes from a signal is being imported. The corresponding attributes are shown as headers when importing (please see previous figures in this example). If less attributes are needed, just omit that attribute when writing the code (e.g. to not import the data type of the signal in this example, remove `ValueType=initialValue.Type`, from the annotation – and don't forget to also remove `parameter String ValueType=""` from the Parameter record definition).

It is possible to select what kind of signals that should be selectable.

In this example initial values of parameters and states are selectable because of using `onlyStart=true` in the annotation. Please note that this means the parameter values that are used when starting the simulation of the model; if a parameter has a start value of 2 but the user has input 14 before simulation, it will be the value 14 that will be imported.

The default text of the button and the tooltip is used in the example. It other texts are wanted; the annotation in the example can be changed to e.g.

```
…__Dymola_importDsin(button="Mytext" "My tooltip",onlyStart=…
```

Some notes about the example:

- This example only shows how to import different data to a function and nothing more, that is, the function in this example is meaningless; the imported data has to be used also, by adding code in the algorithm part of the function to e.g. specify the output of the function.
- To be able to create the function dialog, the record Setup is created. That record is used as input for the function, and supplies the model that data should be selected from, and the data that is imported from that model using the record Parameters.
- __Dymola_translatedModel that is used in the record Setup is a type that has an annotation that makes it possible to select from translated models from a tree structure; this annotation has been used in a previous example also.
- dsin is a file where data of a model is stored.

- The full annotation is:

```
__Dymola_importDsin(
  button=String "comment",
  onlyStart=Boolean,
  onlyInput=Boolean,
  onlyTimeVarying=Boolean,
  nonStaticMessage=String
  fields(
    name=initialName,
    Value=initialValue.value,
    min=initialValue.minimum,
    max=initialValue.maximum,
    fixed=initialValue.fixed,
    unit=initialValue.unit,
    ValueType=initialValue.Type,
    Description=initialValue.description))
```

  Not mentioned in the example above were `onlyInput`, `onlyTimeVarying` and `nonStaticMessage`. If `onlyInput` is set to true, only inputs are imported, if `onlyTimeVarying` is set to true, only time varying variables are imported. `nonStaticMessage` will display the specified string as message when a model is time dependent.

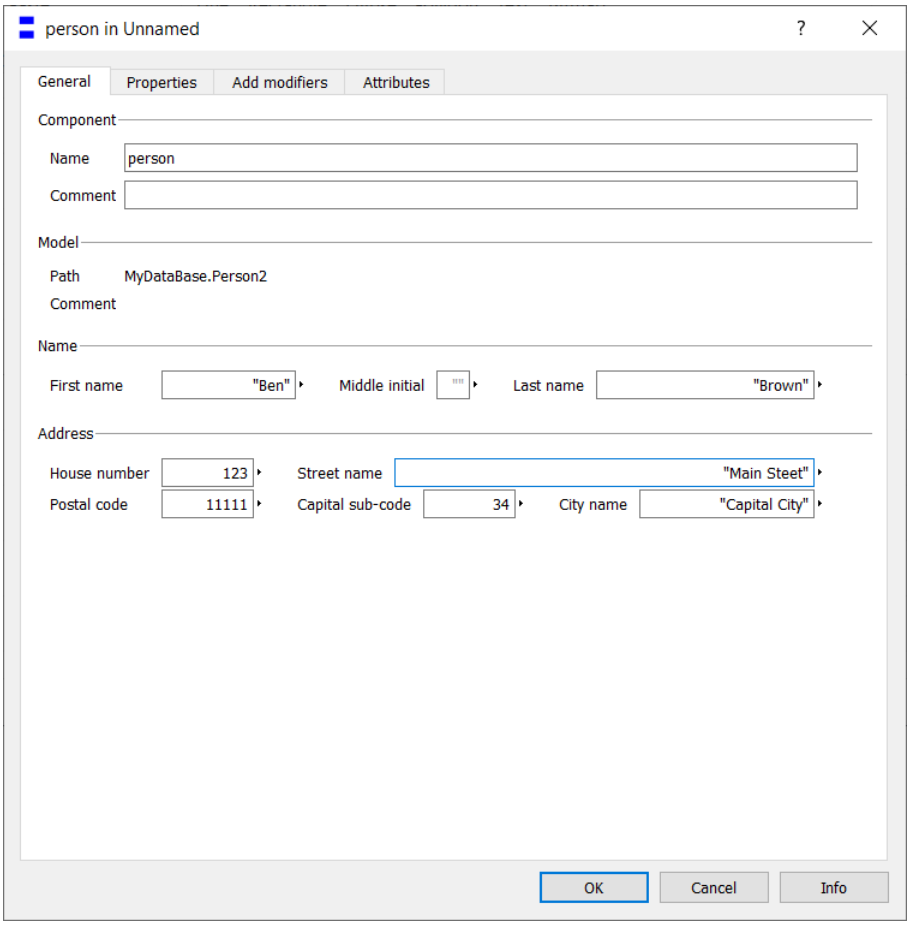
Conditional input of data

It is possible to control whether data input is possible or not for a variable using the annotation `annotation(Dialog(enable=`*`expression`*`))`.

The annotation is easily applied using the Declare variable dialog; the Annotation tab. The Parameter dialog placement group contains an **Enable** input field. Here the condition can be entered.

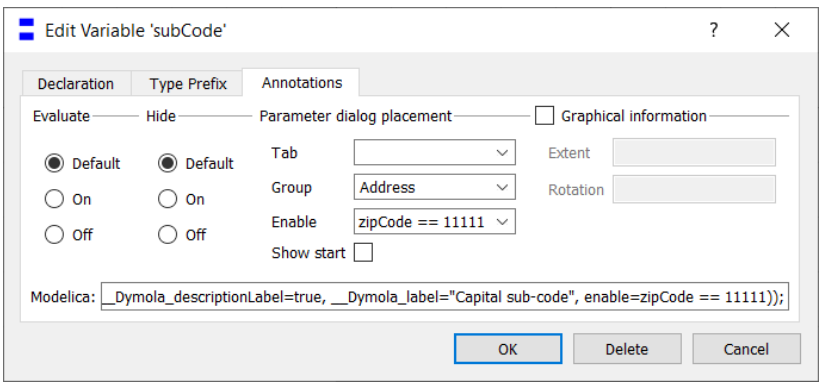
As an example, assume the capital to have zip code 11111, having a number of sub-codes. The field for entering sub-codes should only be possible to access if the zip code is entered as 11111. If you drag an instance of Person into the diagram layer of the edit window and display the parameter dialog by double-clicking the instance (or right-click it and select **Parameters**) the parameter dialog might look the following:

Parameter dialog for an instance of Person.



In this case the field for the sub-code is accessible.

The corresponding Annotation tab for the variable `sub_code` is:



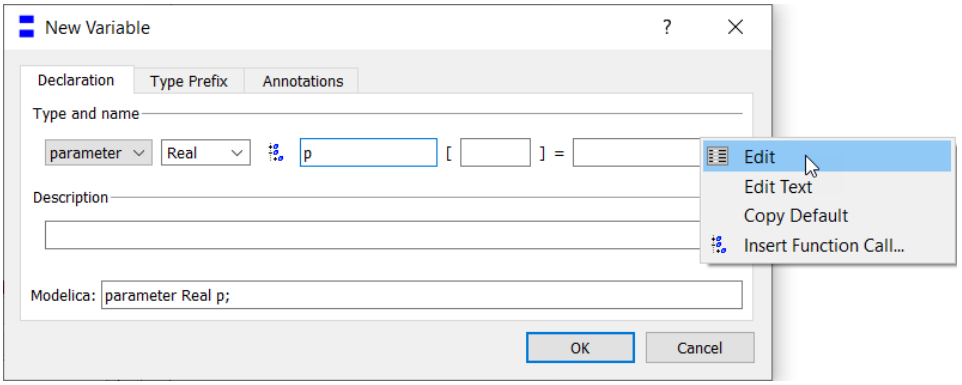
The full Modelica code line in the last line of the window is:

```
Integer subCode "Sub-code for capital" annotation
(Dialog(group="Address", __Dymola_joinNext=true,
__Dymola_absoluteWidth=10, __Dymola_descriptionLabel=true,
__Dymola_label="Capital sub-code", enable=zipCode == 11111));
```

Checking of input data

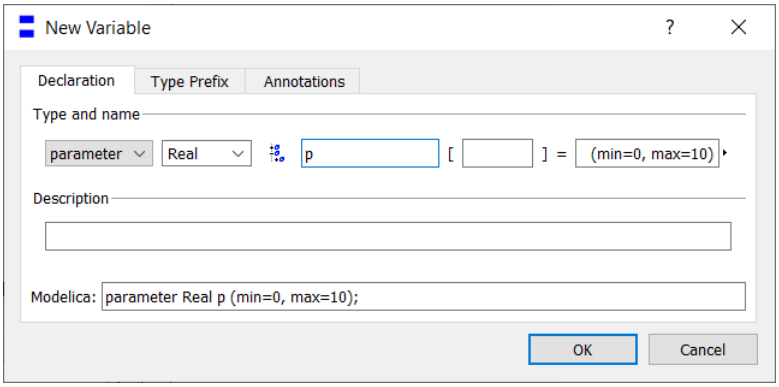
It is possible to declare parameters with minimum and maximum values, which are then checked by Dymola when the user sets a parameter value. The variable is declared with type and name as usual using the command **Text > Insert > New variable...**. (or **Graphics > Insert > New variable...**). Then press the edit button (right-arrow) at the end of the value field to present a menu.

Editing attributes of variables.



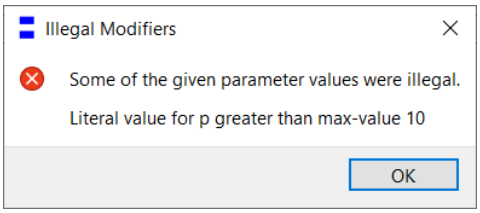
Select **Edit** from the menu and enter the min and max values for the parameter. Assuming that we have specified the range to be [0, 10], the variable dialog shows

Variable dialog with min and max attributes.



If we have a model with such a parameter and try to set a value outside of the valid range, Dymola will display an error message. The parameter dialog cannot be closed until the invalid modifier value has been corrected.

Out of bound error message.



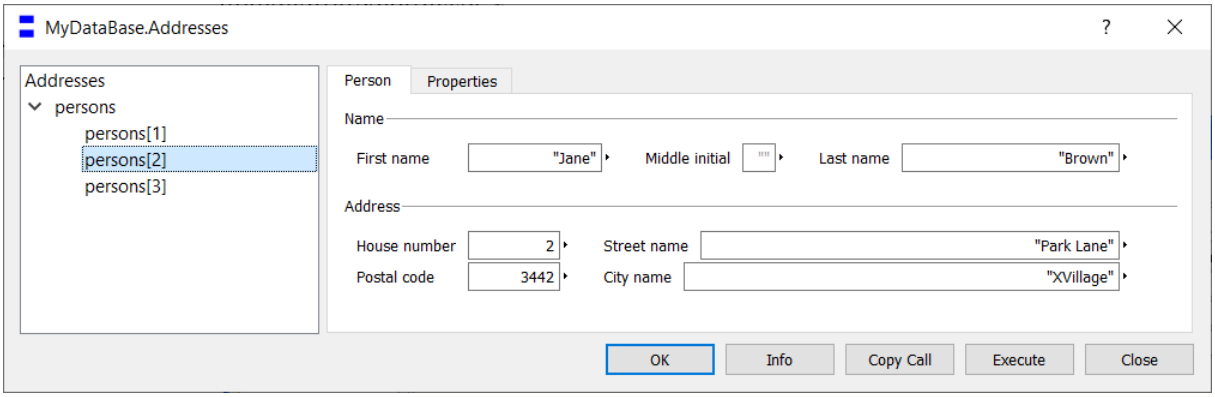
Arrays of records

A simple address book can be created as an array of Person records as follows:

```
record Addresses
  Person persons[:];
end Addresses;
```

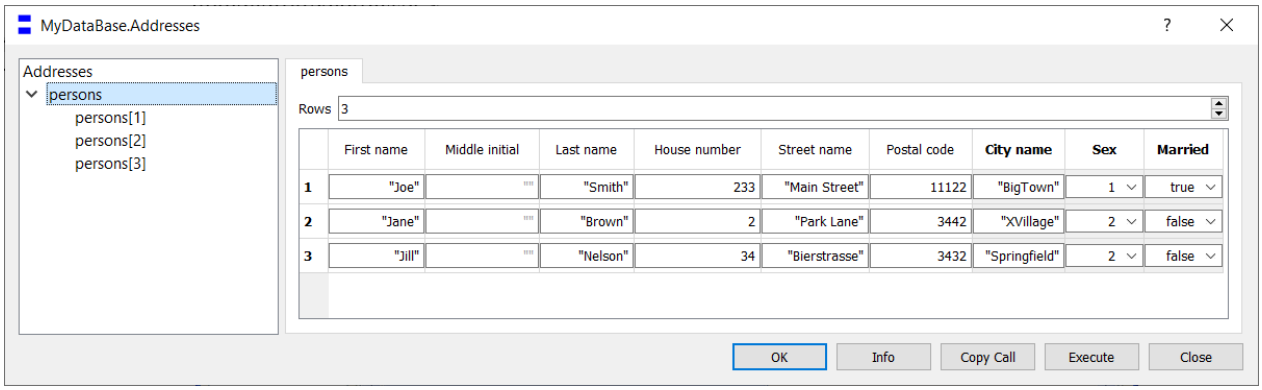
Dialog for one record in an array.

The corresponding dialog for one record in such an array is:



Dialog for an entire array of records.

It is also possible to view all person records at the same time by selecting the array “persons” in the left tree browser:

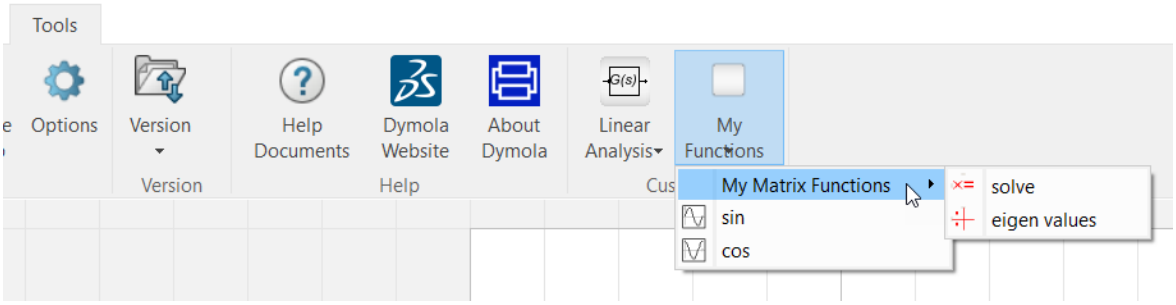


15.2 Extendable user interface – menus, toolbars and favorites

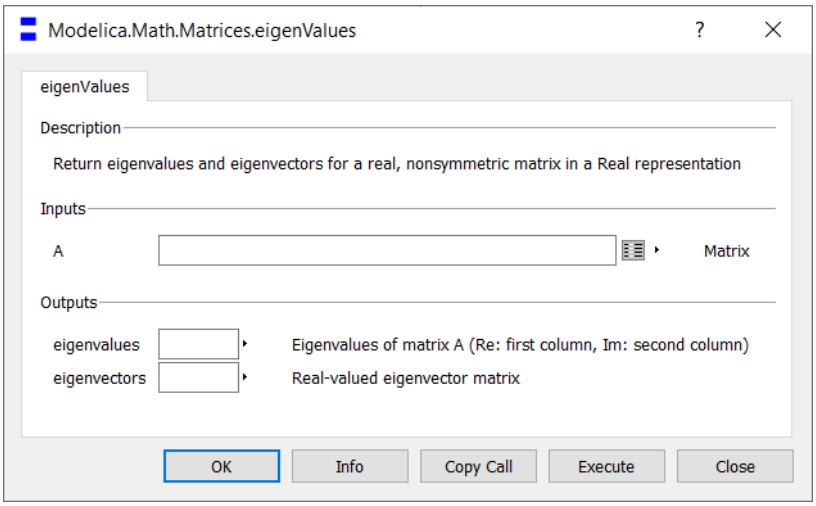
15.2.1 Defining content of menus and toolbars

It is possible to extend the graphical user interface of Dymola by introducing own menus and toolbars from which Modelica functions can be called. Such menus will be available in the **Tools** tab.

The following image shows a new menu **My Functions**. It contains a submenu **My Matrix Functions** with two menu items; **sin** and **cos**.



When selecting **eigen values**, the dialog of `Modelica.Math.Matrices.eigenValues` function is shown:



The definition of menus and toolbars are done by defining a package structure with short extends definitions to the functions to be called.

```
package MyFunctions "My Functions"

  package MyMatrixFunctions "My Matrix Functions"
    function solve=Modelica.Math.Matrices.solve(A=[1,2;3,4],b={1,1})
             annotation (Icon(coordinateSystem(
               preserveAspectRatio=false, extent={{-100,-100},{100,100}},
               graphics={Line(
               points={{-100,-100},{-100,100},{0,0},{100,100},{100,-100}},
               color={0,0,255}, smooth=Smooth.None, thickness=0.5)})));
    function eigenValues=Modelica.Math.Matrices.eigenValues "eigen values"
             annotation (Icon(coordinateSystem(preserveAspectRatio=false,
                extent={{-100,-100},{100,100}}, graphics={Line(
                points={{-100,-100},{-100,100},{0,0},{100,100},{100,-100}},
                color={0,0,255}, smooth=Smooth.None, thickness=0.5)})));
        annotation (Icon(coordinateSystem(preserveAspectRatio=false,
           extent={{-100,-100},{100,100}}, graphics={Line(
           points={{-100,-100},{-100,100},{0,0},{100,100},{100,-100}},
           color={0,0,255}, smooth=Smooth.None, thickness=0.5)})));
  end MyMatrixFunctions;

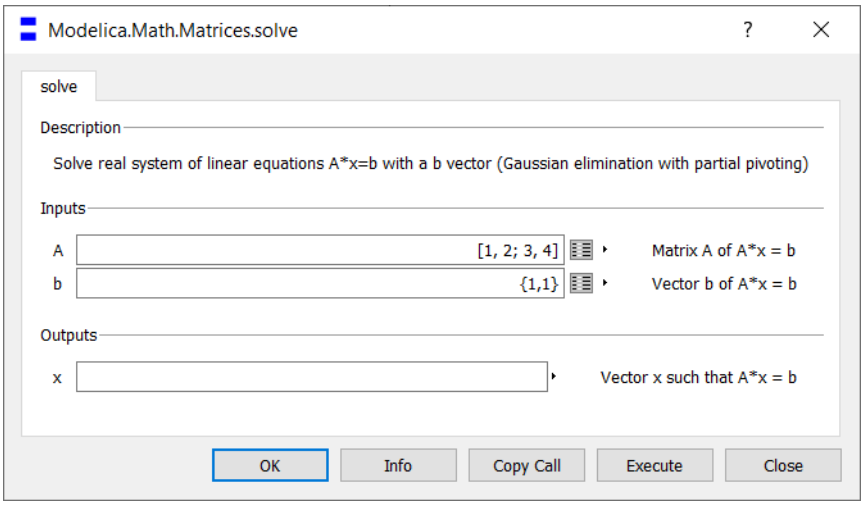
  function sin=Modelica.Math.sin;
  function cos=Modelica.Math.cos;

  annotation (
  __Dymola_toolbar=true,
  __Dymola_menu=true,
  Protection(hideFromBrowser=true));
end MyFunctions;
```

Note that icon annotations have been given for solve, eigenValues and MyMatrixFunctions and that these are used in the menus. To increase the readability of the code above, extra line breaks and indentations have been added for the icon annotations in the code above.

If a description string is given, for example “My Functions”, “My Matrix Functions” and “eigen values” in the example above, those strings are used for the menu, otherwise the defined name.

It is possible to give values to inputs which then becomes prefilled in the dialog; in the example above the command **My Functions > My Matrix Functions > solve** will display:



The package structure is made into a menu by providing (as in the example above) the annotation

```
annotation(__Dymola_menu=true);
```

(The annotation `annotation(__Dymola_toolbar=true)` will for Dymola 2020x and later also produce the same result; in the above model both are present, but only one is needed.)

It is possible to hide such a package from the package browser by (as in the example above) using the annotation:

```
annotation(Protection(hideFromBrowser=true));
```

such a package can be automatically loaded by using a `libraryinfo.mos` file with `category="persistent"`:

```
LibraryInfoMenuCommand(
  Category="persistent",
  Reference="<Class to preload>",
  Text="dummy")
```

It is then not removed when the **File > Clear All** command is given.

The annotation `__Dymola_hideGraphics=true` can be used to hide the graphics coming from such a class. This also means that the base class is only loaded when needed to build the parameter dialog and call the function.

15.2.2 Displaying library-specific menus and toolbars in Dymola (commercial library developers)

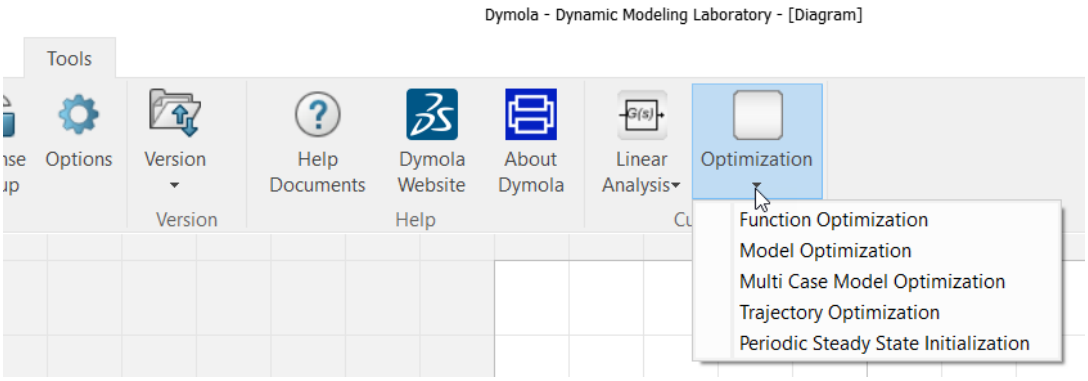
Commercial library developers can in their libraries include menus and toolbars that will be displayed in Dymola when opening the libraries. To do this, the annotation

```
annotation(__Dymola_containsMenu=true);
```

must be present in the top level of the library (e.g. in package.mo). This causes all classes in the library to be searched for menu and toolbar annotations.

The package containing the menu and toolbar annotations described in previous section must be located inside the library.

An example of a menu created this way is the menu that will be displayed when opening the Optimization library:



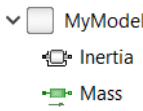
The menu will disappear when performing **File > Clear All** or unloading the library.

15.2.3 Defining packages with users own collection of favorite models

It is possible to define packages with your own collection of favorite models by introducing shortcut to models:

```
package MyModels
  model Inertia=Modelica.Mechanics.Rotational.Components.Inertia annotation (__Dymola_shortcut=true);
  model Mass=Modelica.Mechanics.Translational.Components.Mass(L=1) annotation (__Dymola_shortcut=true);
end MyModels;
```

The models appear in the package browser the usual way:



When dragging from such a shortcut, an instance of the original model is created, possible with prefilled parameter values if a modifier has been used in the short class definition.

Note that a simpler way of working with favorite packages is also available, including the automatic creation of such a package if not existing. For more information, see the chapter “Developing a model”, section “Basic model editing”, subsection “Packages, models, and other classes”.

16 APPENDIX - MIGRATION

16 Appendix — Migration

16.1 Migrating to newer libraries

Dymola supports migration of models using one set of model libraries to another set of model libraries with model components of similar structure, but for example having other class names or different connector names or parameter names. Dymola has commands to build up internal translation tables which will take effect when a model library or a model component is loaded. These commands can be collected in a script allowing all models that use a specific library to upgrade to the new one.

The feature was originally designed to convert models to the Modelica Standard Library, and it has been updated to handle conversion from version 1 and 2 of the Modelica Standard Library (primarily vectorization of blocks-library), and version 2 and 3 (primarily removing SignalType and replacing some modifier equations by normal equations).

Dymola supports multiple upgrade of libraries; see section “Upgrading models and libraries to a new library version” starting on page 1534.

16.1.1 How to migrate

Assume that we would like to migrate myModel that uses a package OldLibrary to exploit a NewLibray. In many cases, in particular when releasing a new version of a library, the library

developer will provide a conversion script, which specifies the translation from the old version to the new version of the library.

Assume that the model is stored in a file named myModel.mo. It is advisable to also have a backup copy of the file. Assume also that a specification of the translation when migrating from using OldLibrary to using NewLibrary is specified in the script file Convert.mos.

To migrate myModel proceed as follows.

1. Start with a fresh Dymola.
2. Use the command **File > Open > Open…** to open the file NewLibrary.
3. In the Dymola main window, in **Simulation** tab, use the **Run Script** button  to run the script Convert.mos.
4. Use the command **File > Open > Open…** to open the file myModel.mo
5. Perform a check using the command **Check**.
6. Hopefully there is no error message and the model can be saved. The conversion is done.
7. In case of error message consult the next two subsections on specifying translation and building a script file.
8. After a migration, the model shall of course be tested and it shall be checked that it gives the same simulation result as the old one.

16.1.2 Basic commands to specify translation

The command to build the translation tables are

```
convertClass("oldClass", "newClass");
convertClassIf("OldClassName","para","val","NewClassName");
convertElement("oldClass", "oldElement", "newElement");
convertModifiers("oldClass", oldParameterBindings,
   newParameterBindings);
convertClear();
```

convertClass

The command

```
convertClass("oldClass", "newClass");
```

builds conversion tables to be applied on extends clauses and type declarations. As an example consider:

```
convertClass("DriveTwoCut",
  "Modelica.Mechanics.Rotational.Interface.Compliant");
```

All components of type DriveTwoCut or classes that contain extends DriveTwoCut will be converted such that they refer to class Modelica.Mechanics.Rotational.Interface.Compliant instead.

Conversion is also applied on hierarchical names. For example

```
convertClass("Modelica.Rotational1D","Modelica.Rotational")
```

will take effect on Modelica.Rotational1D.Clutch and give Modelica.Rotational.Clutch.

Modelica's normal vectorization applies for convertClass, which means that it is possible to let the arguments be vectors of Strings. For example,

```
convertClass({"oldClass1", "oldClass2"},
             {"newClass1", "newClass2"});
```

is equivalent to

```
convertClass("oldClass1", "newClass1");
convertClass("oldClass2", "newClass2");
```

Using convertClass for converting emulated enumerations

The convertClass can be used for converting an emulated enumeration (using package constants) to a proper enumeration. As an example consider the following emulated enumeration in Modelica Standard Library version 2:

```
package RotationTypes
  extends Modelica.Icons.Enumeration;
  constant Integer RotationAxis=1;
  constant Integer TwoAxesVectors=2;
  constant Integer PlanarRotationSequence=3;
  type Temp "Temporary type of RotationTypes"
    extends Modelica.Icons.TypeInteger;
  end Temp;
end RotationTypes;
```

This can now be replaced by a proper enumeration:

```
type RotationTypes = enumeration(
   RotationAxis,
   TwoAxesVectors,
   PlanarRotationSequence);
```

The enumeration automatically handles the constants, and to also convert the Temp-part it is only necessary to add:

```
convertClass("…RotationTypes.Temp","…RotationTypes");
```

The "…" shall be replaced by the proper path to RotationTypes.

Using convertClass for adding component prefixes

In some cases elements of one class should be converted to inner elements of another class, as when converting InertialSystem of ModelicaAdditions to World in Modelica Standard Library version 2. This is done by prefixing the new class-name by "inner", followed by a space and then the new class-name:

```
convertClass("ModelicaAdditions.MultiBody.Parts.InertialSystem,
  "inner Modelica.Mechanics.MultiBody.World world");
```

Other allowed prefixes are "outer ", "parameter", and "constant".

convertClassIf

It is possible to optionally convert components to a different class using

```
convertClassIf("OldClassName","para","val","NewClassName");
```

If the parameter has the given value the component will be converted to this class, and the modifier will be removed. The modifier will otherwise be kept (thus the safe option for a Boolean parameter is give one conversion for true and another for false). Another variant is to base it on whether a connector is connected:

```
convertClassIf("OldClassName","connector","connect",
  "NewClassName");
```

The normal conversion will be used if these conditions are not satisfied and all conversion of elements, modifiers is common for these cases.

convertElement

The command

```
convertElement("oldClass", "oldElement", "newElement");
```

converts references to elements (connectors, parameters and local variables of classes) in equations, connections, modifier lists etc. The class name of the component is still converted according to convertClass.

The conversion uses the model structure after conversion, thus it correctly detects base-classes among the models you convert. However, only the new library is used, thus any inheritance used in the old library is lost. It means, for example, if a connector was renamed in base-class conversion of that connector name must be specified for all models in the old library extending from base-class. By using vector arguments to the conversion functions it is only necessary to list the classes once for each renamed element.

Let us illustrate by an example. Assume that we have a drive train library, where there is a partial class DriveTwoCut specifying two connectors pDrive and nDrive. The new library has a similar class TwoFlanges defining two connectors flange_a and flange_b. We thus give the commands

```
convertClass("DriveTwoCut", "TwoFlanges");
convertElement("DriveTwoCut", "pDrive", "flange_a");
convertElement("DriveTwoCut", "nDrive", "flange_b");
```

Assume that the old library contains the models Shaft and Gear, which are to be converted as Inertia and IdealGear, respectively:

```
convertClass("Shaft", "Inertia");
convertClass("Gear", "IdealGear");
```

Assume that Shaft and Gear extend from DriveTwoCut. Unfortunately, there will be no translation of references in, for example, connect statements to their connectors pDrive and nDrive, since the conversion uses the model structure after conversion. To have a proper translation, we need also to specify

```
convertElement({"Shaft", "Gear"}, "pDrive", "flange_a");
convertElement({"Shaft", "Gear"}, "nDrive", "flange_b");
```

where the vectorization allows a compact definition.

The convertElement command can also be used when a parameter is renamed. For more complex reparameterizations the command convertModifiers is useful.

convertModifiers

The command

```
convertModifiers("oldClass", oldParameterBindings,
  newParameterBindings);
```

specifies how parameter bindings given in modifiers are to be converted. The argument oldParameterBindings is a vector of strings of the form "oldParameter=defaultValue", and the argument newParameterBindings is a vector of strings of the type "newParameter=expression". To use the value of an old parameter in the new expression use %oldParameter%

As an example, assume that in the old model Clutch that the viscous friction coefficient mue is given as

```
mue = mueV0 + mueV1*abs(wrel)
```

where mueV0 and mueV1 are parameters declared in Clutch as

```
parameter Real mueV0 = 0.6;
parameter Real mueV1 = 0;
```

The model Clutch of the new model library uses linear interpolation with respect to the relative velocity, wrel, with a parameter mue_pos to define the interpolation table

```
parameter Real mue_pos[:, :] = [0, 0.5];
```

The original mue-equation is linear and one way of specifying a linear interpolation table is to compute its value for two arbitrary velocities. The model requires that the first velocity is zero, with mue=mueV, and for the other value velocity we use velocity one, with mue=mueV0 + mueV1. Thus at translation we would like to obtain a new modifier

```
mue_pos = [0, value-of-mueV0;
           1, value-of-mueV0 + value-of-mueV1];
```

This is obtained by

```
convertModifiers("Clutch",{"mueV0=0.6","mueV1=0"},
  {"mue_pos=[0,%mueV0%;1,%mueV0%+%mueV1%]}");
```

Example, the declarations in the old model

```
Clutch c1(mueV0=0.4,mueV1=0.1)
Clutch c2(mueV0=2*p);
```

are converted to

```
Modelica.Mechanics.Rotational.Clutch
   c1(mue_pos = [0,(0.4); 1,(0.4)+(0.1)]);
Modelica.Mechanics.Rotational.Clutch
   c2(mue_pos=[0,(2*p); 1,(2*p)+(0)]);
```

Note that since c2 did not specify a value for mueV1 the conversion used the default value. The substitution automatically adds parenthesis for the substituted arguments, thus avoiding the need for parenthesis in macros that are familiar to a C programmer. The parentheses are sometimes redundant and can be removed by going to the parameter dialogs of the corresponding components.

Prefer convertElement over convertModifiers even for parameters, and only use convertModifiers when there is a need for more than a one-to-one conversion of parameters.

Using convertModifiers for replacing modifiers by equations

A special case is converting modifier-equations (for BaseProperties-members) to normal equations when converting to Modelica 3. This is handled by:

```
convertModifiers("…BaseProperties",{"p", "T" ,…},
   {"equation"});
```

Using convertModifiers for removing vectorization

The special case of removing vectorization of a block (as was done between version 1 and 2 of Modelica) has two cases, either the size is explicitly given as a size-parameter or implicitly by the size of a number of a vector-parameters. The first case is handled by conversion of the size-parameter to the special value "size" – where a value different from one indicates that the conversion will replace the block by an array of appropriate size:

```
convertModifiers("Modelica.Blocks.Math.Sin",{"n=1"},
   {"size=%n%"});
convertElement({"Modelica.Blocks.Math.Sin", "outPort ", "y");
convertElement({"Modelica.Blocks.Math.Sin", "inPort ", "u");
```

The other case is handled by converting each vector-parameter to a scalar, and in case any of them is a vector of length different from one the block is replaced by a vector of blocks.

```
convertModifiers("Modelica.Blocks.Math.Gain",
                 {"k={1}"},
                 {"k=scalar(%k%)"});
convertElement({"Modelica.Blocks.Math.Sin",
   "outPort.signal", "{y}");
convertElement({"Modelica.Blocks.Math.Gain", "outPort", "y");
convertElement({"Modelica.Blocks.Math.Gain", "inPort", "u");
convertElement({"Modelica.Blocks.Math.Gain", "outPort.signal",
   "{y}");
```

The conversion of outPort.signal is used in case anyone used this hierarchical name.

This will convert

```
Modelica.Blocks.Math.Sin sin1(n=1);
Modelica.Blocks.Math.Sin sin2(n=2);
Modelica.Blocks.Math.Gain gain1(k={4});
Modelica.Blocks.Math.Gain gain2(k={2,3});
```

To

```
Modelica.Blocks.Math.Sin sin1;
Modelica.Blocks.Math.Sin sin2[2];
Modelica.Blocks.Math.Gain gain1(k=4);
Modelica.Blocks.Math.Gain gain2[2](k={2,3});
```

Simplifying the result of convertModifiers

In some cases the result of conversion will be a large expression involving several package constants/enumerations. This can be simplifying by adding a fourth argument with value true to convertModifiers. An example will be converting initType for the rotational inertia of Modelica Standard Library version 2.1 to fixed-modifiers for the attributes (and replacing the phi_start by a start-value modifier):

```
convertModifiers("…Inertia",{"initType"}, {"phi.fixed=
if(%initType%==…InitialState or %initType%==
…InitialAngle or +%initType%==…InitialAngleAcceleration or
%initType%==…InitialAngleSpeedAcceleration) then true else
false",
"w.fixed=if (%initType%==…SteadyState or
%initType%==…InitialState or %initType%==…InitialSpeed or
%initType%==…InitialSpeedAcceleration or
%initType%==…InitialAngleSpeedAcceleration) then true else
false",
"a.fixed=if (%initType%==…SteadyState or
%initType%==…InitialState or
%initType%==…InitialAcceleration or
%initType%==…InitialAngleAcceleration or
%initType%==…InitialSpeedAcceleration or
%initType%==…InitialAngleSpeedAcceleration) then true else
false"}, true);
convertModifiers("…Inertia",{"phi_start"},{"phi.start=%phi_star
t%"});
```

The additional true-argument ensures that the result of the conversion of

```
Inertia inertia(
 initType=Modelica.Mechanics.Rotational.Types.Init.SteadyState,
 phi_start=1);
```

is just

```
Inertia inertia(
    a(fixed=true),
    phi(fixed=false, start=1),
    w(fixed=true));
```

Note that the simplification applies both to the expressions that are part of convertModifiers, and also to the original expression. This explains why the conversion is not the default, but it is advisable for large expressions.

Removing modifiers with convertModifiers

In some cases the modifier is just removed. This can be handled by converting the modifier to an empty list of modifiers. In some cases this applies to a common class where it would be

tedious to explicitly list all conversion where the class is used, and in those cases one can use a fourth argument to convertModifiers to force the conversion to be applied in other cases.

```
convertModifiers("…RealInput",
                 {"SignalType"}, fill("",0), true);
```

In this case all SignalType modifiers for RealInput components are removed.

convertClear

The command convertClear() clears the translation tables.

16.1.3 How to build a convert script

In order to convert a model using one library to another it is recommended to begin constructing a conversion script for the library. Even if the script is not complete after converting one model it can be reused for the next model, and only amend it with additional lines for the additional library components in that model. If a library developer restructures a Modelica library, it is recommendable to construct such a script.

Below it is explained how to construct a convert script and how to amend it for additional models. For clarity assume that we would like to convert a drive train model, myOldModel, that uses components in the model library "Drive Trains" to a model, myNewModel that instead uses the components in the library Modelica.Mechanics.Rotational.

1. Copy myOldModel.mo to myNewModel.mo.
2. Make a local conversion script, say Convert.mos. Depending on what you are converting start as follows:
   a. If you have a script for converting similar models: Use that as starting point, and at the end add the following lines (and remove similar ones):

      ```
      openModel("myNewModel.mo");
      checkModel("myModel");
      ```

   b. If you are starting from scratch, use the following as a template:

      ```
      clear
      // Start
      //
      // End
      openModel("myNewModel.mo");
      checkModel("myModel");
      ```

3. Converting class names.
   a. In the Dymola main window select **File > Clear Log** and then use the command button **Simulation > Run Script** to run the script file Convert.mos.
   b. There will be error messages such as `Error: Component type specifier Shaft not found.` Go through these entire messages and list all model types that are missing.

c. For each missing type find the new one in Modelica.Mechanics.Rotational. You can do that by opening both libraries and comparing icons and reading documentation. For component Shaft, we select Modelica.Mechanics.Rotational.Inertia.

d. Use a text editor to edit Convert.mos with a contents as

```
clear
// Conversion of not found Component type specifiers
convertClass("Shaft",
  "Modelica.Mechanics.Rotational.Inertia");
 convertClass("Clutch",
  "Modelica.Mechanics.Rotational.Clutch");
convertClass("Gear",
  "Modelica.Mechanics.Rotational.IdealGear");
//
openModel("myNewModel.mo");
checkModel("myModel");
```

4. In the Dymola main window select **File > Clear Log** and then use the command button **Simulation > Run Script** to run the script file Convert.mos.

5. Error messages saying

```
Use of undeclared variable shaft1.pDrive
Use of undeclared variable shaft1.nDrive
```

indicate that the connectors of typical components have changed name. To fix that we include in Convert.mos before openModel("myNewModel.mo");

```
convertElement({"Clutch", "Shaft", "Gear"},
                "pDrive", "flange_a");
convertElement({"Clutch", "Shaft","Gear"},
               "nDrive", "flange_b");
```

6. Error messages of the type

```
Error: Modifier 'fnMax' not found in Clutch.
```

indicate that a parameter has changed name. In simple cases when a simple renaming works use convertElement. Otherwise use convertModifiers.

7. Run the updated Convert.mos file (when asked if update myModel in the file myNewModel, answer No)

8. Keep a copy of the conversion script, Convert.mos, since it can be useful for converting similar models.

9. Now save the model.

In some rare cases it might be necessary to edit the model by hand or it is necessary develop model wrappers or a new model component.

Note, that default values for parameters are not translated. For example, if there is a model component m1 that has a parameter p declared as

```
parameter Real p = 1.0;
```

and the new model component also has a parameter p declared as

```
parameter Real p = 0;
```

then the old default value of 1.0 is lost and the new one being zero is used. If it is important to preserve the old parameter default values, this can be done by making a new model component that extends the new "m1" and modifies its parameter values according to the old m1.

16.2 Upgrading to new version of Modelica Standard Library

16.2.1 Introduction

This user guide describes how to upgrade your own models and libraries to use the new version of Modelica Standard Library.

The upgrading of a model or a library to use a new version of Modelica means renaming of classes and components. Dymola provides an automatic procedure based on scripts specifying the name conversion rules. When upgrading your model library, Dymola also automatically creates such a conversion script which in turn will be used when you convert models or other libraries building upon your library. The automatic conversion feature supported by Dymola allows a user to have automatic conversion if for example names of classes or components are changed.

16.2.2 Basics

Before describing the conversion procedure we need to describe some basic issues

- Naming of versions of libraries
- Which versions of other libraries to use in a model or a library
- Specifying default version of the Modelica Standard library

Naming of versions of libraries

The upgrading and handling of versions of libraries builds on the standardized Modelica annotation "version" specifying a version name for each Modelica package. The version names are hierarchical and not decimal, thus the order is e.g. "1.0", "1.1", "1.1.1", "1.2", ..., "2.2", "2.2.1", "2.2.2", "3.0", …,"3.2.3", "4.0.0"). For non-default versions the directory name is given by appending space and the version name to the library, e.g. version 0.5 of MyLibrary is stored in the directory "MyLibrary 0.5".

Modifying this is described in "Specifying the version of a package and checking of conversion scripts" on page 1533, and it also modified every time you upgrade the library, "Upgrading to a new Modelica version" on page 1528.

Which versions of other libraries to use in a model or a library

To ensure that the automatic loading feature of Dymola loads consistent libraries, each package has an annotation "uses" to specify which versions of referenced libraries it builds

on. This information is also exploited when upgrading models from using one version of a library to another.

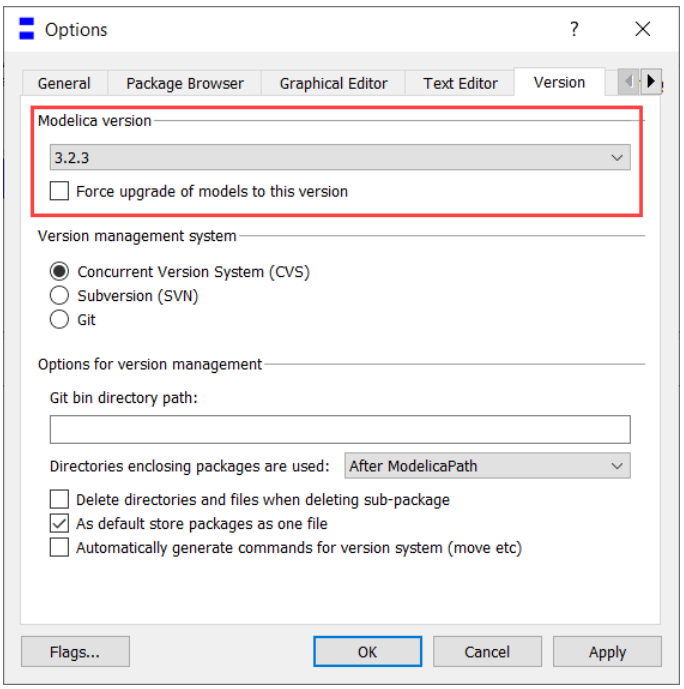
The version information is handled automatically by Dymola in most cases. The design has been chosen to make it easy to maintain in a simple way. Version numbers respectively uses-information is only stored once per package and apply to the entire package.

Specifying default version of the Modelica standard library

The specification of which version of the Modelica standard library to use influences the loading of all other libraries since they should be consistent with the Modelica library. It can be specified according to the following rules

1. The basic rule is that the loading of the first model or library using the Modelica library specifies which version of Modelica to use.
2. There is a possibility to force an automatic upgrading of a model or library to a certain version by using the **Versions** tab in **Tools > Options**:

Conditional use of Modelica 3.2.3

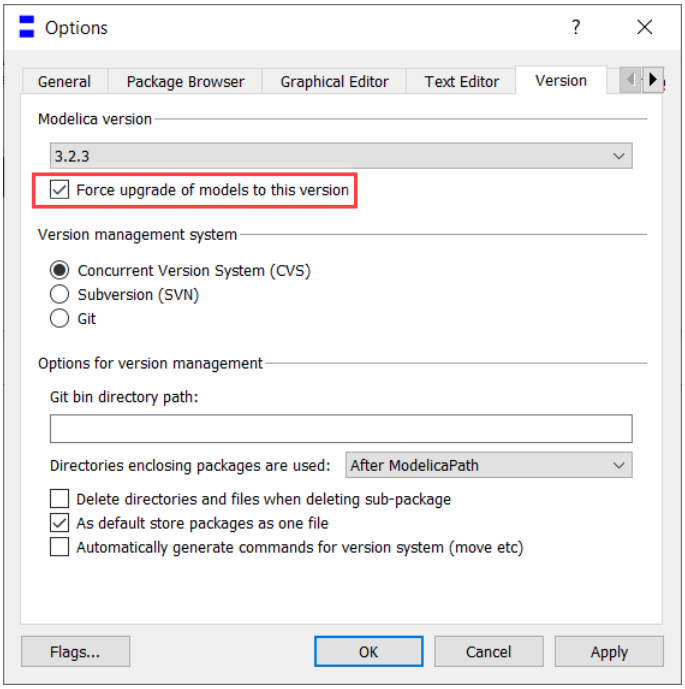


Ticking **Force upgrade of models to this version** means that loading e.g. a library that uses Modelica 3.0 will force that library to be upgraded to use Modelica 3.2.3. If another version of the Modelica library is already loaded, you will be asked to use the command **File > Clear All**, which unloads all libraries.

If you modify the version (if possible, depending on the Dymola version used), click **OK**. The setting will be saved for the next time you run Dymola.

3. Setting the default version to a certain version and ticking **Force upgrade of models to this version** will also prevent automatic conversion to another version. The following setting will lead to an unconditional use of Modelica 3.2.3:

Unconditional use of Modelica 3.2.3



Your models will then automatically open the correct Modelica library and related libraries. In case you later start to upgrade some of your libraries to a later version of Modelica they will continue to use the old version provided you follow the following rule:

- Use default names for libraries (i.e. the ones suggested by Save).

This rule also ensures easy handling of the library in general.

16.2.3 Upgrading to a new Modelica version

The general procedure

Specifically, if upgrading from Modelica 2 to Modelica 3, see also MigrationModelica2.pdf. Note that there are limitations which Dymola versions support conversion from Modelica 2 to Modelica 3, see section “Dymola version limitations when upgrading Modelica 2 to Modelica 3” on page 1531.

The procedure to upgrade models to a new Modelica version is in principle simple to perform provided some conditions are fulfilled. In order to have smooth upgrading, it may be useful to first consider the following issues

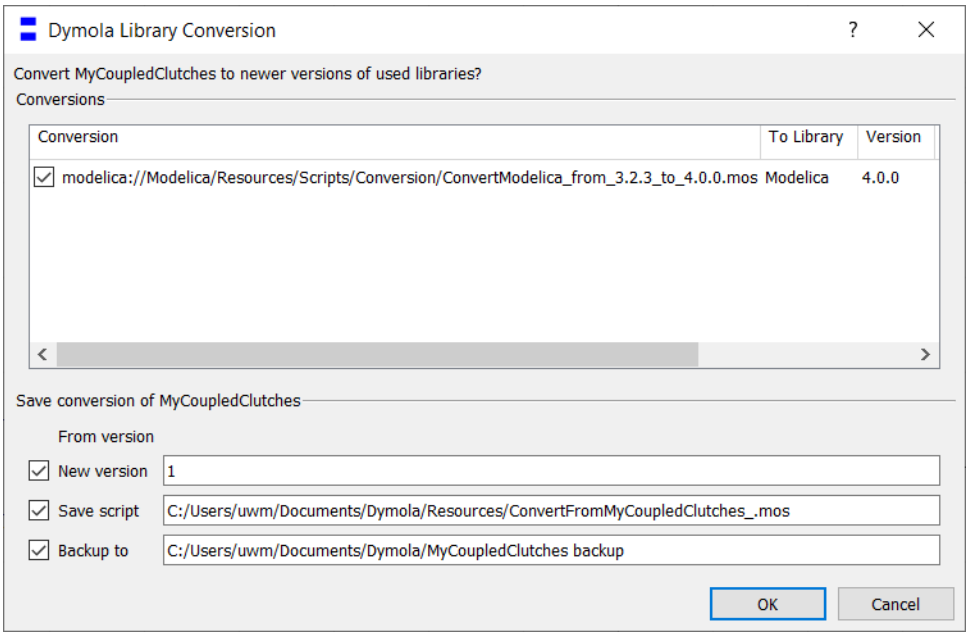
1. Perform **Check** on the library before conversion, and verify that test-examples are consistent.

2. Save the library after **Check**. This automatically updates the uses-information.
3. Install updated versions of all libraries used by this library. For commercial libraries, please, contact your distributor for a new release. For non-commercial and in-house libraries please contact the maintainer.
4. The conversion procedure is easiest performed and most reliable if the models are stored in the standard way. Packages may be stored in one file having the same name as the package and .mo extension. Packages may also be stored using hierarchical directories. It is not advisable to upgrade files produced by the **File > Save > Save Total…** command, because they include typically several packages and a model. This is discussed further below in the section "Using old models after upgrading to the latest Modelica version" on page 1532.

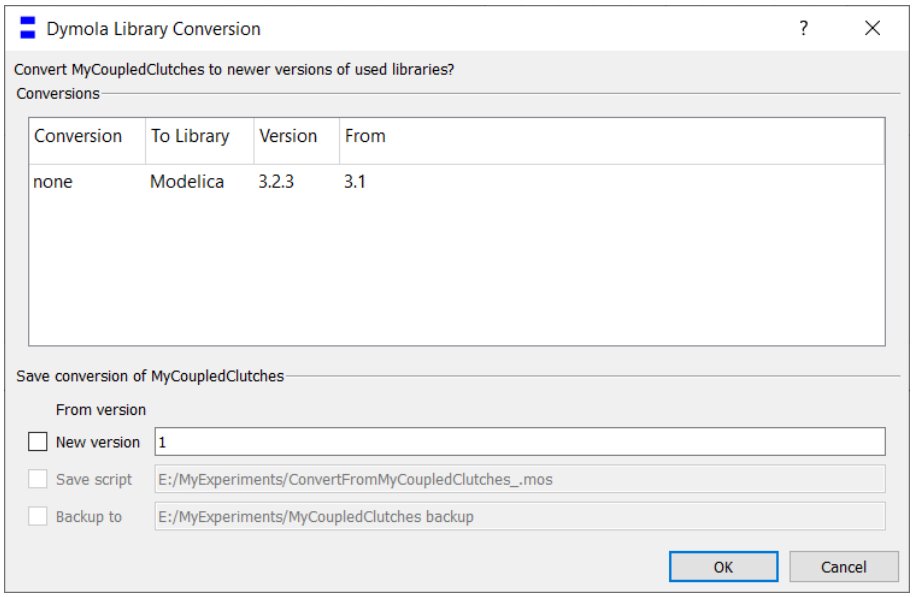
The conversion procedure is as follows:

1. Start Dymola
2. You should ensure that either you have loaded the wanted Modelica version or that this version is the default version and **Force upgrade of models to this version** is ticked.
3. You open the model or library you would like to upgrade. Dymola will prompt you about the upgrading. (The reason for illustrating this with two images is to show the current Dymola dialog, but also an older version where a conversion script was not applied due to not being conversion between different main Modelica versions.)

Conversion including version information – current Dymola 2023



Conversion including version information – older Dymola version



During the conversion Dymola prompts

a. for a version number of the converted library

b. (if relevant) if and where to store the conversion script which contains information relevant for users of your library, e.g. if the renaming of library components caused a change in the name of any component in your library

c. (if relevant) about storing backup of the original

We recommend that you accept the defaults proposed by Dymola. You may of course give another version number. The applied conversions are written to the Syntax Error log. It may happen that you do not get any prompt for conversion when you open the package. This is typically the case when there is no uses annotation for the Modelica library in your package. Dymola then does not detect uses of the Modelica library in sub-packages.

After conversion

1. Re-check the model or the library, and verify that test-examples are consistent.
2. Update documentation to take into account the new structure.
3. Distribute the library (including conversions) to other users and convert other libraries. You should also consider distributing the automatically generated backup for users who want to continue using the older version.

Conversion of mutually dependent libraries

You should in general first convert a library and verify this library in itself and then convert libraries and models using this library. In some cases this is not possible.

The most problematic case for conversion is if you have two (or more) mutually dependent top-level libraries, neither of which specifies any version or uses-information.

To handle this case we recommend the following procedure:

1. Update the libraries with version and uses-information. (There will otherwise be an unnecessary conversion step.) For giving version number see section "Specifying the version of a package and checking of conversion scripts" on page 1533. Additionally save all libraries to ensure that the uses-information is correct.
2. Ensure that all libraries are stored with default names without version number, e.g. for MyLibrary either as a file MyLibrary.mo or as a directory MyLibrary with a file packge.mo inside. This enables demand-loading of the libraries which is needed for the next step.
3. Trigger conversion of one of the libraries, by opening it. This will read the other libraries, and you will be prompted to save new versions of all of them.

Dymola version limitations when upgrading Modelica 2 to Modelica 3

Dymola 2018 FD01 and newer do not support Modelica 2. Conversion from Modelica 2 to Modelica 3 is still supported in Dymola 2018, but it is easier to verify the conversion in a previous version of Dymola, e.g. Dymola 2015, where Modelica 2 is still fully supported. For more information about conversion from Modelica 2 to Modelica 3, see MigrationModelica2.pdf.

Potential problems with conversion

The conversion procedure has been tested with several libraries and the conversion handles large and advanced libraries. However, in some advanced cases the command **Check** might detect incompatibilities. This is either because the conversion of some library is incomplete or because the required conversion is too complex. In some cases it is easy to determine the problem and correct the partially converted model. If not, please contact Dassault Systèmes AB and/or the distributor of the used libraries.

An example of too complex a conversion is a block-source that can be either scalar or vector (conversions assume this is a scalar). Later hierarchical modifiers can then turn this into a vector-source. For this case the block-source should have been automatically converted into a vector of blocks.

Reverting to old versions

In case you want to revert to the backup of the original library and are not using a version handling system you can copy all of the files from the backup directory to the original directory (overwriting the new files).

Another possibility is to remove the converted files and then rename the backup directory by removing everything in the directory name starting from the first space, e.g. "MyLibrary 0.8" or "MyLibrary backup" is renamed to "MyLibrary".

Updates of conversions

This user guide centers on the handling of the upgrade of the Modelica Standard Library. However, this conversion procedure is not unique to this conversion of Modelica but also works for other conversions, e.g. MyLibrary 0.8 to 0.9 and for any conversions of your own libraries. For changes that are not directly caused by upgrades of other libraries it requires that the library maintainer also provides a conversion script.

When upgrading a library that already has conversion scripts, the existing scripts will also automatically be updated (by appending the new script to the old one), and none-conversions handled in an appropriate way. The updated conversion scripts should be seen as a fall-back strategy and the recommended procedure is always to keep all libraries and models consistent with the latest version of libraries.

16.2.4 Using old models after upgrading to the latest Modelica version

Dymola can only have one version of each library loaded at the time. It allows you to use old, non-upgraded models. You have the following possibilities

1. Manually load an old version of Modelica before loading the model.
2. Load the model directly in Dymola, i.e. before you open the Modelica library, and ensure that **Force upgrade of models to this version** (as explained above) either is un-ticked or is ticked for the version you want to use.
3. Use the command **File > Save > Save Total...** of the model before using either of the approaches 1 and 2 above. A save total model using components from the Modelica Standard library includes a shortened version of the Modelica library including the components needed. Loading a save total model means an explicit loading of the Modelica library as in alternative 1.

16.2.5 Determining what libraries a model use

In most cases the uses-information is automatically updated by Dymola, when you use models of library in another library:

- Drag'n'drop a component from one library to a model in another.
- Extend from a class in one library and place it another library.
- Duplicate a class.

This handles the graphical uses, and if you write models textually there are number of alternatives:

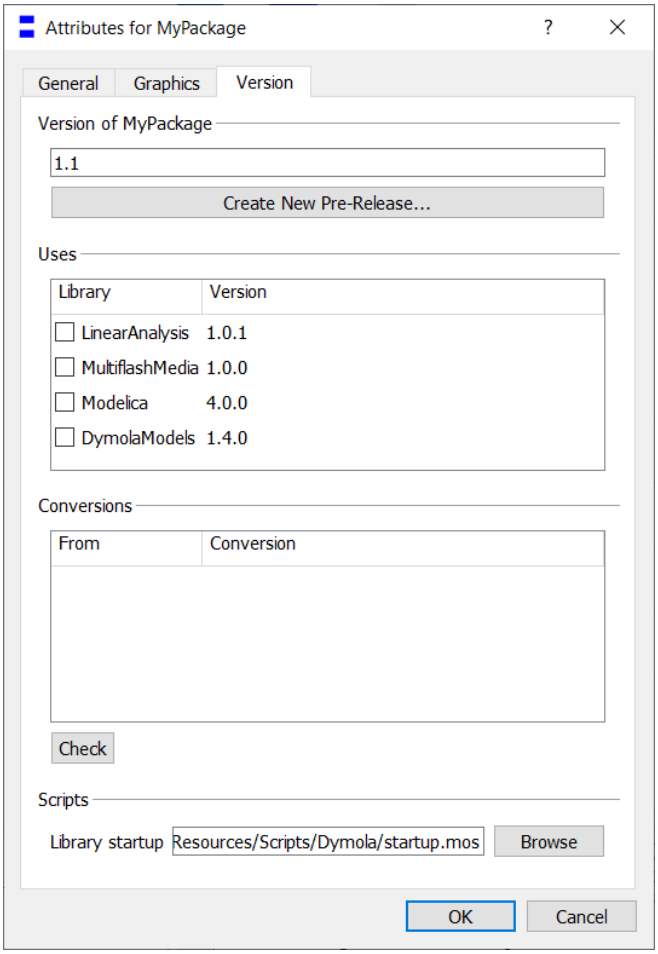
- Edit the version and uses-information using the **Version** tab in the command **Graphics > Attributes**. This is described in the next section.
- Write the uses-annotation textually.
- Save the model or library, since Dymola automatically adds uses-information (provided the used library has a version number, and is opened in Dymola).

16.2.6 Specifying the version of a package and checking of conversion scripts

General

For a top-level package or model you can edit the version information from **Graphics > Attributes**, in the **Version** tab.

Version tab



This dialog also allows you to edit the uses-information by selecting which libraries a package uses. All existing uses-annotations are included and checked, and all loaded non-used packages with version information are included as non-checked.

If the uses-annotation specifies a package that is not loaded you can also edit which version is used.

For models and/or packages inside another package the version information is read-only. The version information is also shown in the Documentation layer.

Available conversion scripts are displayed. Selecting a conversion script and clicking the **Check** button opens a browser for selecting the package to convert from. When this package is selected, a check of the conversion script is performed, and the result is displayed.

If the old library is converted to additional libraries you should ideally open them before checking the conversions, e.g. Modelica normally converts some parts to the ObsoleteModelica library.

When prompted for the old library you will first be prompted for the old library (including the version number). If other libraries were merged into the library, you will also be prompted for them (without version number).

Selecting these libraries are optional.

For information on how to handle library startup scripts, that is, the Scripts group, see the documentation of the command **Graphics > Attributes**, the **Version** tab, see the index entry "*Graphics tab : attributes*".

Working with library pre-release, including saving conversion script to file

You can create a pre-release of a library, and then convert from pre-release to released package, with an option to save the conversion file in a selected location.

The workflow to create a new version from, for example, MyPackage version 1.1 is:

- In the package browser, right-click MyPackage and select **Attributes**, the **Version** tab, and click on **Create New Pre-Release** to start working on the next release by setting the version to 1.2 Working (or something similar, the suggestion by default is 1.2 Intermediate).
- Ensure that **Tools > Options**, the **Package Browser** tab, have **Yes, also conversion scripts** for the two rename settings.
- Rename classes and components, move them around etc.
- (Optional: If you realize that the changes were larger than anticipated, you could change it to 2.0 Working.)
- (Optional: If it is getting close to release, you could change it to 1.2 Alpha.)
- In the context menu for MyPackage in the package browser, select **Attributes**, the **Version** tab, and click on **Convert from Pre-Release…**. A dialog is opened where to save the conversion script.
- When you have decided where to save the conversion script (or not to save it), you can click **OK** in the dialog to create the new version. Note that the version is automatically updated in the dialog.

16.2.7 Upgrading models and libraries to a new library version

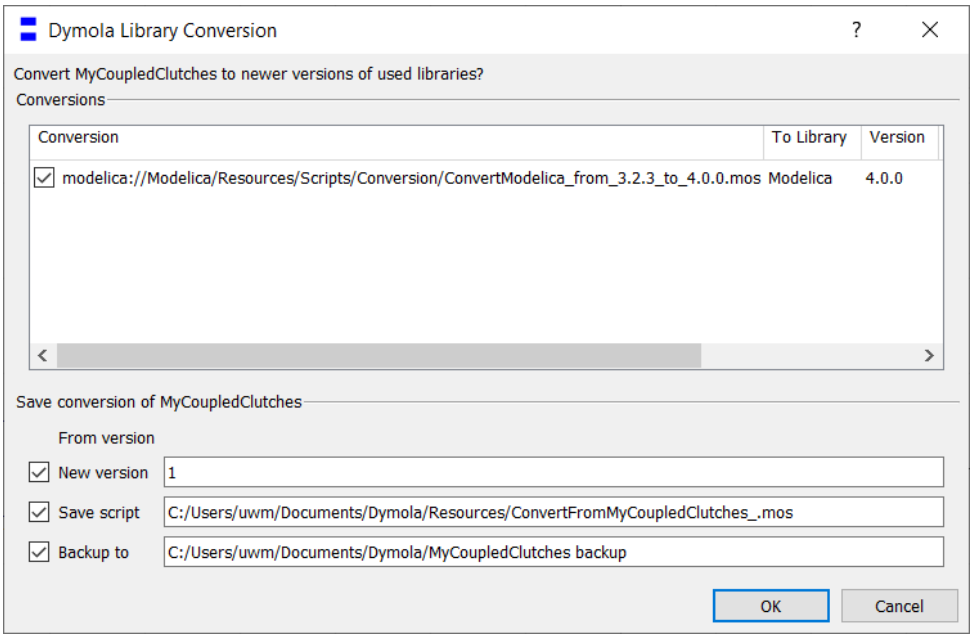
We are here providing a reference for what happens at conversion, including some of the minor details of the conversion.

If you want to force to a specific library you should start by loading that version of the library. You then load the model(s) or library you want to upgrade and will be prompted to upgrade them. The applied conversions will be written to the Syntax Error log.

In some cases there will be warnings for conversions that are too complex or the converted library does not pass through check, please consult the library documentation in that case.

In addition to converting you will be asked for a new version number, to save the conversion script for users of the library, and create a backup of the original. The conversion script contains information relevant for users of your library, e.g. if the renaming of library components caused a change in the name of any component in your library.

Conversion including version information



Thus models using your library will automatically upgrade to the new version.

You can also add your own conversions to this script.

In some cases the automatically generated conversion script will be empty - in this case the reference to the script will be replaced by a none-conversion, and the conversion script will contain a comment saying that it is not used.

The default for the new version is that the least significant version number is increased by one. Dymola will also store the old version as backup and after backing up the files save the converted library at the original place. To avoid problems it is verified that the backup directory is non-existent, empty, or already contains a backup of the Modelica files and scripts.

If the model is read-only and no conversion is needed Dymola will silently accept the model as being compatible with the library.

The dialog contains information about what scripts are used for which conversion. De-selecting a conversion script should only be done if you the conversion have already been performed (and for some reasons the uses-information was not updated).

If the need for upgrading is found during the translation of a model the translation will be interrupted, you will be prompted to convert, and the translation will then re-start.

If you have loaded one of several mutually dependent libraries, you will be asked to convert all of them at the same time and there will be several 'Save conversion' groups (but only one set of conversions).

Note that pressing **Cancel** cancels all conversions and until you exit Dymola or use the command **File > Clear All** and re-open the files, and you should perform one of these actions after pressing **Cancel**.

Note also that Dymola supports multiple conversions of libraries. A typical case is that a customer does not upgrade to each new Dymola version, but "jumps" a certain version. When upgrading to the present Dymola version in this case, certain libraries may demand to be upgrading to the "jumped" version before upgrading to the present version. Dymola handles this by executing more than one upgrading script in the same operation.

As an example, consider MyLib that should be converted from 1.2. to 1.6. That can be done the following, considering the corresponding scripts exist and work:

```
annotation(conversion(
  from(version="1.2", to="1.3", script="modelica://MyLib/scripts/
    Convert_from_1.2_to_1.3.mos"),
  from(version="1.3", to="1.4", script="modelica://MyLib/scripts/
    Convert_from_1.3_to_1.4.mos"),
  from(version="1.4", to="1.5", script="modelica://MyLib/scripts/
    Convert_from_1.4_to_1.5.mos"),
  from(version="1.5",script="modelica://MyLib/scripts/
    Convert_from_1.5_to_1.6.mos")));
```

What will happen (which differs from just having multiple scripts) is that these conversions will be run in sequence as completely separate conversions, i.e. 1.2->1.3, 1.3->1.4, 1.4->1.5, 1.5->1.6; thus if you change the name of a class between 1.2 and 1.3 and then rename parameters between 1.3 to 1.4, and then rename both the parameter and the class between 1.5 to 1.6 it should work. (Each conversion script may have renaming of classes and parameters in arbitrary order.)

An exception is if any intermediate library version requires manual conversion; which is handled by converting models to a deprecated library. This deprecated library may be missing in, or may be incompatible with, the newer Dymola version.

16.3 Preparing libraries for migration

If libraries are to be distributed (e.g. commercial libraries), they should be prepared for migration when e.g. a new version of Modelica Standard Library is released.

Dymola supports the Modelica annotations for version handling, including the use of the Modelica URI scheme 'modelica://' for specifying the location of conversion script files in the conversion annotation.

Please see Modelica Language Specification for further details concerning this. The specification can be found in the Modelica site; www.Modelica.org.

16.4 Updating Modelica annotations

General

The built-in function `updateModelicaAnnotations` can be used to update Modelica annotations, in particular to remove deprecated annotations for text-primitives. The function is defined as:

```
function updateModelicaAnnotations "update Modelica annotations
  to follow the standard and preserve behavior"
input String className;
input Boolean changeLinePattern := true
  "Linepattern.none->LinePattern.Solid";
input Boolean orderBitMapExtent := true
  "Order Bitmap extent";
input Boolean removeTextDeprecated := true
  "Remove deprecated text annotations";
input Boolean renameTextColor := false
  "Text.lineColor->Text.textColor";
input Boolean checkThickness
  "Check if line thickness attribute is used";
output Boolean ok;
end updateModelicaAnnotations;
```

Some comments about some inputs:

The input `orderBitmapExtent` is used to allow images to be created as flipped.

The input `removeTextDeprecated` removes the following annotations for text-primitives: `fillColor`, `pattern`, `fillPattern`, and `lineThickness`.

For the input `renameTextColor` see below.

As an example, annotations can be updated (with default settings) for a library `MyLibrary` by executing `updateModelicaAnnotations("MyLibrary");`. The library must be loaded before giving the command.

Handling the textColor attribute

The attribute `textColor` was introduced in the Modelica Language Specification 3.3 in text annotations. You could still use the attribute `lineColor` in text annotations, but that use was stated as to be deprecated. In Modelica Standard Library (MSL) version 4.0.0, this is implemented.

To be able to select how to handle these attributes when, for example, wanting to work with older models, libraries, and MSL versions, there is in Dymola 2020x a new Boolean flag `Advanced.Editor.NewTextAttributes` available.

The flag is by default false, meaning that existing attributes `lineColor` in text annotations are preserved, but for MSL 4.0.0 and later, attributes `textColor` are created.

If you set the flag to true, the attribute `textColor` is always used in such cases.

You can automatically convert `lineColor` to `textColor` in text annotations for a library, use (replace "Modelica" by the name of your library):

```
updateModelicaAnnotations("Modelica",
removeTextDeprecated=true, renameTextColor=true);
```

Handling scaling of thick lines

From Dymola 2022x, the scaling of lines with line thickness > 0 (that is, line thickness > "single") is improved. The result is:

- You can now use thick lines in icons instead of rectangles.
- The rendering of thick lines when you, for example, use a zooming of 25 % in the diagram layer, is improved.

You can check if your models use the line thickness attribute in any icon drawing primitives, where it is most likely to have an impact. However, it is the expectation that the new rendering is an improvement. You can check a model by running the built-in function

```
updateModelicaAnnotations("MyModel", checkThickness=true);
```

where `MyModel` is the name of your model. Both the attributes "thickness" and "lineThickness" are checked. Notes:

- The tested model must be writable.
- Only the icon layer is tested.

17 APPENDIX - INSTALLATION

17 Appendix — Installation

This chapter describes the installation of Dymola on Windows and Linux, and related topics.

The content is the following:

In section 17.1 **"Installation on Windows"** starting on page 1542 the installation on Windows is described, including installation of Dymola software, C compiler and license (sharable or node-locked). The sub-section "Additional setup" starting on page 1556 treats specific issues as installing Dymola as administrator on a computer that should be used by non-administrators and remote installation of Dymola. Finally, change of setup, removal of Dymola and installing updates are described.

In section 17.2 **"Installation on Linux"** starting on page 1581 the installation on Linux is described, in a similar way as the previous section. The sub-section "Additional setup" starting on page 1582 describes e.g. compilation of model code and simulation from the command line.

In section 17.3 **"Dymola License Servers on Windows"** starting on page 1585 the installation of a license server on Windows is described, as is the borrowing of licenses.

In section 17.4 **"Dymola License Server on Linux"** starting on page 1597 the installation of a license server on Linux is described, as is the borrowing of licenses.

In section 17.5 **"Utility programs"** starting on page 1600 a utility program for finding a host id on a computer is described.

In section 17.6 **"System requirements"** starting on page 1601 the hardware and software requirements/recommendations are listed.

In section 17.7 **"License requirements"** starting on page 1606 the license requirements for various features are listed.

In section 17.8 **"Troubleshooting"** starting on page 1611 the solution to various problems are described. It might be license file problems, compiler problems, issues with Simulink, change of language etc.

17.1 Installation on Windows

This section refers only to the Windows version of Dymola.

To install Dymola the following tasks must be performed:

- Install the Dymola software and libraries.
- Install a C compiler (if it has not been done before).
    - Install the Dymola license file.
    - Install a license server (sharable license only).

Following installation, the user may do additional setup. The installation of updates and removal of Dymola is also described below.

17.1.1 Dymola as 32-bit and 64-bit application

From Dymola 2018 FD01, the Dymola program is only available as a 64-bit application. The Dymola program (and its associated DLLs) is located in the folder `Program Files\Dymola 2023\bin64` after installation.

17.1.2 Installing the Dymola software

Dymola and appropriate libraries is distributed on a single DVD or downloaded electronically. With electronic download, the DVD-image is provided as two separate .zip files. **Note** that both zip-files must be extracted to the same location before starting the installation.

Starting the installation

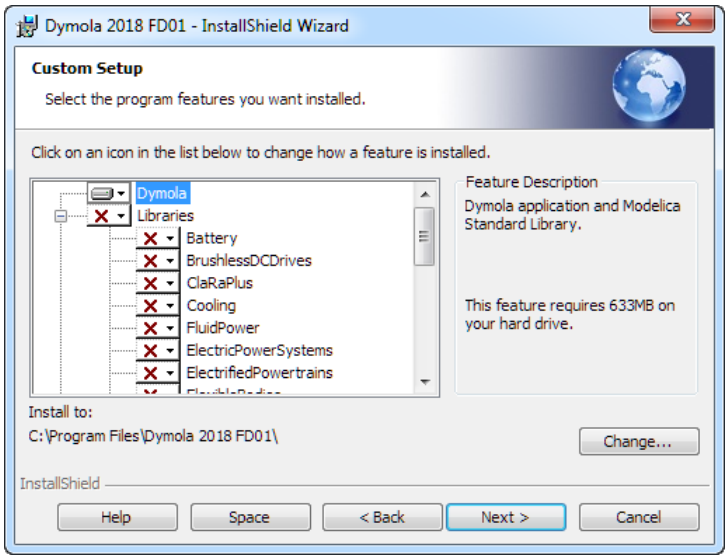
Please note that Administrator privileges are required for this installation. When Dymola has been installed, any user can run it.

The installation normally starts when you insert the distribution DVD. If autostart has been disabled, please start `D:\setup.exe` (assuming your DVD drive is labeled D) from Windows Explorer by double clicking on the file or use the **Start** button in Windows, select **Run**, enter `D:\setup.exe` and click **OK**.

Dymola installation setup.



Clicking **Next>** will display license conditions that must be accepted in order to proceed. Accepting by selecting that alternative and then clicking **Next>** will display the following:

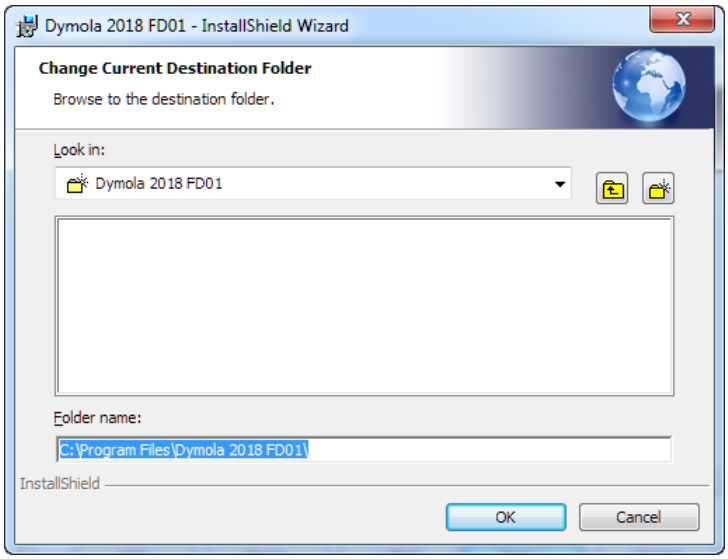


Location of installation directory

The first choice in the installation procedure is the type of installation and the name of the Dymola installation directory. The default is: `C:\Program Files\Dymola` + the version number of Dymola.

This path is displayed under `Install to:`. If the path should be changed, click on the **Change...** button. Here the path can be changed; a change has to be acknowledged by clicking **OK**.

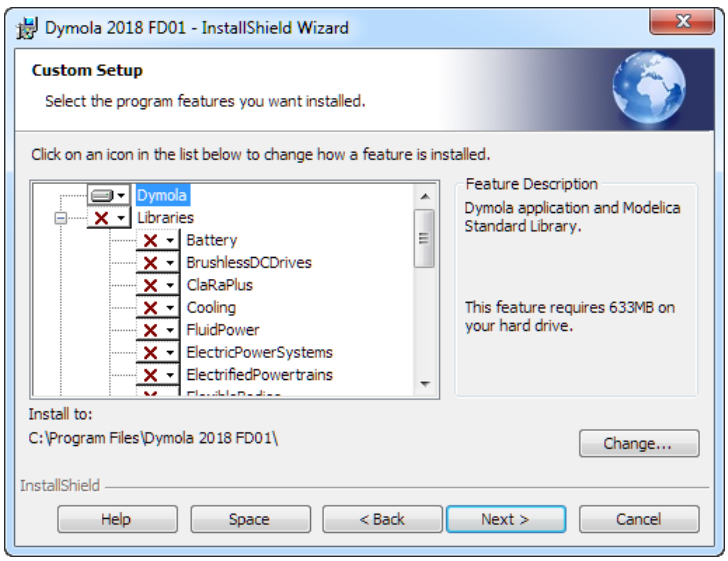
Dymola installation directory.



Selecting components

The second choice is to select optional components of the distribution. By unselecting components some space can be saved.

Component selection.

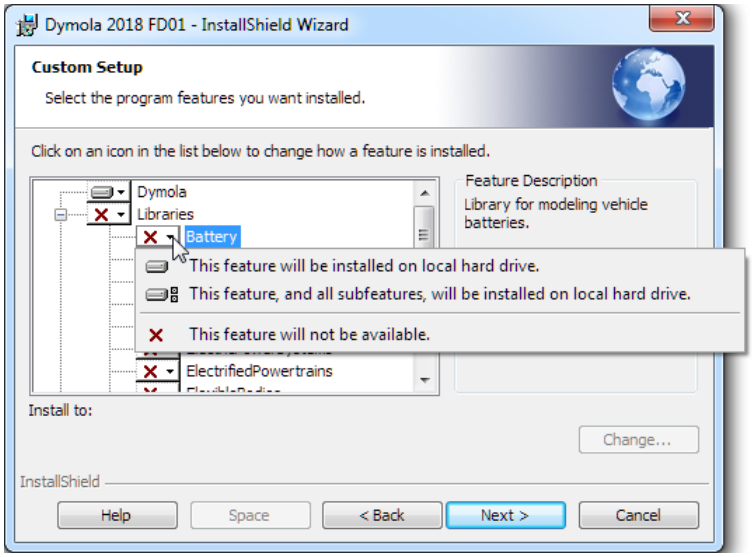


The first alternative **Dymola** is the default contents of the Dymola distribution, including the development environment and the Modelica standard library. This component should always be installed (except when only a license server should be installed).

The **Libraries** section contains several commercial libraries that require a license option to use. Install libraries according to your current options.

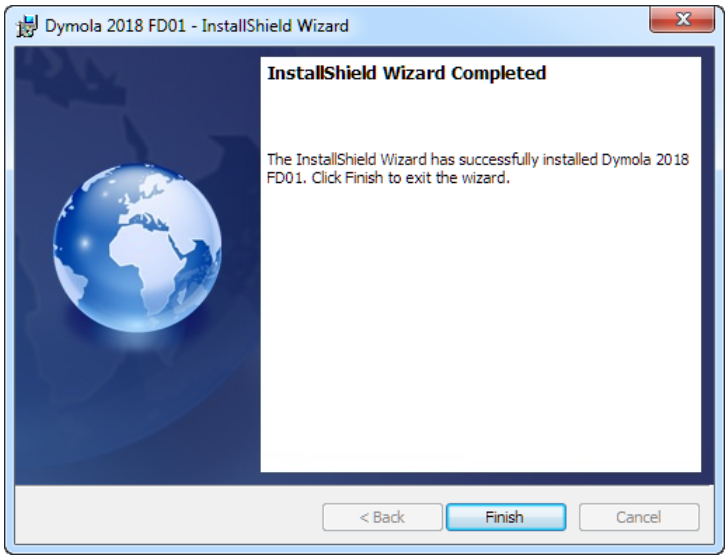
The last section, **License server**, makes it possible to install Dymola license server without having to install Dymola. Please note that the **Dymola** component should be unchecked in that case.

To add/remove a component from the installation, click on it and select the appropriate alternative in the menu.



When Dymola is successfully installed the following will appear:

Installation of Dymola has finished.



17.1.3 Installing a C compiler

To translate models in Dymola you must also install a supported C compiler. The C compiler is not distributed with Dymola. The C compiler needs to be installed only once, even if you install multiple versions of Dymola. You can select a Microsoft compiler or a GCC compiler. Furthermore, you can select to cross-compile for Linux on Windows.

Compilers

Microsoft compilers

Dymola supports Microsoft Visual Studio 2017 and Visual Studio 2019, the following editions:

Visual Studio 2017:

- Visual Studio Professional 2017
- Visual Studio Enterprise 2017
- Visual Studio 2017 Desktop Express **Note!** This compiler only supports compiling to Windows 32-bit executables
- Visual Studio 2017 Community
- Visual Studio 2017 Build Tools **Notes:**
    - The recommended selection to run Dymola is the workload "Visual C++ build tools" + the option "C++/CLI support…"
    - Installing this selection, no IDE (Integrated Development Environment) is installed, only command line features

- This installation is not visible as specific selection when later selecting the compiler in Dymola, the alternative to select is the same as for any Visual Studio 2017 alternative: **Visual Studio 2017/Visual C++ 2017 Express Edition (15)**.
- For more information about installing and testing this compiler with Dymola, see www.Dymola.com/compiler.

Visual Studio 2019:

- Visual Studio Professional 2019
- Visual Studio Enterprise 2019
- Visual Studio Community 2019
- Visual Studio 2019 Build Tools **Notes:**
  - The recommended selection to run Dymola is the workload "C++ build tools" + the option "C++/CLI support…"
  - Installing this selection, no IDE (Integrated Development Environment) is installed, only command line features
  - This installation is not visible as specific selection when later selecting the compiler in Dymola, the alternative to select is the same as for any Visual Studio 2019 alternative: **Visual Studio 2019/Visual C++ 2019 (16)**.
  - For more information about installing and testing this compiler with Dymola, see www.Dymola.com/compiler.

Dymola also supports older Microsoft compilers (Visual Studio 2015 Professional edition and Express for Windows Desktop edition, and Visual Studio 2012 Professional and Express edition). Note however that Visual Studio 2013 editions are not supported, due to the logistics of supporting multiple old versions for all solvers.

Note. When installing any Visual Studio compiler, make sure that the option "C++/CLI support…" is also selected to be installed.

For more information about compiler please visit

```
www.Dymola.com/compiler
```

where you find more detailed information about the compilers, including links to Microsoft's website. Note that you need administrator rights to install the compiler.

The C compiler can be installed before or after you install the Dymola. You can run Dymola and browse models, but to translate any model you must install the C compiler.

To get a small improvement of the simulation performance, you can activate the global optimization in the compiler, by setting the flag

```
Advanced.Define.GlobalOptimizations = 2;
```

before generating code. (The default value of the flag is 0.)

This flag works the same for all Visual Studio compilers. Note that although the simulation performance is somewhat improved setting this flag, the compilation of the code might take

substantially longer time for large models. The setting corresponds to the compiler command `/Og`.

Note. The Microsoft Visual C++ 2015 redistributable package is automatically installed.

Intel compiler

From Dymola 2022, Intel compiler is not supported.

GCC compiler

Dymola 2023 has limited support for the MinGW GCC compiler. The following GCC versions have been tested (hence, at least the versions in that range should work fine):

- 32-bit: MinGW GCC 5.3, 6.3, and 8.2
- 64-bit: MinGW GCC 5.3, 7.3, and 8.1

To download any of these free compilers, please visit

www.Dymola.com/compiler

where you find more detailed information about the compilers, including links for downloading. Needed add-ons during installation etc. are also specified here. Note that you need administrator rights to install the compiler.

Please note:

- To be able to use other solvers than Lsodar, Dassl, and Euler, you must also add support for C++ when installing the GCC compiler. Usually you can select this as an add-on when installing GCC.
- There are currently some limitations with GCC:
    - Embedded server (DDE) is not supported.
    - Support for external library resources is implemented, but requires that the resources support GCC, which is not always the case.
    - No support for Dymola runtime concept. Consequently, FMUs must be exported with the code export option enabled, to be useful. (The code export option means having any of the license features **Dymola Binary Model Export** or the **Dymola Source Code Generation**.) Note! When migrating to Modelica Standard Library (MSL) version 4.0, MinGW gcc versions older than 5 are not guaranteed to work for source code export.
    - For 32-bit simulation, parallelization (multi-core) is currently not supported for any of the following algorithms: Radau, Esdirk23a, Esdirk34a, Esdirk45a, and Sdirk34hw.
    - Compilation may run out of memory also for models that can compile with Visual Studio. The situation is better for 64-bit GCC than for 32-bit GCC.

In general, 64-bit compilation is recommended for MinGW GCC. In addition to the limitations above, it tends to be more numerically robust.

Linux cross-compiler (WSL)

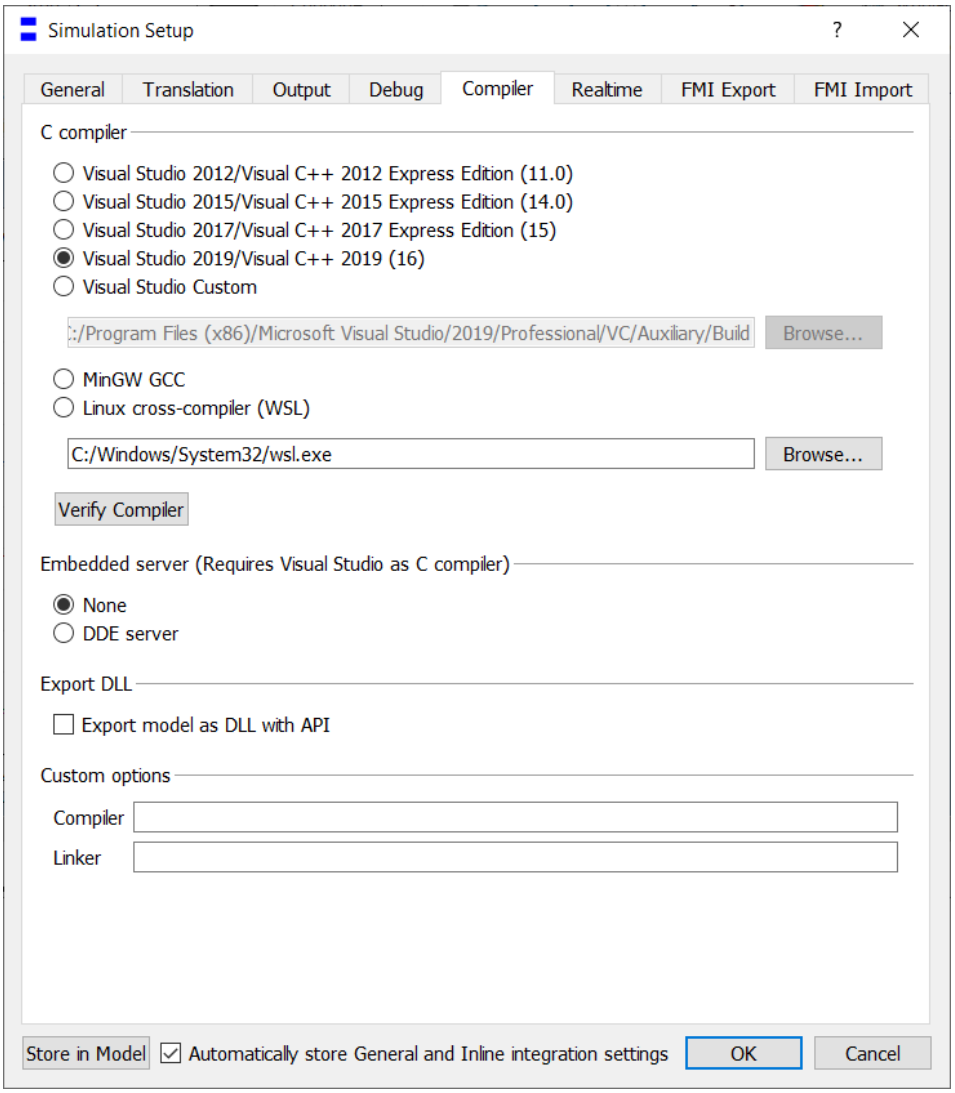
Dymola on Windows supports cross-compilation for Linux via the use of Windows Subsystem for Linux (WSL). The default WSL setup is 64-bit only and Dymola adopts this limitation.

For more information about the needed installations etc., please see "Cross-compilation for Linux on Windows" starting on page 1576.

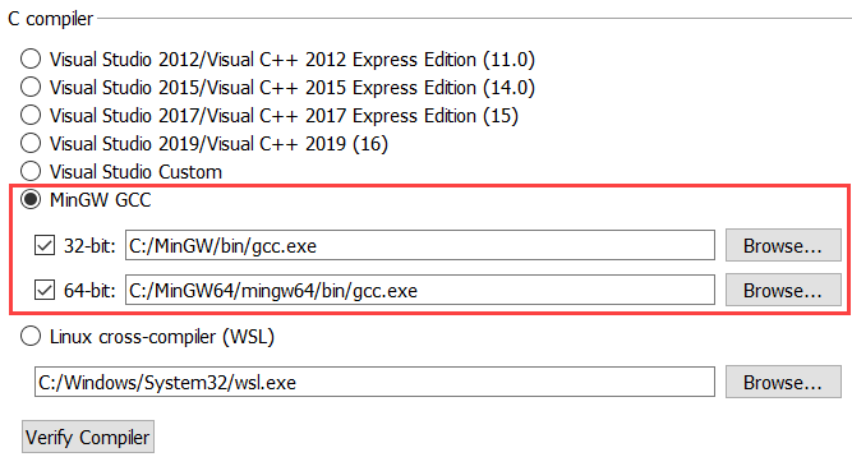
Selecting a compiler

Selecting compiler is required.

To change the compiler Dymola uses to translate the model, use the command **Simulation > Setup** and the **Compiler** tab, see also the index entry "*simulation setup : compiler tab*" in the index in the end of the manual. (Below is an example of the **Compiler** tab).



If you select the GCC compiler, subchoices for the compiler are seen:

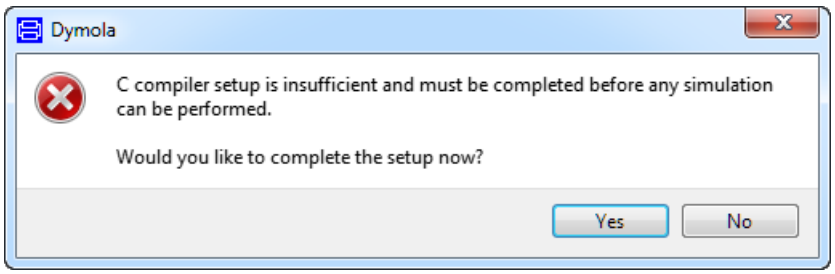


The selected compiler is stored as a per user setting and for the future kept for new installations of Dymola. Switching compiler does not modify Dymola/bin.

Advanced users can enter custom options for compiler/linker using the last two input lines in the Simulation Setup dialog.

Note the importance of the **Verify compiler** button. As an example, you cannot see from the menu if you have a valid MinGW compiler available, you must use the **Verify compiler** button to see if this is the case.

During the startup of Dymola, the compiler setting is checked. If insufficient, the following message appears:



If you click **Yes**, the menu for the compiler setup is opened to let you complete the setup.

Classes which contain “Library” annotations to link with external libraries in C are supported for Microsoft Visual Studio compilers. If you link with your own C-libraries, you have to recompile them as multi-threaded; Dymola only supports linking with multi-threaded libraries in Microsoft Visual Studio compilers. For GCC compilers, see the limitations above.

For information about possible compiler problems, please see the troubleshooting section “Compiler problems” on page 1613.

17.1.4 Installing the Dymola license file

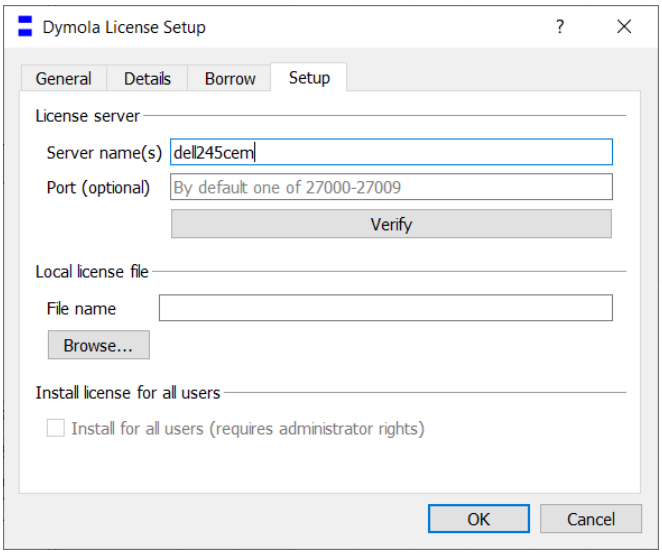
After installation, Dymola will initially start in “trial” mode. While running in trial mode you can continue with installing the license file.

Setting up a sharable license

Sharable licenses are requested by Dymola from a license server. The information normally required on the client computer is just the name (or IP number) of the license server. (For handling of the license server itself, usually handled by IT administration, see dedicated sections about license servers later in the document.)

Start Dymola and select **Tools > License Setup** and then the **Setup** tab. Enter the name or IP number of the server. If so instructed by the system administrator, also enter the port number. By default, leave this field empty.

License server setup.

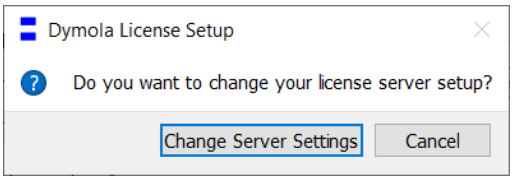


If you want to use redundant servers, you can add three server names/IP numbers, separated by space. Note that one or three servers must be specified.

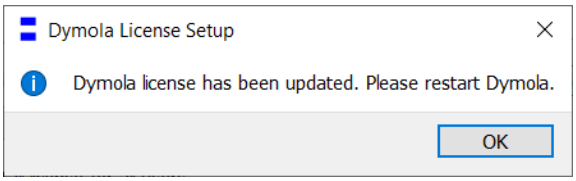
You have the option of installing the license file only for the currently logged in user, or for all users on this computer. The latter requires administrator rights.

To test the selected server before changing to it, you can click **Verify**. (Note that **Verify** also works for the present server, if you have any.)

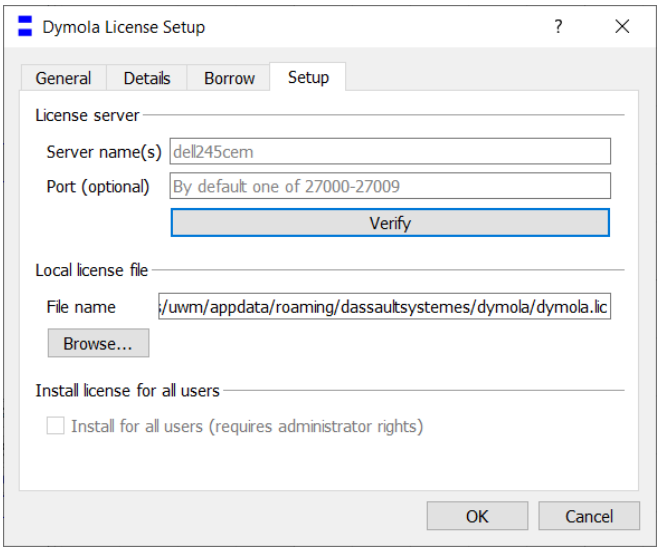
If the server you selected server is ok, and you want to select to use it, click on the **OK** button. Dymola will ask for confirmation before overwriting your old license information.



After changing the license server setup, you must restart Dymola to use the new server.



Note that when looking at the **Setup** tab when having a valid license, you get information about the current license server, as well as the location of the local license file used:



(You can of course still type in a new server name.)

If the server name cannot be detected, or if you have a nodelocked license, then nothing is shown in the server name field.

For information about diagnostics when connecting to the license server, see dedicated sections about license servers later in the document.

Installing a node-locked license

Node-locked licenses are stored locally on the computer running Dymola and are not shared with other computers.

Obtaining a host id

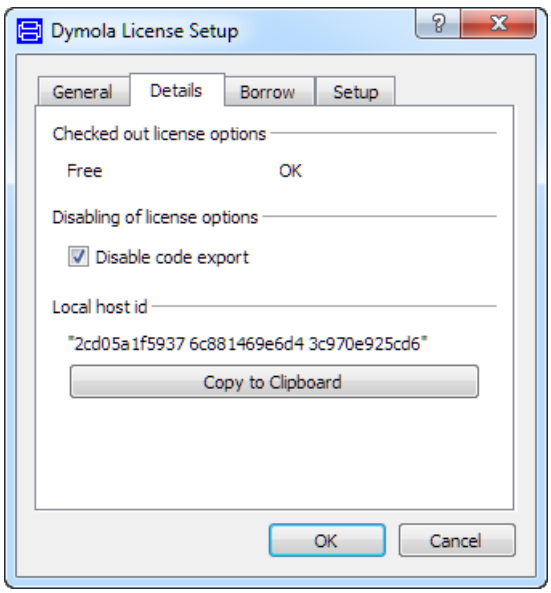
To order a *node-locked license key*, the relevant host id of the computer where Dymola should run must be supplied to your Dymola distributor. The license that you will receive will contain this information.

There are two ways finding out this host id, depending on whether a Dymola trial version is installed before or not. The host id can always be fond using the utility program `hostid.exe`. Please see section “Obtaining a host id” on page 1600 for more information about this program.

If the Dymola trial version has already been installed, Dymola can be used to find the host id. Start Dymola and select **Tools > License Setup**, and then the **Details** tab. Click on **Copy to Clipboard** to copy the local host id.

Please note that some laptops present different host id´s depending on whether they are connected to a docking station or not. In such a case, please copy all host id´s.

Local host id of the computer running Dymola.



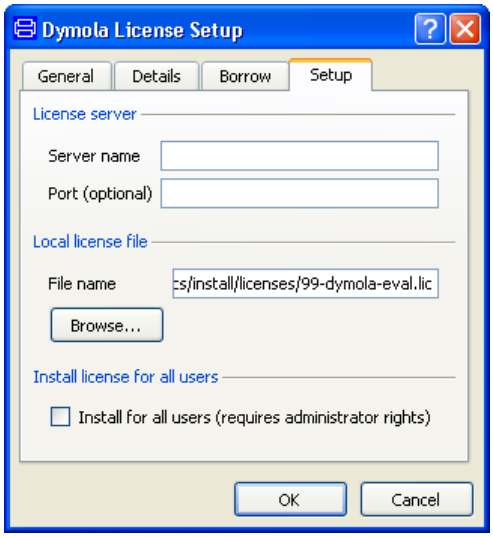
Compose an e-mail containing your local host id (host id´s) and send it to your Dymola distributor.

Installing the node-locked license

When you have received your license file, do save the license somewhere on your computer.

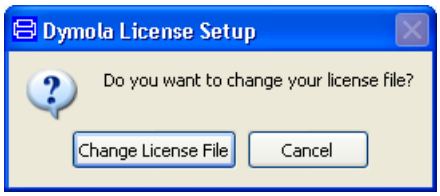
Start Dymola and select **Tools > License Setup**, select the **Setup** tab. Click on the **Browse** button and open the license file you saved. The path of the license file is shown in the dialog.

Specifying the license file.

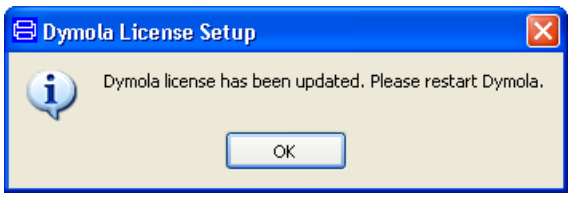


You have the option of installing the license file only for the currently logged in user, or for all users on this computer. The latter requires administrator rights.

Click on the **OK** button. Dymola will ask for confirmation before overwriting your old license information.



After changing the license server setup, you must restart Dymola to use the new server. You may delete the saved license file, Dymola has created a copy.



Specifying the license key by the environment variable DYMOLA_RUNTIME_LICENSE

In some specific cases, you need to point to the license key explicitly. To specify the license key (when not using the default license file), it is possible to set the environment variable

DYMOLA_RUNTIME_LICENSE. This variable can either point to a license file or specify a license server.

An example pointing to a license file could be:

```
set DYMOLA_RUNTIME_LICENSE=C:\My Documents\dymola.lic
```

An example pointing to a license server could be:

```
set DYMOLA_RUNTIME_LICENSE=27000@my.server.com
```

The main use of this feature is for the Dymola runtime concept below.

Dymola runtime concept

Models developed by users that lack export options can still be run at other computers using a runtime concept. Dymola runtime concept requires the user of the model to have a Dymola license. The license file should be defined by the environment variable `DYMOLA_RUNTIME_LICENSE`, see above.

Note that the runtime concept is not supported for the GCC compiler; see limitations in section "GCC compiler" starting on page 1548.

For information about license requirements in general, see section "License requirements" starting on page 1606.

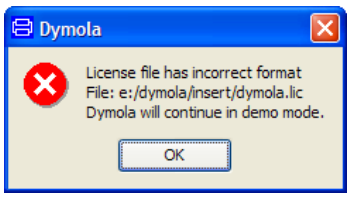
For more specific information about export options in particular, see the chapter "Simulation Environments", section "Code and Model Export".

Upgrading from Dymola 6.1 and earlier

The license file format of Dymola has been upgraded to include the latest security technology. For that reason, license files for earlier versions of Dymola are not compatible with Dymola 7.0 and later, and license files for Dymola 7.0 and later are not compatible with older versions of Dymola.

If Dymola 7.0 and later finds an old license file at start-up, a diagnostic message about incorrect license file format is displayed. Dymola will then continue execution in trial mode.

Dymola has started with an old license file.



17.1.5 Additional setup

Language

The selection of translation file for Dymola is by default based on the regional language setting.

Please note that currently only English and Japanese are available in Dymola.

If the regional language setting should not be used, there are two ways of overriding it.

The first is to use a command line setting of the language: `-translation <language>`. Two examples are important:

- A customer that wants to run Dymola in Japanese on a machine with regional language setting other than Japanese. This can be done by starting Dymola with the command **"C:\Program Files\Dymola 2023\bin64\Dymola.exe" –translation ja** (given using a 64-bit Dymola from the default location).
- A customer that wants to run Dymola in English on a machine with regional language setting Japanese. This can be done by starting Dymola with the command **"C:\Program Files\Dymola 2023\bin64\Dymola.exe" –translation none** (given using a 64-bit Dymola from the default location).

The second way to override the default selection of translation file is to specify what translation file to use, this can be done with the command line option `-translationfile "<filename.qm>"` when starting Dymola. One specific opportunity here is to use a translation file other that the one in the Dymola distribution. The file can be located anywhere on the machine, since the command line option demands the path of the file to be specified. An example could be to start Dymola with the command **"C:\Program Files\Dymola 2023\bin64\Dymola.exe" –translationfile "E:\Extra\NewJapaneseTranslationFile.qm"** (given using a 64-bit Dymola from the default location, and a translation file NewJapaneseTranslationFile located in E:\Extra).

Note that command line options can be included in shortcuts to Dymola, see section "Creating shortcuts to Dymola" below.

Location of startup directory

The startup directory is the directory that is defined as the current working directory when starting Dymola. The current working directory is used as default location for opening files and for saving simulation results.

Having started Dymola, you can change both the startup directory and the current working directory (they do not necessarily have to be the same). For more information, refer to the index entry "*directory : startup directory*" and "*directory : current working directory*" in the index in the end of the manual.

When installing Dymola, an environment variable DYMOLAWORK is automatically defined. It is set internally to the startup directory. If this directory is not set, the variable is set to `…Documents\Dymola`. That subdirectory will be created if it doesn't exist.

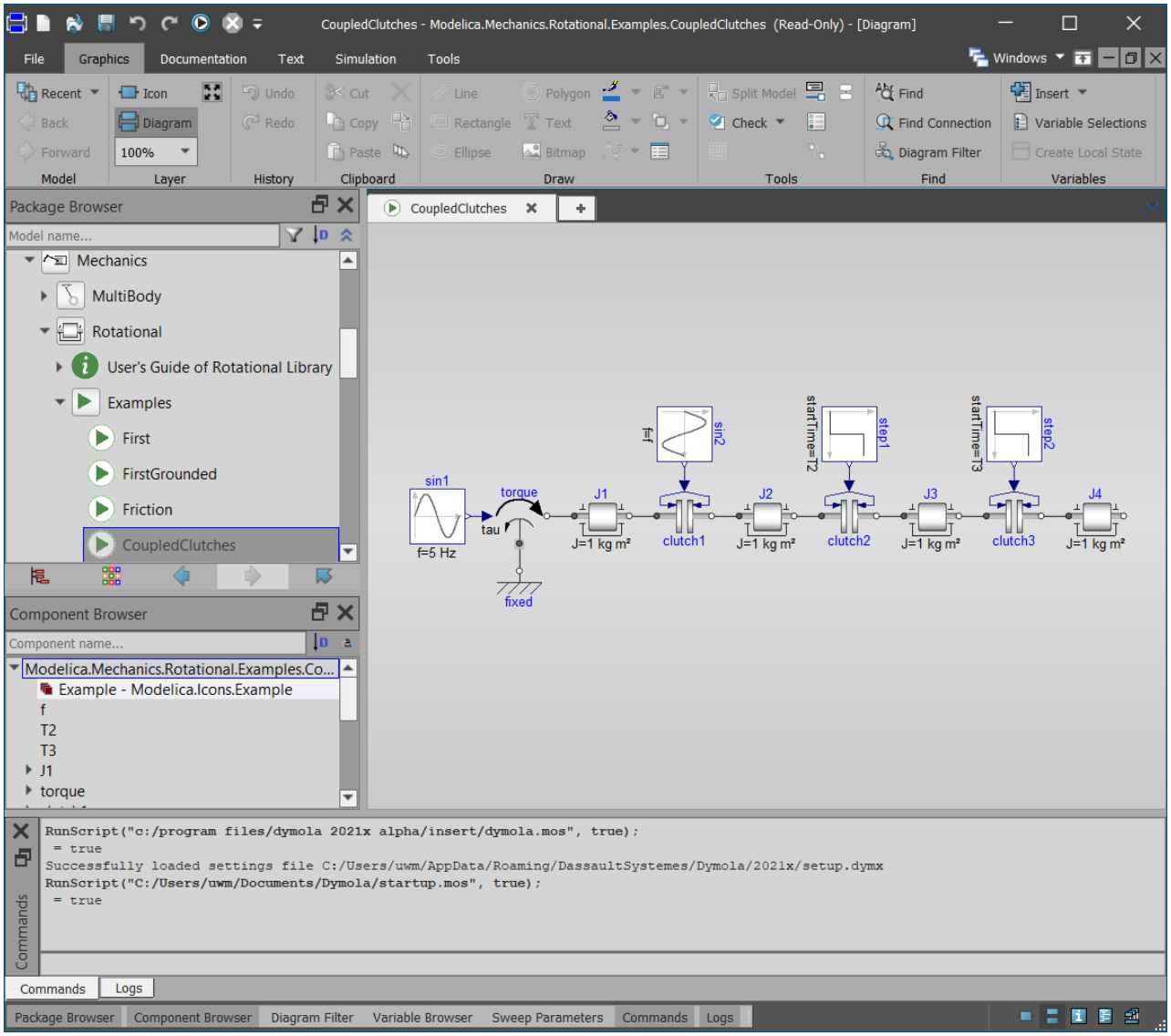
Note. The environment variable DYMOLAWORK can only be changed internally from Dymola and cannot be used to change the startup directory.

Using high resolution (4K) screens

In general, it is recommended to use Windows 10 when using 4K screens. To work with DPI scaling in Dymola, start Dymola.exe with `-highdpi` from the command prompt of Windows.

Support for Dymola in “dark mode”

Dymola can be started with the command line switch `-dark` or `/dark`. This will start Dymola with an alternative darker color theme:



To enable the traditional white (high contrast) background in the graphical editor also in “dark” mode, you can set the flag:

```
Advanced.UI.HighContrastDiagram = true
```

(The flag is by default `false`.) This can help if a model is hard to see against the grey background.

Creating shortcuts to Dymola

Shortcuts to start Dymola with startup script.

Sometimes it is convenient to create shortcuts to the Dymola program, typically to make Dymola use a startup script to, for example, open specific packages and set flags.

A shortcut is created as follows:

1. Click the right mouse button on the desktop.
2. Select **New > Shortcut** from the popup menu.
3. Browse for the Dymola program `Program Files\Dymola 2023\bin64\dymola.exe`.
4. Enter a suitable name and finish the creation of the shortcut.
5. Right-click on the newly created shortcut.
6. Select **Properties** from the popup menu.
7. Select the **Shortcut** tab of the dialog window.
8. If wanted, add command line options in the **Target** field. In our example, to add a startup script `Dymola_startup.mos` that is located in `E:\MyExperiments\MySettings\` as command line option, the full line after adding this option will be `"E\Program Files\Dymola 2023\bin64\dymola.exe" "E\MyExperiments\MySettings\Dymola_startup.mos"` (Note that there must be a space between the two paths.)
9. Note that from Dymola 2017 FD01, the working directory can be handled automatically in the settings file; it is not needed to change anything in the **Start in** field. See the index entry "*directory:startup directory*" in the manual for finding more information.
10. Click **OK** to create the shortcut.

Remote installation of Dymola

Dymola (whether downloaded as a zip file or on DVD) consists of a number of files. Remote installation of Dymola is possible. For example, the following command makes a quiet installation of Dymola and all libraries with Modelica version 3:

```
setup.exe /s /v"INSTALLLEVEL=201 /qn"
```

The value of the `INSTALLLEVEL` property controls which components are installed, presently only 201 and unspecified is used. Unspecified means omitting the commercial libraries, and the command is:

```
setup.exe /s /v" /qn"
```

Note that you need to run any of these commands as administrator, one way is to start the command shell as administrator.

Working with a Modelica version that is newer than the one in the distribution.

To work with a Modelica Standard Library (MSL) version that is newer than the one in the Dymola distribution, you must:

- Download and install the new MSL version. You can use on-demand installation; see "On-demand installation of downloaded libraries" on page 1565.
- Set the global flag `Advanced.PlaceDymolaSourceFirst=2`. For more information of the flag, see below.
- If you want to use the new version as default version, do the following:
    - Apply the command **Edit > Options…**, select the **Versions** tab, select the MSL version you want, and tick **Force upgrade of models to this version**.
    - Click **OK** to save the new default version.

Note! The libraries in the Dymola distribution may not support the new MSL version.

The reason why you must set this flag is that there are links in Modelica Standard libraries to internal libraries that are used when compiling models. If dymola/source and dymola/bin/lib are loaded before MSL, they will be used instead of the corresponding newer internal libraries, and this will not work. The `Advanced.PlaceDymolaSourceFirst` controls in what order files are loaded. The possible values are:

- 0 place dymola/source first for compilation
- 1 (default) presently this value works the same as the value 0
- 2….always place dymola/source last

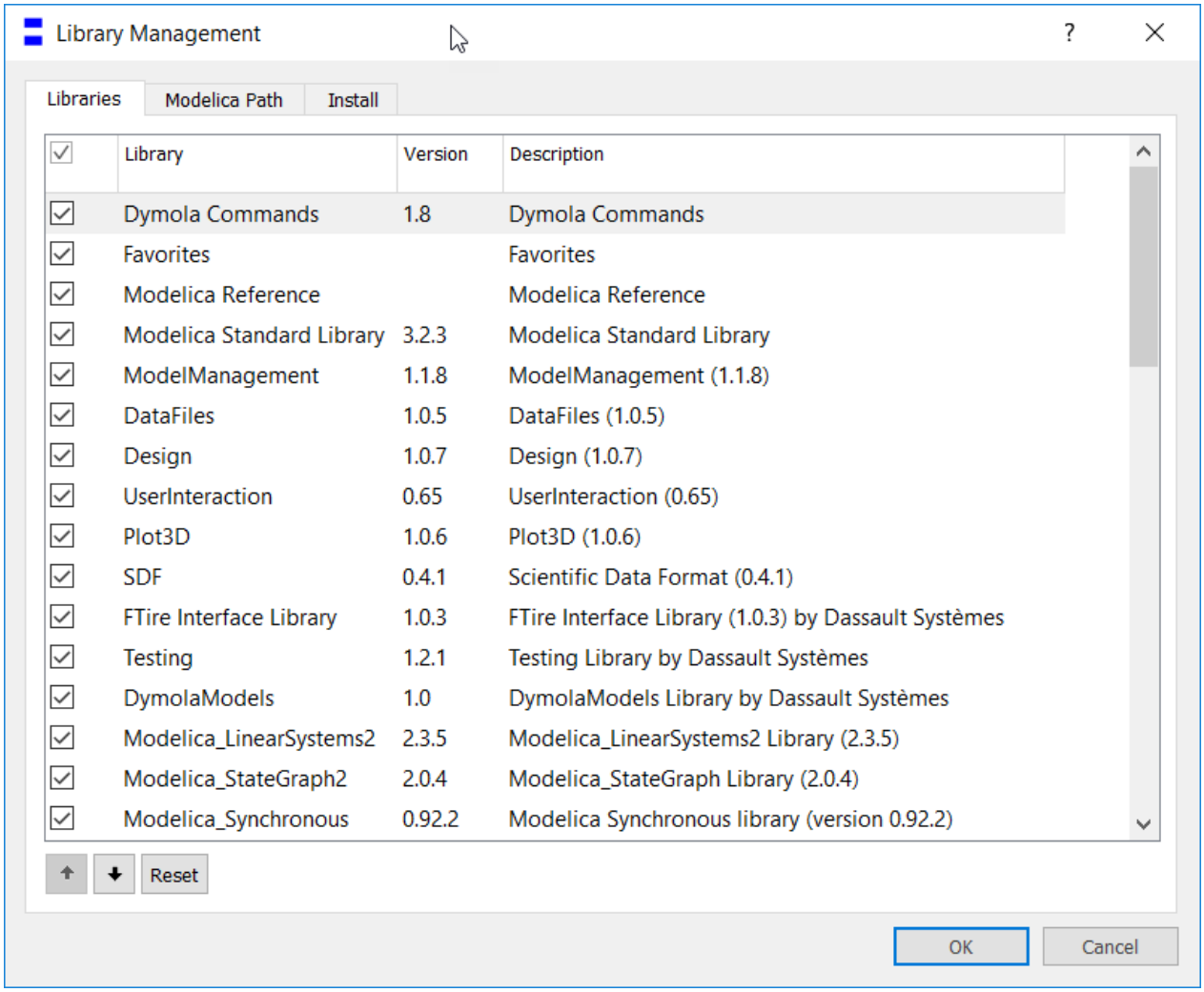
Installing earlier Modelica versions including compatible libraries

If you are working with MSL 4.0.0 and need to work with an older Modelica model that has not been converted to the new MSL version, you can yourself install MSL 3.2.3 libraries separately. They are included in the Dymola media as `extra\CompatibilityLibraries MSL 3.2.3.zip`. To use them, unpack them in, typically, `…Program Files\Dymola 2023\Modelica\Library`. **Note**. Do not forget to apply the command **Edit > Options…**, select the **Versions** tab, select the MSL version you want, and tick **Force upgrade of models to this version** to select the proper version.

If you want to install older Modelica versions than that in the distribution, please contact support at https://www.3ds.com/support. (**Note** that Dymola 2018 FD01 and newer do not support Modelica 2. Conversion from Modelica 2 to Modelica 3 is still possible in Dymola 2018, but it is easier to verify the conversion in a previous version of Dymola, e.g. Dymola 2015, where Modelica 2 is still fully supported. See MigrationModelica2.pdf for more information.)

Customizing the File > Libraries menu

You can customize the **File > Libraries** menu by the command **Tools > Library Management**, the **Libraries** tab:



Notes!

- The content of this tab depends on the MODELICAPATH environment variable, which can be changed by for example the **Modelica Path** tab – if you make changes in the **Modelica Path** tab, for example, adding directories, and click **OK** in that tab, the content of the **Libraries** tab may change. See below for more information about MODELICAPATH.

- When you want to install downloaded libraries, you can also select if they should appear in the **File > Libraries** menu when installing, if you use on-demand installation. See “On-demand installation of downloaded libraries”on page 1565.

In the Libraries tab you can:

- Include or exclude all libraries by using the checkbox in the heading
- Include or exclude any library in the list by the checkbox in front of it
- Move the selected library upwards or downwards using the arrow buttons

Any action is immediately implemented when clicking **OK**.

The setup is saved between sessions.

To restore the default library configuration, you can click **Reset** in the menu above.

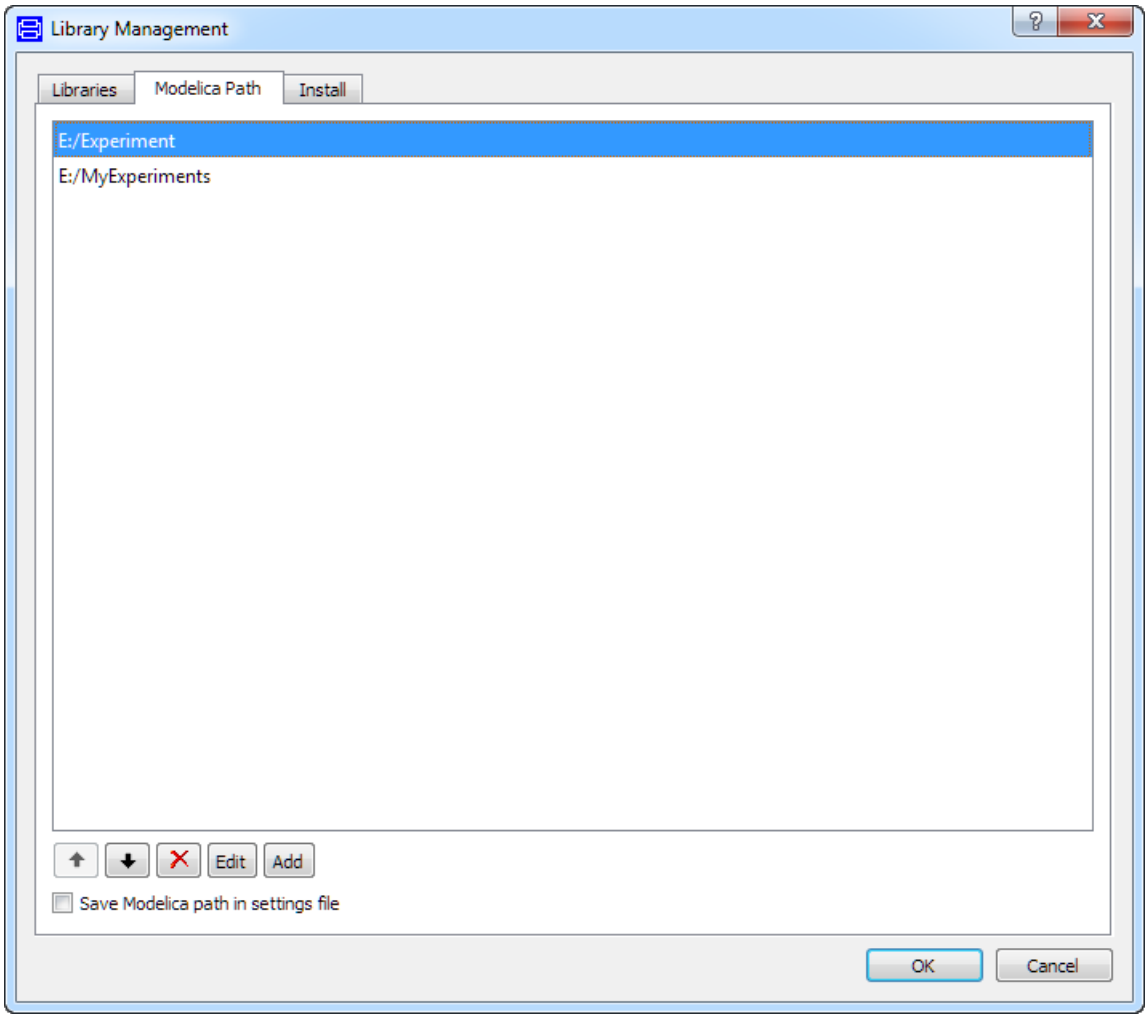
Managing library locations

The environment variable MODELICAPATH and how to manage it

Dymola will find libraries by searching all directories in the environment variable MODELICAPATH. If not set by the user, MODELICAPATH contains only dymola/Modelica/Library.

Note the option to use enclosing package paths in addition to MODELICAPATH in the search, see next section.

It is possible to manage the MODELICAPATH from within Dymola. The most convenient way is to use the command **Tools > Library Management**, the **Modelica Path** tab. This opens a dialog that makes it possible to change, add, or delete directories in the environment variable. An example of the dialog is:



Note that the default path dymola\Modelica\Library is not displayed.

You can use the buttons in the dialog to sort, delete, edit, and add paths.

Keys can be used for the actions as well:

- Move up: **Ctrl+Up**
- Move down: **Ctrl+Down**
- Edit: **Space**
- Add: **Plus**

If you only want to add or delete a directory, you can also use the built in function

```
AddModelicaPath(path, erase=false);
```

The function appends a directory, specified by the string variable path, to the Modelica path (if erase=false (default). If `erase=true` the specified directory is instead erased from MODELICAPATH).

An alternative is to use

```
Modelica.Utilities.System.setEnvironmentVariable("MODELICAPATH"
, "…");
```

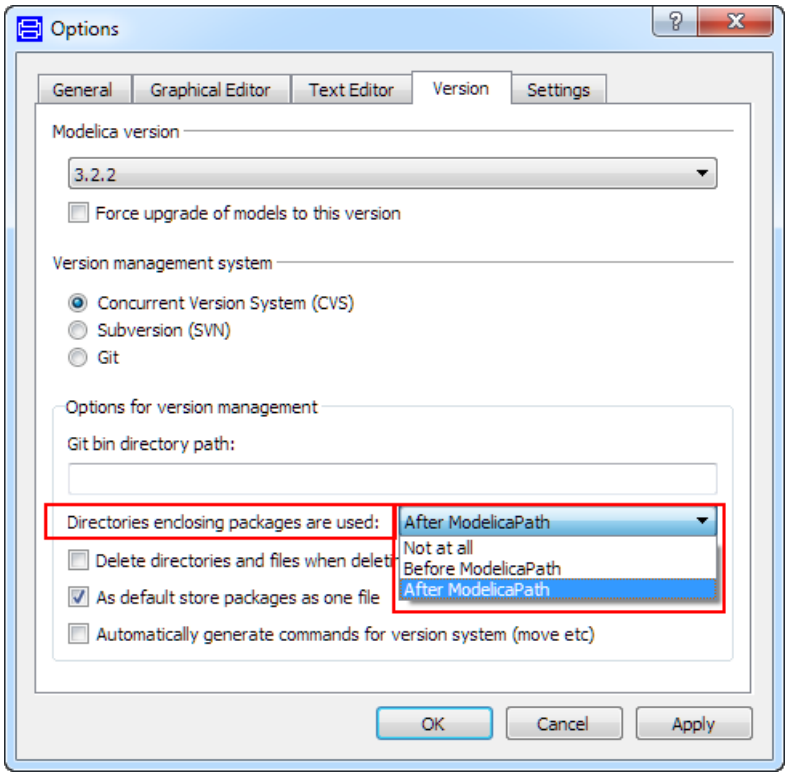
Notes:

- The menus **File > Libraries** and **File > Demos** are rebuilt after a change of MODELICAPATH.
- To keep the change of MODELICAPATH between sessions, it must be saved. This can be done by ticking the setting **Save Modelica path in settings file** in the menu above. This checkbox corresponds to the checkbox **Modelica path** in **Tools > Options**, the **Settings** tab.

Option to also use enclosing package paths in addition to MODELICAPATH

To make it easier to, for example, automatically find multiple libraries that are stored in the same directory, there is an option to also use directories enclosing packages in addition to MODELICAPATH.

This option is available as **Directories enclosing packages are used** in the **Version** tab reached by the command **Tools > Options**:



- **After ModelicaPath** (default) means that the enclosed directories are searched after having searched the paths in MODELICAPATH.
- **Before ModelicaPath** means that the enclosed directories are searched before searching the paths in MODELICAPATH.
- **Not at all** means that only the paths in MODELICAPATH.

The option is also available as a flag `Advanced.EnclosingDirectoryModelicaPath`. The value of the flag can be 0 ("Not at all"), 1 ("Before"), or 2 ("After"). The default value is 2.

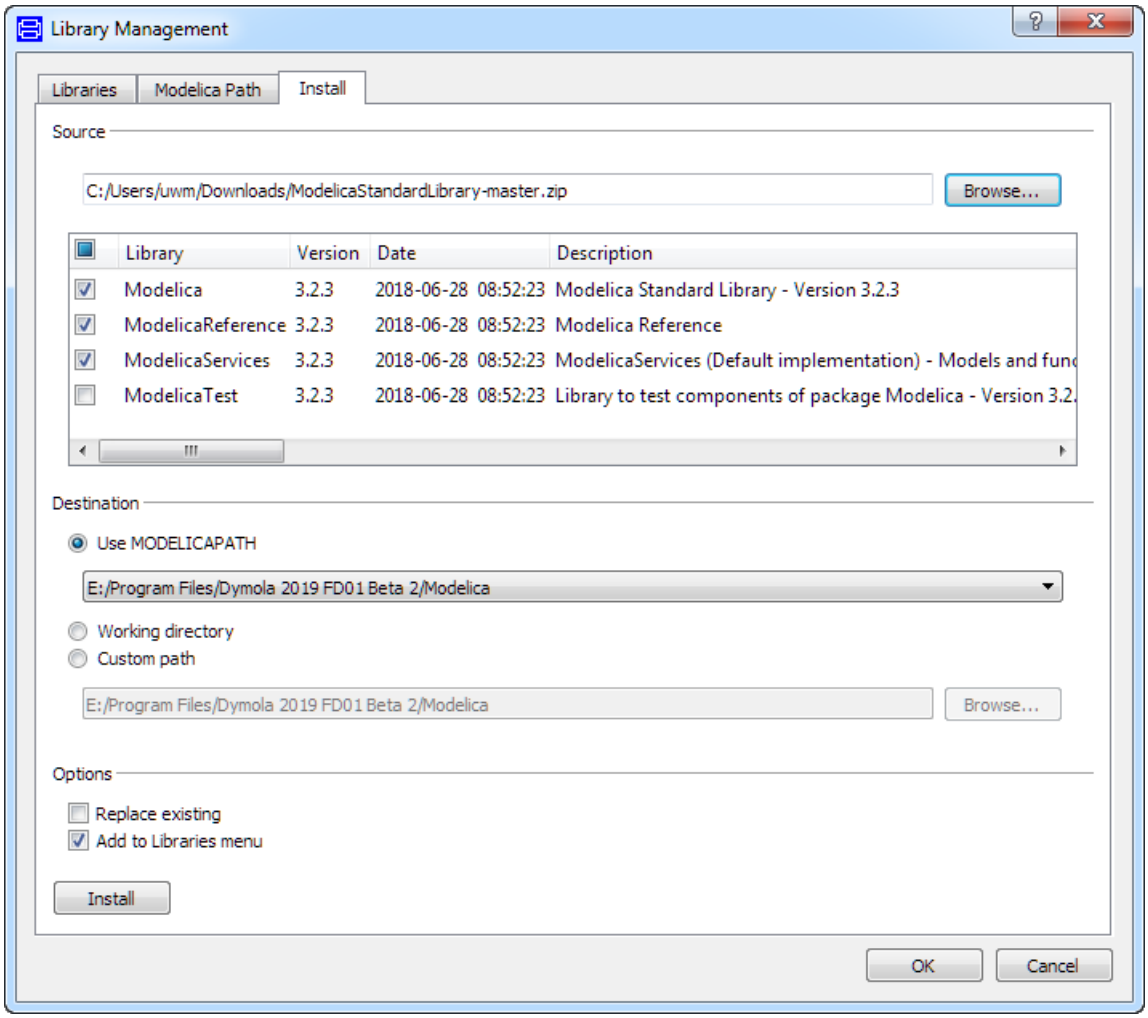
Note that the content of MODELICAPATH is not changed; the option specifies additional locations to use.

On-demand installation of downloaded libraries

To use on-demand installation of downloaded libraries, you can use the command **Tools > Library Management**, the **Install** tab. Here you can as source select a mo/moe file or a zip file and install from this source. When loaded, the file content is presented for review before installation. As zip files may contain several packages, checkboxes are provided to enable partial installations.

Only reasonable combinations of packages within zip files are considered. In particular, once a package.mo or package.moe is found on a certain folder level, deeper searches for other packages are truncated.

As an example, consider using the file `ModelicaStandardLibrary-master.zip` as a source:



Here we have chosen to omit the `ModelicaTest` library and to use a folder from MODELICAPATH [21] as destination. Note that some destinations may require running Dymola as Administrator (Windows) or root (Linux).

[21] For more information about MODELICAPATH, see previous section.

You may also choose whether to replace any existing library or whether to add the library in the **File > Libraries** menu or not. If **Add to Libraries menu** is checked, the file `libraryinfo.mos` [22] is copied [23] or created, else it is omitted. In the former case, the MODELICAPATH environment variable is updated if necessary.

More about libraries and the building of menus of libraries and demos

General information

Dymola can automatically recognize different libraries in order to e.g. build the **File > Libraries** and **File > Demos** menus. It is easy to add new libraries and to add new versions of existing libraries.

All information about a library exists in a local file, so it is possible to just "unzip" a subdirectory containing a package, and it will automatically be recognized by Dymola.

No update of a common file is needed, hence no need for special installation scripts. It also makes it easy to delete libraries, just delete the directory.

Using library information

Associated with each package is a Modelica script that is automatically located by Dymola at program start. This script can contain a set of commands that describes the package and builds e.g. **File > Libraries**.

The script is called `libraryinfo.mos` and can be stored in three alternative locations. Assuming the package XYZ is stored as `dymola/Modelica/Library/XYZ`, the script can be stored as any of:

- `dymola/Modelica/Library/XYZ/libraryinfo.mos`
- `dymola/Modelica/Library/XYZ/Scripts/libraryinfo.mos`
- `dymola/Modelica/Library/XYZ/Resources/Scripts/Dymola/libraryinfo.mos`

(The above is also the order of searching for this file.)

Note that you must yourself create the scripts for your own libraries if you e.g. want to add them in menus. It is wise to look both below, and at the already present `libraryinfo.mos` files for libraries already in the **File > Libraries** menu when doing this.

Building menus

There is currently a low-level script command to build libraries and demos menus, e.g.:

For a simple example of a library (please also compare the section "Customizing the File > Libraries menu" above):

---

[22] For more information about `libraryinfo.mos`, see next section.
[23] Any existing `libraryinfo.mos` is copied without validation. If it is incorrect, the library may not show up in the **File > Libraries** menu.

```
LibraryInfoMenuCommand(
  category="libraries",
  text="Cooling",
  reference="Cooling",
  version="1.2",
  isModel=true,
  description="Cooling Library (1.2) by Dassault Systèmes",
  ModelicaVersion=">=3.2.2"
  pos=525);
```

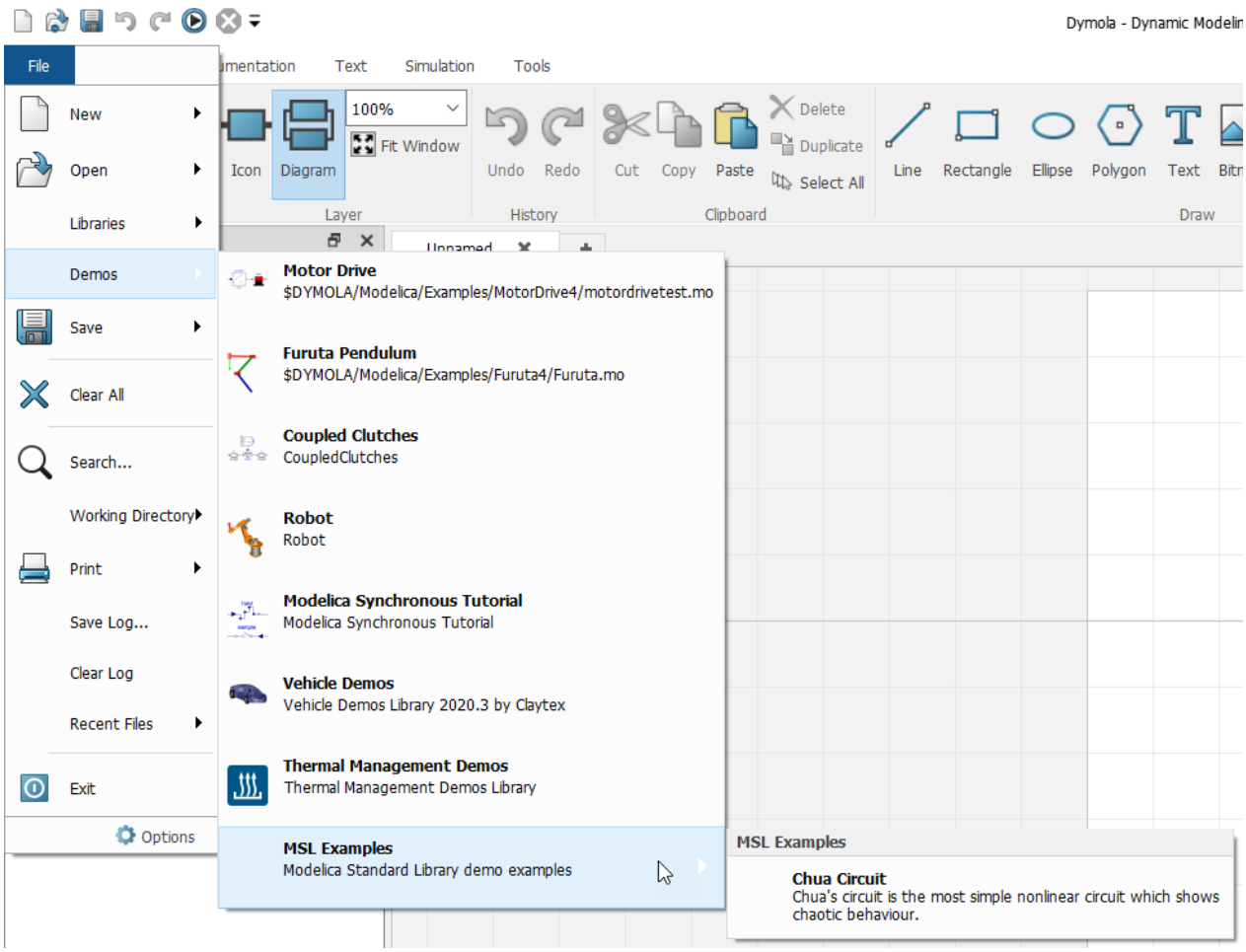
For a simple example of a demo:

```
LibraryInfoMenuCommand(category="demos",
  text="My Demo",
  reference="MyDemo.MyDemoModel",
  isModel=true,
  description="My demo, basic",
  pos=9999);
```

A demo with sub-category of a menu could be:

```
LibraryInfoMenuCommand(category="demos",
  subCategory={"MSL Examples", "Modelica Standard Library demo
examples"},
  text="Chua Circuit",
  reference="Modelica.Electrical.Analog.Examples.ChuaCircuit",
  isModel=true,
  description="Chua's circuit is the most simple nonlinear circuit which
     shows chaotic behaviour.",
  pos=9999);
```

The result of the command **File > Demos** when using this function in libraryinfo.mos is:



The attributes have the following meaning (the list contains more attributes than the examples above):

Attribute	Meaning
category	Primary menu category (“libraries”, “demos”, or “persistent”)
subCategory[:]	Optional sub-category of the menu entry, a vector of zero to two strings. The first string is a sub-menu name; the second string is a longer description text. For example, `subCategory = {"MSL Examples", "Examples from the Modelica Standard Library"}`
text	Text shown in menu
reference	Model path or command string
isModel	If true, the text is a model path, otherwise a command.
description	Longer description, for example shown in status bar

version	Version of library (does not apply to demos). See **Important** below.
ModelicaVersion	Required version of Modelica Standard Library, e.g. "> 2". The value "2" means ">=2".
pos	Position in the menu. The menu alternatives are sorted according to this attribute, lowest numbered at the top.
requiredFlags[:]	If flags are specified, they have to be true for the menu to be active. If a flag is preceded by ! it means that the flag must be false for the menu to be active.
iconName	For specifying an external icon, if needed. Please see next section.

To handle different libraries and groups of libraries, and to make sure Dymola has a consistent ordering of Libraries, Dassault Systèmes allocates ranges of positions to different library vendors. For example, 0 to 999 could be reserved for Dassault Systèmes, 1000 to 1999 for DLR, etc.

Important: For libraries, the version must be specified in the libraryinfo.mos file, and in the corresponding library file (package.mo). The latter is done by the version annotation. The version specified must be the same in both files.

Support for external icons for the menus File > Libraries and File > Demos

To support external icons in the **File > Libraries** and **File > Demos** menus, there is an input parameter `iconName` of type String available in `LibraryInfoMenuCommand` (see table above). The default value is `"libraryicon.png"` which means that the easiest solution to add an external icon is to create an icon with that name in the same folder as `libraryinfo.mos`. Notes:

- Other file formats are also supported, for example, svg.
- A relative path to the icon can also be added if the icon is located in another folder than `libraryinfo.mos`.
- Oversized icons are scaled to fit in the menus.

Adding a menu separator

It is possible to add a separator (horizontal line) in the menus. For example,

```
LibraryInfoMenuSeparator(
  category="libraries",
  pos=101);
```

The arguments have the meaning described in the table above.

Option to use a custom setup file

It is possible to run Dymola with a custom setup file. This enables having different setups for different projects; for example:

- Preloading project-specific libraries

- Maintaining conflicting and alternative library versions by having different Modelica paths
- Setup tailored for simulation, customer-specific presentations, or with GUI layout optimized for projectors.
- Using different compiler settings for different customers

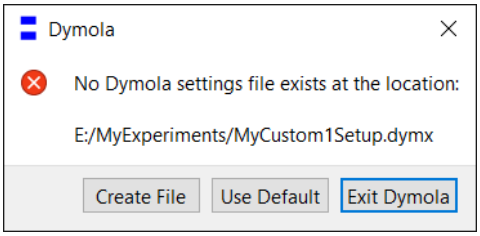
(For more information about the content and handling of the setup file, see the index entry "*setup file*".)

You can in fact copy the whole installation to another location, and use a custom setup file when starting the copy, using the command line as below. This copy of Dymola will be totally independent from the original, except for the licensing.

To specfy the path to the custom setup.dymx file, the command line argument `-setup` is used, for example:

```
Dymola.exe -setup "E:\MyExperiments\MyCustom1Setup.dymx"
```

If this cannot be found, an error message appears:



You have the following alternatives:

- **Create File** tries to create an empty `setup.dymx` at the specified location.
- **Use Default** uses the default setup file, on Windows typically:

  ```
  C:\Users\<user>\AppData\Roaming\DassaultSystemes\Dymola\
  2023\setup.dymx
  ```

- **Exit Dymola** exits Dymola. If Dymola is started with any of the command line arguments `-nowindow` or `-externalinterface` then Dymola is exited with an error code.

Loading a package with user-defined menus and toolbars that should not be deleted by File > Clear All

A package with user-defined menus and toolbars where the menus and toolbars should not be deleted by the command **File > Clear All** can be automatically loaded by a libraryinfo.mos file with `category="persistent"` (see also above section).

```
LibraryInfoMenuCommand(
  category="persistent",
  reference="<Class to preload>",
  text="dummy")
```

For more information about creating such menus, see the chapter “User-defined GUI”, section “Extendable user interface – menus, toolbars and favorites”.

License expiration settings

The default behavior of Dymola is:

- To start to warn the user that a license is to expire, 30 days before expiration.
- To continue in trial mode if a license has expired or is faulty.

The behavior can be configured with a command line argument

Consider a user wanting to have the first warning 5 days before the license is expiring, and wanting to terminate Dymola if the license is not found or invalid. Assuming a 64-bit Dymola with default location on a MS Widows computer, Dymola could be started with the following command line using the Command Prompt in Windows:

```
"C:\Program Files\Dymola 2023\bin64\Dymola.exe" /days -5
```

The value (5) controls how many days that should be left to expiration when warning, and the minus before the value is added if Dymola should terminate if the license is not found or invalid.

“-” can be used instead of “/“; the example above will then be:

```
"C:\Program Files\Dymola 2023\bin64\Dymola.exe" -days -5
```

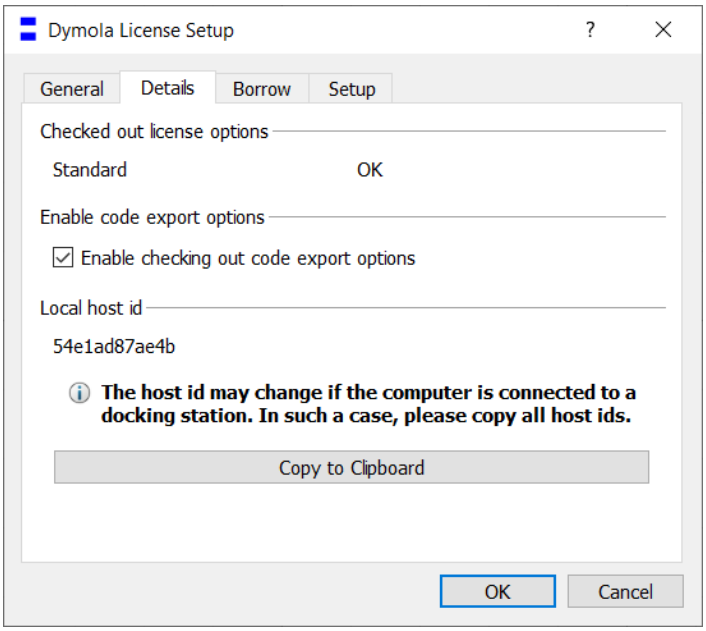
1 day is the minimum start time for warning of license expiration; the warning cannot be completely disabled.

Preventing checking out license options from a license server

It is possible to prevent Dymola from checking out certain license options from the license server, if a sharable license is used. (It is also possible using a node-locked license, e.g. if a user wants to test if a certain model still works without a certain library.)

Using the command **Tools > License Setup** and then looking in the **Details** tab reveals license options currently checked out.

Example of license options checked out.



If the user wants to prevent some option from being checked out, it can be done in a number of ways:

- For code export options, you can untick **Enable checking out code export options** in the menu above. Doing this, you will prevent both BinaryModelExport and SourceCodeGeneration to be checked out. The setting will be remembered between Dymola sessions. The setting can also be accoplished using the flag `Advanced.EnableCodeExport=false;`

  For more information concerning code export, please see the chapter “Simulation Environments”.

- By modifying the shortcut to Dymola.
- By starting Dymola with a certatin command line option using the Command Prompt in Windows.

Modifying the shortcut will result in prevention of check out of specified options each time Dymola is started using that shortcut, as. Starting Dymola using a modified command from the command prompt in Windows will only result in prevention of check out of specified options in that session.

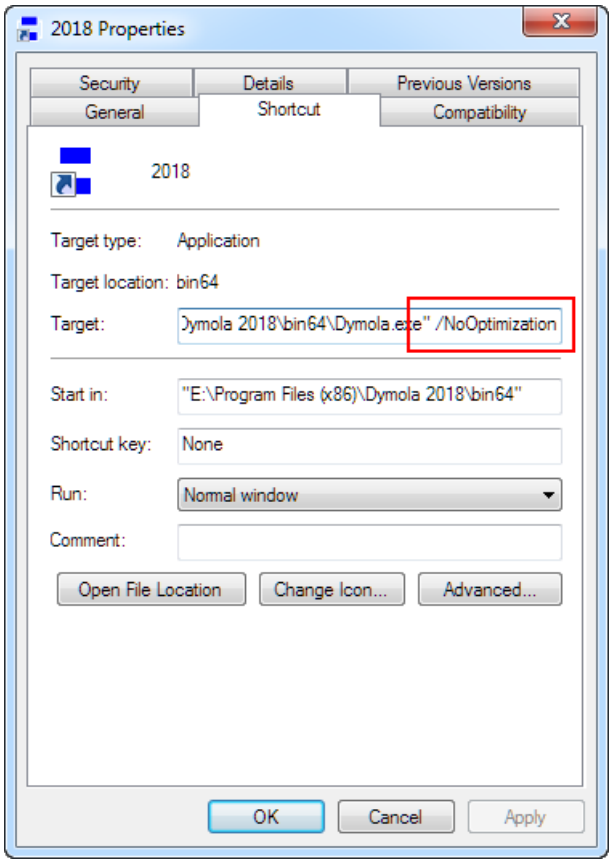
Since the command for prevention of checking out license options is generic, it is very important to use the correct name of the option, including correct use of captitals. The best way is to look at the checked out options using the command above, and mark and copy the name (for example, `Optimization`) of the option that should not be checked out, to insert that name when using any command.

Modification of the shortcut to Dymola

If a new shortcut is needed, please look at the section “Creating shortcuts to Dymola” above.

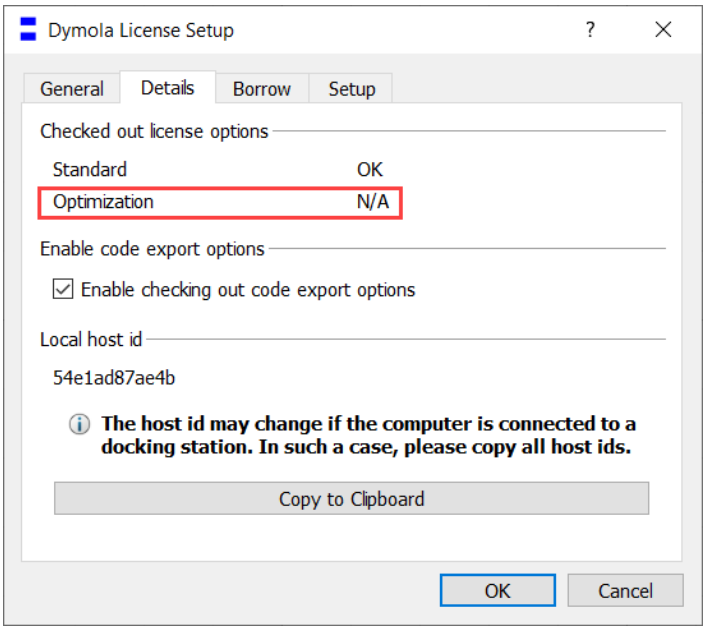
To modify the shortcut to prevent checking out a certain option, right-click the shortcut and modify the **Start in**: by adding `<space>/No<optionname>` in the end of the command. If the option Optimization should not be checked out, the shortcut should be modified like in the figure below. (We prevent check-out of the `Optimization` option.)

Modified shortcut.



Closing Dymola and starting it again, the following information will be found in the license tab:

Prevention of checking out a license option.



Now the `Optimization` license option will not be possible to check out. As long as the shortcut is not modified, Optimzation will not be possible to check out from Dymola started by that shortcut.

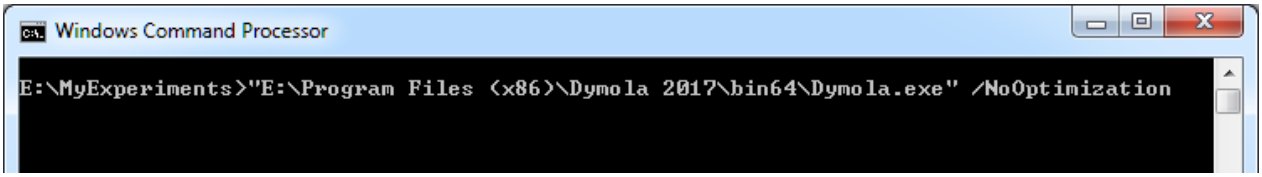
To enable check out of Optimization, Dymola must be closed and then restarted using a shortcut without the command line option for Optimizaton.

More than one option can be prevented from check out – just add more strings like the one used. Do not forget the space.

Starting Dymola using a modified command in Command Prompt of Windows

A Windows command prompt can be activated using **Help > All Programs > Accessories > Command Prompt** in Windows.

To start one session in Dymola where the license option Optimization cannot be checked out like in the example above, the command in the command prompt will look like:



Cross-compilation for Linux on Windows

Introduction

Dymola on Windows supports cross-compilation for Linux via use of Windows Subsystem for Linux (WSL). The default WSL setup is 64-bit only and Dymola adopts this limitation.

Install WSL

Installing WSL will give you a command-line Linux environment on your Windows computer. This environment can then compile code in a native Linux environment, such as case Ubuntu 20.04 LTS (Long Term Support) as used in the example below. The example below shows installation and setup on Windows 10 version 20H2.

The WSL Linux environment can compile the generated model C code from Dymola in order to produce a Linux executable dymosim or a Linux FMU.

To install WSL and download a suitable Linux distribution, in our case Ubuntu 20.04 LTS:

- Open a Command Window in Administrator mode.
- Install WSL and Ubuntu by giving the command:

  ```
  wsl --install -d Ubuntu-20.04
  ```

  You may be asked to reboot your computer to complete the installation.
- Create you user account when prompted.
- Update your system and install the C compiler and other tools:

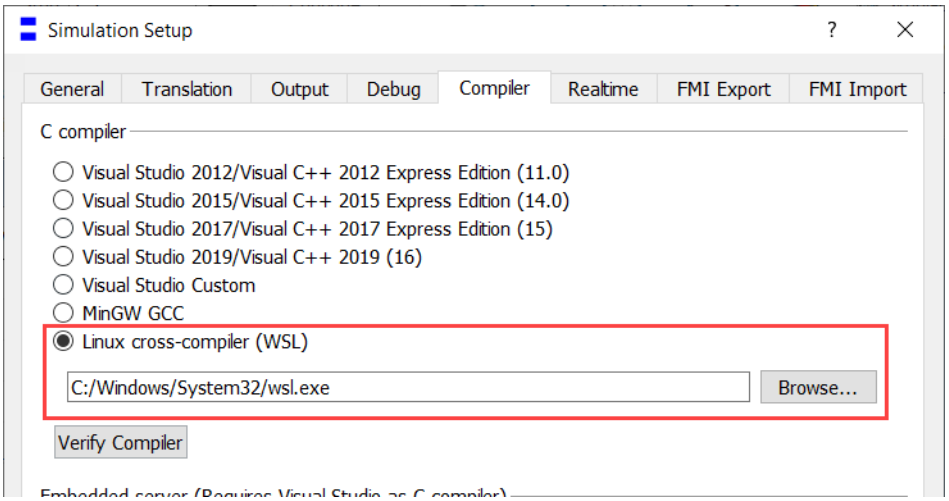
  ```
  sudo apt update
  sudo apt upgrade
  sudo apt install gcc g++ zip dos2unix
  ```
- Ensure that WSL can change file permissions. That can be done in WSL by ensuring that the file `/etc/wsl.conf` has the following two lines (and creating that file if it does not exist):

  ```
  [automount]
  options = "metadata"
  ```

  Important! Restart WSL or reboot the computer after any changes of this file.

For a general description, about installing WSL, valid from Windows 10 version 2004, and for Windows 11, see https://docs.microsoft.com/en-us/windows/wsl/install. Here you can also find a link on how to install WSL on older Windows 10 versions.

To activate the WSL installation, select **Linux cross-compiler (WSL)** in the simulation setup, reached by the command **Simulation > Setup**, the **Compiler** tab:



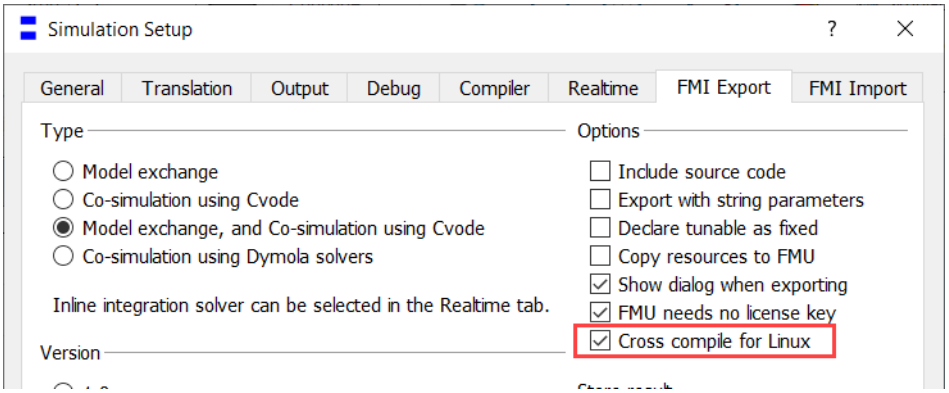
To test the WSL installation, click **Verify Compiler**.

If you merely wish to use the WSL as the main compiler, your setup is done here. For cross-compilation for FMU exports, see next section.

The settings are saved between sessions.

FMU export for multiple platforms

Dymola partially supports cross-compilation when exporting FMUs on Windows. It is partial in the sense that your main compiler environment is limited to Visual Studio or MinGW and the cross-compiler environment is limited to WSL. So first, you need to select either Visual Studio or MinGW in the simulation setup (see the above figure). Then, to activate cross-compilation, activate the setting **Cross compile for Linux** in the simulation setup, reached by the command **Simulation > Setup**, the **FMI Export** tab:



Now, when exporting an FMU, you will get 64-bit Linux binaries in addition to Windows binaries.

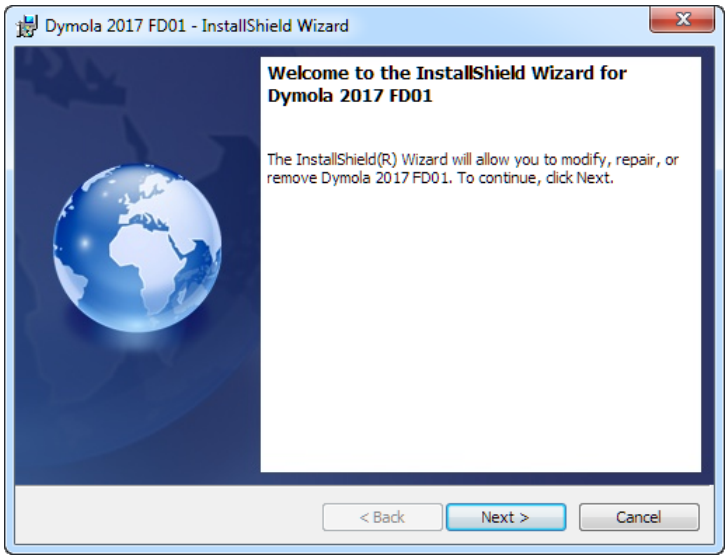
Important! In this case, you should *not* select **Linux cross-compiler (WSL)**; the setting **Cross compile for Linux** sets the suitable cross-compilation.

To activate the setting **Cross compile for Linux** corresponds to setting the flag `Advanced.FMI.CrossExport=true`. (The default value of the flag is `false`.)

Note that the value of the flag/setting is *not* saved between sessions.

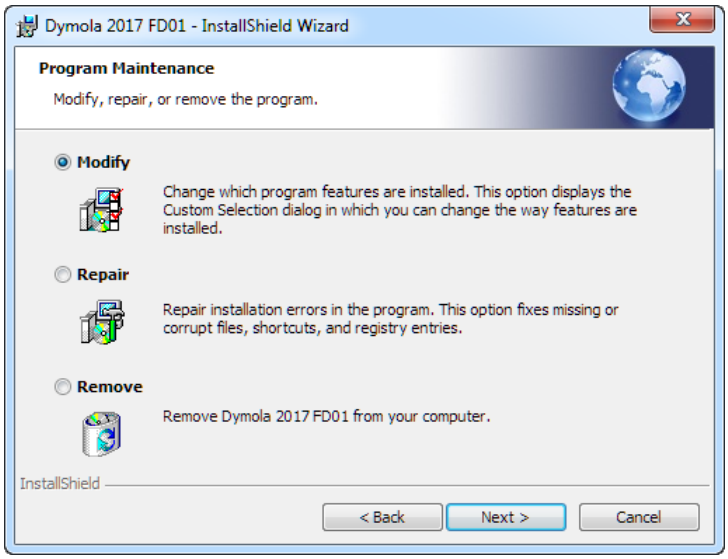
17.1.6 Changing the setup of Dymola

Under Windows, you can change the setup of Dymola, for example to install additional libraries. Click on the **Start** button in the Taskbar, select **Control Panel** and open **Programs and Features**. Select the relevant version of Dymola and click on the **Change** button.



Selecting **Next>** will display

Changing Dymola setup.



To change the setup, choose **Modify**. The rest of the procedure will be the same as when installing Dymola from scratch. Please see previous sections. To restore files in the Dymola distribution that have been deleted by mistake, choose **Repair**. **Remove** will remove the installation.

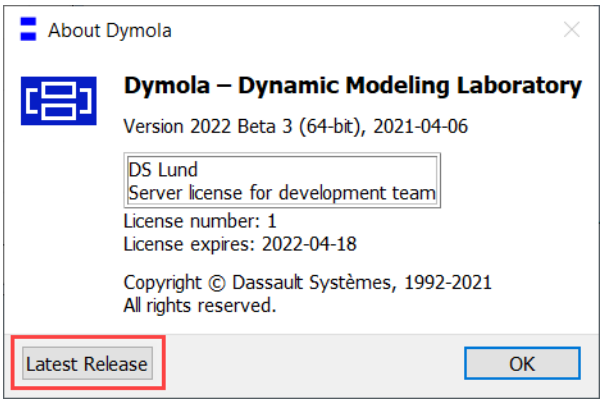
17.1.7 Removing Dymola

Please see previous section. Do not delete or rename the Dymola directory. Microsoft Windows Installer keeps track of all installed directories and will try to repair if altered. The installation will by default use a directory name that reflects the version of Dymola, but this can of course be changed during setup.

Note that files that you have created in the Dymola distribution directory, for example by running the demo examples, are not deleted when removing Dymola. The remaining files and directories (if any) may be deleted through the Explorer.

17.1.8 Getting information about the latest Dymola release

To get information about the latest Dymola release available, use the command **Tools > About Dymola**, and then click **Latest Release**.



17.1.9 Installing updates

Updated versions of Dymola are either distributed on CD, or can be downloaded from a location provided by your sales channel.

Multiple versions of Dymola can be installed, but you cannot install into an existing Dymola directory. Configuration settings and the license file are shared by all installed versions of Dymola.

17.2 Installation on Linux

This section refers only to the Linux version of Dymola.

This section covers Linux-specific parts of the installation. For general items, e.g. how to handle the Dymola installation wizard; please see corresponding section on Windows installation, in particular section "Installing the Dymola license file" on page 1552.

The default directory for installation on Linux is `/opt/dymola-<version>-x86_64`. As an example, the default directory for installation of Dymola 2023 on Linux is `/opt/dymola-2023-x86_64` (the package manager on the target system however typically allows choosing another default location).

Dymola 2023 runs on Red Hat Enterprise Linux 8.2, 64-bit, with gcc version 8.3.1, and compatible systems. (For more information about supported platforms, do the following:

- Go to https://doc.qt.io/
- Select the relevant version of Qt, for Dymola 2023 it is 5.15. For earlier versions of Dymola, to find the relevant Qt version, see the corresponding Dymola Release Notes.
- Select Supported platforms)

Any later version of gcc is typically compatible.

In addition to gcc, the model C code can also be compiled by clang.

You can use a dialog to select compiler, set linker flags, and test the compiler by the **Verify Compiler** button, like in Windows. This is done by the command **Simulation > Setup**, in the **Compiler** tab.

You can however still change compiler by changing the `CC` in

```
/opt/dymola-<version>-x86_64/insert/dsbuild.sh
```

Dymola 2023 is supported as a 64-bit application on Linux. Corresponding support for 32-bit and 64-bit export and import of FMUs is included.

For Dymola 2023, the default setting is to use 64-bit Dymola when translating models.

Notes

- 32-bit compilation for simulation might require explicit installation of 32-bit libc. E.g. on Ubuntu: `sudo apt-get install g++-multilib libc6-dev-i386`
- Dymola is built with Qt 5.15.0 and thereby inherits the system requirements from Qt. This means:
    - Since Qt 5.15 no longer supports embedding of the XCB libraries, these must now be present on the platform running Dymola. See the table in https://doc.qt.io/qt-5.15/linux-requirements.html for the list of versions of the ones starting with "libxcb". Note that the development packages ("-dev") mentioned outside the table are not needed.
    - The library `libxcb-xinput.so.0` might require explicit installation.

- For FMU export/import to work, zip/unzip must be installed.

Please also note that you have to use the Optimization library version 2.x or higher to use multi-criteria design optimization on Linux; the older Design.Optimization package does not support multi-criteria design optimization on Linux.

The library UserInteraction is not supported on Linux.

More Linux-specific notes are available using the command

```
man dymola
```

17.2.1 Installing Dymola

Dymola for Linux is distributed as an RPM package. The package is installed using the command

```
# rpm –i name-of-distribution.rpm
```

Optional libraries are installed through separate RPM files.

For installation on e.g. Debian or Kubuntu systems conversion to the `deb` format is required using the `alien` command:

```
# alien –cki name-of-distribution.rpm
```

Setup and environment variables

The shell script `/usr/local/bin/dymola-<version>-x86_64` (see above concerning "-<version>") contains commands to set environment variables before starting Dymola, but will need editing if Dymola is installed in a non-standard location; then the following environment variables must be defined in order to run Dymola:

DYMOLA Directory root of the distribution (`/opt/dymola-<version>-x86_64`).

DYMOLAPATH Search path for additional Dymola libraries and the license file. The directories of the path may be separated by blanks or colon. `DYMOLAPATH` is optional if the license file is in `$DYMOLA/insert`.

MODELICAPATH Search path for libraries. Concerning the use of MODELICAPATH, please see section "Managing library locations" starting on page 1562.

(Dymola defines an environment variable DYMOLAWORK which value is the Dymola startup directory. For more info, see section "Location of startup directory" on page 1557. Note that the startup directory/current working directory cannot be an UNC path in Linux.)

17.2.2 Additional setup

Subjects in the corresponding section on Windows are not applicable unless explicitly referenced from here.

Compilation of model code

Dymola produces C code which must be compiled in order to generate a simulation model. On Linux systems we rely on an ANSI/ISO C compiler already installed on the computer.

On Linux systems the compilation of the generated C code is performed by a shell script, `/opt/dymola-<version>-x86_64/insert/dsbuild.sh` (see above concerning "-<version>"). If necessary this script can be modified to provide special options to the compiler, add application-specific libraries etc. Simulation performance can be improved by tuning the compilations options in this script, however note that the compiler time may increase significantly by doing so.

Dymola supports external C libraries on Linux. Classes which contain "Library" annotations to link with external libraries in C are supported.

Simulation from the command line

The simulator executable `dymosim` can be executed from the shell. To do so the environment variable LD_LIBRARY_PATH must be set:

```
# export LD_LIBRARY_PATH=/opt/dymola-<version>-x86_64/bin/lib
```

Security-Enhanced Linux (SELinux) might display the message (below example for 64-bit application):

```
dymosim: error while loading shared libraries: /opt/dymola-
2023-x86_64/bin/lib/libds.so: cannot restore segment prot after
reloc: Permission denied
```

If this message is displayed the following commands must be executed (for this 64-bit example):

```
# chcon -t textrel_shlib_t /opt/dymola-2023-
x86_64/bin/lib/libds.so
# chcon -t textrel_shlib_t /opt/dymola-2023-
x86_64/bin/lib/libGodessMain.so
```

Note that running simulations in the Dymola environment do not require these changes.

Working with a Modelica version that is newer than the one in the distribution

Please see corresponding section for Windows installation.

Customizing the File > Libraries menu

Please see corresponding section for Windows installation.

Managing library locations

Please see corresponding section for Windows installation.

On-demand installation of downloaded libraries

Please see corresponding section for Windows installation.

More about libraries and the building of menus of libraries and demos

Please see corresponding section for Windows installation.

Option to use a custom setup file

For Linux the feature is the same as in Windows (see that section), but Linux have other paths; for example, the default path for the default setup file in Linux is, under the user´s home directory:

```
.dassaultsystemes/Dymola/2023/setup.dymx
```

Loading a package with user-defined menus and toolbars that should not be deleted by File > Clear All

Please see corresponding section for Windows installation.

License expiration settings

Please see corresponding section for Windows installation, but note the paths are different in Linux. Also, an option is specified with "-" and not "/".

Preventing checking out license options from a license server

In the corresponding section on Windows the alternative of starting Dymola using a modified command is applicable also for Linux (with relevant changes for Linux). Please see page 1575.

17.2.3 Removing Dymola

Remove the Dymola distribution by using the `rpm -u` command.

17.3 Dymola License Servers on Windows

This section refers only to the Windows version of Dymola.

There are two possible license servers for Dymola on Windows. The default one is FLEXnet Publisher license server. As an alternative, the Dassault Systèmes License Server (DSLS) can be used.

17.3.1 FLEXnet Publisher license server

Background

These are instructions for manually installing a FLEXnet Publisher license server for Dymola on Windows. They only apply to users with a sharable license. For non-sharable licenses (the common case), installation of the license file is automatic.

All files needed to set up and run a Dymola license server on Windows, except the license file, are available in the Dymola distribution, in `Program Files\Dymola 2023\bin`.

Dymola is installed on all machines that will run the software. On the designated machine, the license server is then installed as described below.

The license server consists of two daemon processes:

- The vendor daemon (called `dynasim.exe`) dispenses licenses for the requested features of Dymola (the ability to run Dymola and various options). This program is specific for Dymola.
- The license daemon (called `lmgrd.exe`) sends requests from application programs to the right vendor daemon on the right machine. The same license daemon can be used by all applications from all vendors, as this daemon processes no requests on its own, but forwards these requests to the right vendor daemon. (For use of the newer web based `lmadmin`, see "Using the web based license server manager lmadmin instead of lmgrd" on page 1589.)

If you are already running an application that uses FLEXnet Publisher, you most likely already have a running license daemon. In this case, only the vendor daemon (`dynasim.exe`) is required.

Flexera Software recommends that you use the latest version of the FLEXnet Publisher `lmgrd.exe` at all times as it includes bug fixes, enhancements, and assures the greatest level of compatibility with all of your FLEXnet Publisher licensed applications. Flexera Software guarantees that it will work correctly with all earlier versions of FLEXnet Publisher.

Old license daemons cannot be used!

Dymola requires support of FLEXnet Publisher version 11.16.2.1. A recent version of `lmgrd.exe` is part of the Dymola distribution.

Installing the license server

This section describes the simple case where we assume there are no other FLEXnet Publisher license daemons. We also assume that the Dymola program itself should not be installed on the server. For updating of a present license server, see separate section below.

To purchase a license server, the relevant host id of the computer where the license server should run must be supplied to your Dymola distributor before purchasing the license. The license that you will receive will contain this information. To find out the host id of that computer, the utility program `hostid.exe` can be used. Please see section "Obtaining a host id" on page 1600 for more information.

1. Before installation of the license server, the Dymola license file (*`filename`*`.lic`) may have to be updated with the actual name (or IP-number) of the server, if the license file contains a line identifying the server:

   ```
   SERVER server.name.here 000102DE37CD
   ```

   The part *`server.name.here`* must be changed to the name of the actual server before installing the license file. It should be noted that the last part (the hostid) cannot be edited by the user.

2. Install *only* the Dymola software component **License server** (see beginning of this chapter). A folder will be created containing all needed files, default `C:\Program Files\Dymola 2023\bin`.

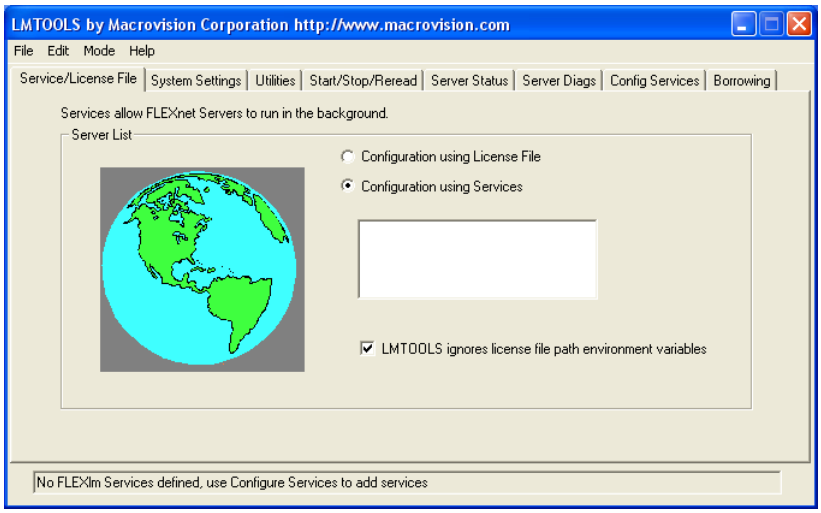
3. Start the utility program `lmtools.exe` (one of the above files).

4. In the **Service/License File** tab:

   a. Select the radio button **Configuration using Services**.

   b. Activate **LMTOOLS ignores license file path environment variables**.

License server setup.



5. In the **Config Services** tab (please see figure on next page):

   a. Enter a new service name, e.g. "Dynasim License Server".

   b. Enter the path to the license daemon, `Dymola 2023\bin\lmgrd.exe`.

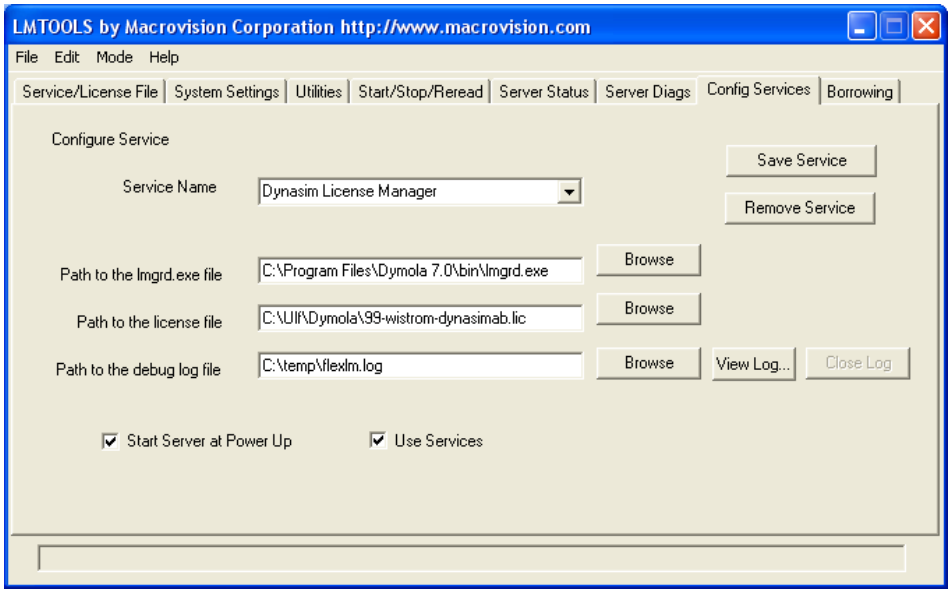
   c. Enter the path to your server license file.

d. Enter the path to a debug log file (anywhere you want).

e. Enable **Use Services** and then **Start Server at Power Up**.

f. Click on **Save Service**. Click on **Yes** to confirm.

Configuration of the license server.



6. In the **Start/Stop/Reread** tab:

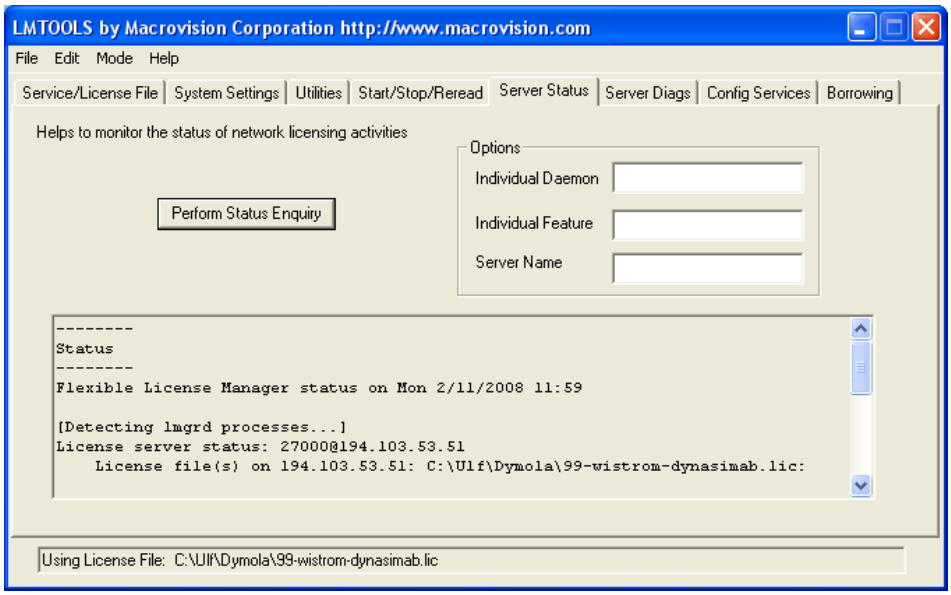
   a. Select the Dynasim license server.

   b. Click on **Start Server**.

7. In the **Server Status** tab:

   a. Click on **Perform Server Enquiry** and check the output of the log window. You should see lines identifying the server processes and what features are available.

Checking the operation of the license server.



b. Also check the log file to verify that the server has started and that Dymola features can be checked out. The following is an example of the FLEXnet Publisher logfile:

```
12:30:48 (lmgrd) pid 2728
12:30:48 (lmgrd) Detecting other license server manager (lmgrd) processes...
12:30:48 (lmgrd) Done rereading
12:30:48 (lmgrd) FLEXnet Licensing (v11.4.100.0 build 50818 i86_n3) started
on 194.103.53.51 (IBM PC) (2/11/2008)
12:30:48 (lmgrd) Copyright (c) 1988-2007 Macrovision Europe Ltd. and/or
Macrovision Corporation. All Rights Reserved.
12:30:48 (lmgrd) US Patents 5,390,297 and 5,671,412.
12:30:48 (lmgrd) World Wide Web:  http://www.macrovision.com
12:30:48 (lmgrd) License file(s): C:\Ulf\Dymola\99-wistrom-dynasimab2.lic
12:30:48 (lmgrd) lmgrd tcp-port 27000
12:30:48 (lmgrd) Starting vendor daemons ...
12:30:48 (lmgrd) Started dynasim (pid 4180)
12:30:48 (dynasim) FLEXnet Licensing version v11.4.100.0 build 50818 i86_n3
12:30:48 (dynasim) Server started on 194.103.53.51 for:  DymolaStandard
12:30:48 (dynasim) DymolaAnimation DymolaModelCalibration
DymolaModelManagement
12:30:48 (dynasim) DymolaOptimization DymolaRealtime     DymolaSimulink
12:30:48 (dynasim) DymolaFlexibleBodiesLib DymolaHydraulicsLib
DymolaPowertrainLib
12:30:48 (dynasim) DymolaSmartElectricDrivesLib
12:30:48 (dynasim) EXTERNAL FILTERS are OFF
12:30:48 (lmgrd) dynasim using TCP-port 2606
12:30:56 (dynasim) TCP_NODELAY NOT enabled
10:39:20 (lmgrd) Detecting other lmgrd processes...
10:39:35 (lmgrd) FLEXlm (v7.2c) started on x.x.x.x (3/27/2001)
```

```
10:39:35 (lmgrd) FLEXlm Copyright 1988-2000, Globetrotter Software
10:39:35 (lmgrd) US Patents 5,390,297 and 5,671,412.
10:39:35 (lmgrd) World Wide Web:  http://www.globetrotter.com
10:39:35 (lmgrd) License file(s): C:\DAG\dymola.lic
10:39:35 (lmgrd) lmgrd tcp-port 27000
10:39:35 (lmgrd) Starting vendor daemons ...
10:39:35 (lmgrd) Started dynasim (pid 124)
10:39:36 (dynasim) Server started on x.x.x.x for:DymolaStandard
10:39:36 (dynasim) DymolaSampledLib DymolaLiveObjects DymolaRealtime
10:39:36 (dynasim) DymolaSimulink    DymolaAnimation DymolaSupport
10:39:36 (lmgrd) dynasim using TCP-port 1042
```

The license server should now be correctly configured. Please start Dymola to verify correct operation. The FLEXnet Publisher logfile (see above) should contain additional lines showing what features were checked out. You can also do **Perform Status Enquiry** to check how many licenses are currently checked out.

Note. The license server by default uses the ports 27000-27009. They can be configured if needed, e.g. if there are issues with firewalls.

Using the web based license server manager lmadmin instead of lmgrd

From Dymola 2021 you can use the web based license server manager `lmadmin` instead of the older command line based `lmgrd`, for both Windows and Linux.

Note. The Flexera FlexNet license server manager `lmadmin` is not included in the Dymola distribution; you must download this software yourself.

Some short notes about the handling:

- start: run lmadmin/lmadmin
- access: http://localhost:8090
- stop: Administration – Server Configuration – Stop Server
- log on first time: user: “admin”, passwd: “admin”

For more information, please see the documentation for Flexera Software LCC FlexNet Publisher Licensing Toolkit 11.16.

Installation of license server when a license server already exists on the target machine

Limiting ourselves to the case of a single server, there are two strategies handling multiple vendor daemons:

- Use a single license server manager
- Use one license server manager per vendor daemon

How you should add a vendor daemon for the first case depends on the license server manager used:

- If using lmgrd, a restart is required
- If using lmadmin, the new daemon can be added via the GUI

For the second case above (using one license server manager per vendor daemon), you must deal with the clashing of listening ports for the multiple license server managers. However, the license server manager will, during startup, alert such clashes. Hence, a simple strategy would be to first try the default port and upon failure explicitly try ports in the default range (27000-27009) or some other range, depending on the IT department port blocking policies.

For more information, please see the documentation for FlexNet Publisher 2016, in particular the section “Managing Licenses from Multiple Software Publishers” in fnp_LicAdmin.pdf.

Updating the license server with new license file

When receiving a new Dymola license file (`filename.lic`), first perform step 1 in the first section “Installing the license server” above.

Now you can replace the old license file with the new (it might be good to temporarily keep the old one by adding e.g. _previous to the name of the old file, in case you must go back to that one).

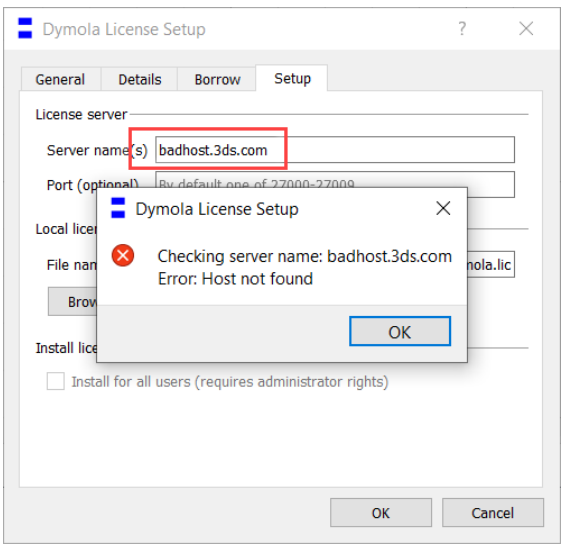
Now you can start the utility program lmtools.exe. In the Start/Stop/Reread tab, click the button **ReRead License File**.

The license server is now updated.

Connecting to the license server host

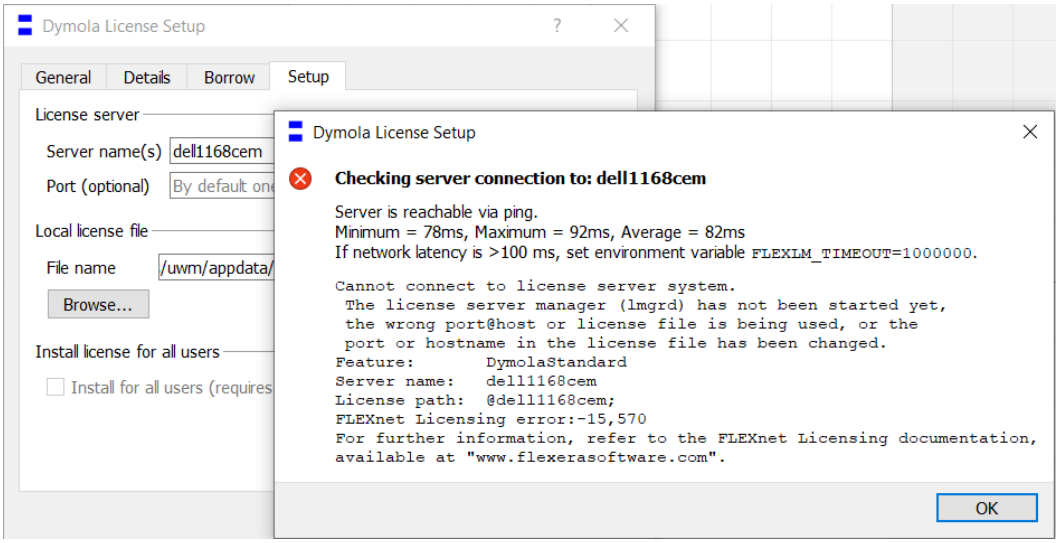
Testing the license server at startup

When trying to connect to the license server host, a basic DNS lookup is performed. If the host cannot be found, the following message appears:



If the host is found, but Dymola fails to contact the license server, a ping test is performed to check if the network is alive, and if timing problems occur. Also, a test checkout of Dymola

Standard license is requested from this server. An example with successful ping test but failed Dymola license checkout is:



Note the tip about the environment variable to increase the time of the network timeout, if needed.

It is still possible to change the license file. You do this by answering **Change Server Settings** on the question whether to update the license server. This question follows clicking **OK** in the dialog above. This can be used for additional testing.

License borrowing

Overview

Dymola on Windows can support "borrowing", the possibility to transfer a license from a license server to laptop for a limited period of time. If Dymola is used on a computer that is intermittently disconnected from a license server, that license can be issued as a sharable license with borrowing facility. Such a license can be borrowed from a license server via a special checkout and used later to run an application on a computer that is no longer connected to the license server.

For license borrowing, an end user initiates borrowing and specifies the expiration date a borrowed license is to be returned to the sharable license pool. While still connected to the network, the application is run from the client computer. This writes licensing information locally onto the client computer. The client computer can now be disconnected from the network.

The license server keeps the borrowed license checked out. The client application automatically uses the local secured data file to do checkouts during the borrowing period. Upon the expiration of the borrowing period or the early return of a borrowed license, the local data file no longer authorizes checkouts and the license server returns the borrowed

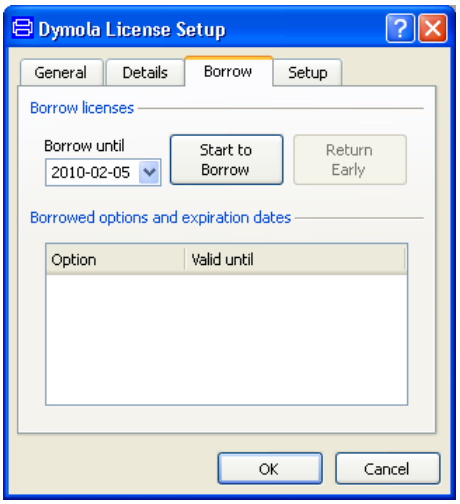
license to the pool of available licenses. No synchronization is required between the license server machine and the client machine when the borrowing period expires.

License borrowing

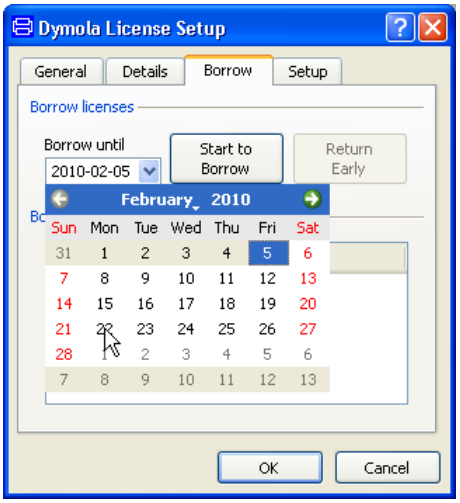
License borrowing and early returns are performed from Dymola.

In order to borrow, do the following:

1. While Dymola is connected to the server, use the command **Tools > License Setup**, and select the **Borrow tab**.

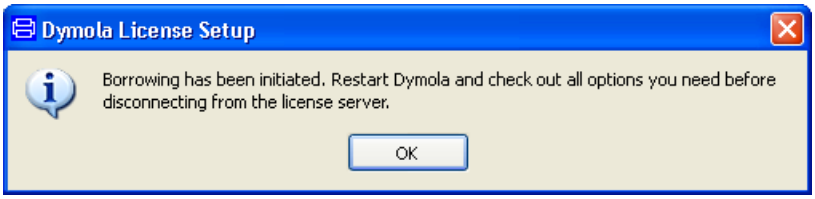


2. Select an end date, either by changing the date in the input field for **Last date borrowed** or by clicking on the arrow to display a calendar for selection of date. Clicking the arrow will display:



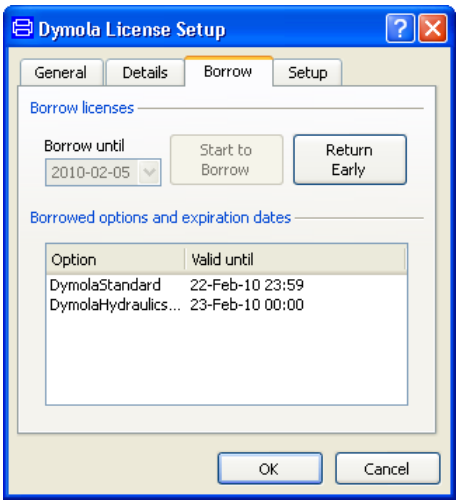
Here the possible selection of dates is clearly visible. Clicking on a date will change the input field to that date.

3. Click on **Start to Borrow**. The following message will appear:



4. Click **OK** and **OK** and restart Dymola (while still connected to the server); now the basic borrowing is performed. (Borrowing will be indicated in several ways, please see next section.)

5. Open all libraries/options that you will need during your borrowing time. This will ensure that the appropriate license features are stored locally. The list in the lower half of the dialog displays currently borrowed licenses and when they will be automatically returned to the server.



In this example, the Hydraulics library was opened; DymolaStandard indicates borrowing of Dymola without any options.

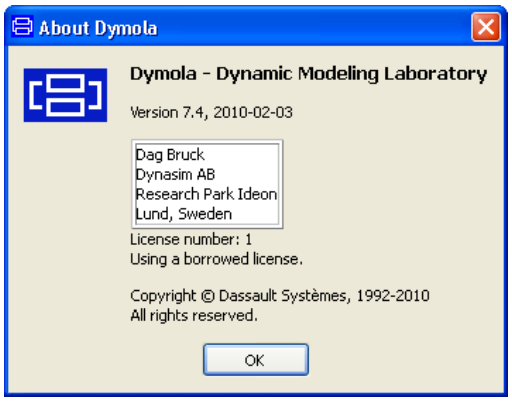
Please be careful not to open libraries/options that might be needed for others unless you really intend to do so. (Borrowing an option only available for one user only might not be appreciated by others.)

6. Finally disconnect from the license server **while Dymola is still running**. This step will create the local license file with the borrowed license. After disconnecting Dymola can be stopped.

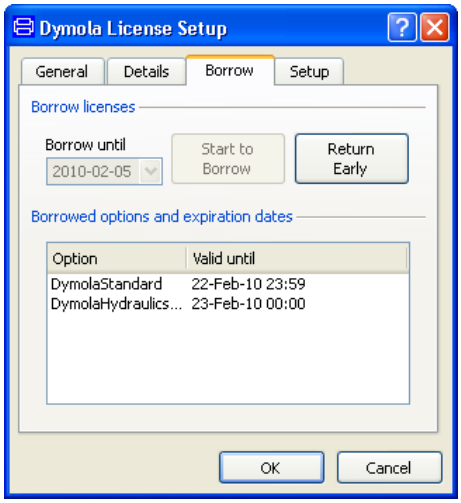
Running Dymola

During the borrowing period, Dymola can be started and stopped as often as needed. When license borrowing is used, Dymola displays it on the splash screen shown when starting Dymola and when using the command **Tools > About Dymola**:

Borrowing period in About dialog.



Most information is given using the command **Tools > License Setup**, in the **Borrow** tab.

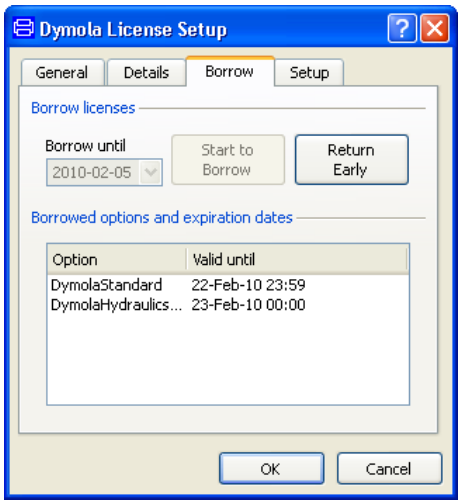


Returning a license before expiration of borrowing (early return)

Currently borrowed licenses can be returned early when the computer is connected to the license server again.

In order to do an early return, do the following:

While Dymola is connected to the server, use the command **Tools > License Setup**, and select the **Borrow tab**.



Now click on **Return Early**. The license (including all listed options) is returned to the server. Next time Dymola is restarted, the license is checked out the usual way.

It is a good idea to check e.g. the splash screen when starting up to convince oneself that the return was successful (in that case borrowing will not be mentioned in the splash screen).

A license returned to the license server cannot be checked out again until after approximately 2 minutes. If licenses are returned by e.g. exiting Dymola, but Dymola is restarted within approximately 2 minutes, the return is never performed.

License server options file

FLEXnet include tools for the local administrator. The options file allows the license administrator to control various operating parameters of the Dymola license server.

For example, it allows the administrator to

- Allow or deny the use of options by users.
- Reserve licenses for specified users.
- Control how many licenses can be borrowed and for how long.

The options file shall be called `dynasim.opt` and placed in the same directory as the Dymola license file of the license server.

An example of an options file that reserves a Dymola + Hydraulics library license for the user Bob is

```
RESERVE 1 DymolaStandard USER Bob
RESERVE 1 DymolaHydraulics USER Bob
```

Applicable “feature” and user names can be found in the license server log file. The details of the options file are described in Chapter 5 of “FLEXnet Licensing End User Guide”, which is available on request.

17.3.2 Dassault Systèmes License Server (DSLS)

General

As an alternative to the FLEXnet license server on Windows, Dassault Systèmes License Server (DSLS) is now also supported for Dymola, for sharable and nodelocked licenses.

Some notes about installation and use:

- To generate a target (host) id to order a license key, start Dymola with the command `dymola.exe /DSLS` (without license) and then, in Dymola, use the command **Tools > License Setup > Details** to copy the target id.
- A nodelocked DSLS license key should be saved in `C:\ProgramData\DassaultSystemes\Licenses`.
- The DSLS license server can be downloaded from the same location as the Dymola media. After downloading, you can install it as described in the DSLS documentation.
- To start Dymola with DSLS, use the command `dymola.exe /DSLS`
- To connect to an existing DSLS license server, use, in Dymola, the command **Tools > License Setup > Setup**. Type in the server name and press **OK**. The default port number can be used in almost any case.
- Dymola checks the contact to the license server every 5 minutes. If the connection is lost, increasingly severe messages are displayed. If the connection is down for 20 minutes, Dymola gives a final message before termination. Note that, even at this point, you are given the opportunities to save what has been modified, but the termination cannot be canceled.

Limitations:

- Borrowing of licenses (the Dymola command **Tools > License Setup > Borrow**) is not supported.
- Running FMUs that were exported without export license is not supported.

For more details, see the Knowledge Base article QA00000061268. Note that a DS Passport is needed to see this article.

Cloud service for license management (Managed DSLS)

Cloud service for license management, that is, the license server is managed for you by Dassault Systèmes, is supported from Dymola 2022x. The feature is sometimes referred to as "Managed DSLS". The advantages for you are:

- No need to manage serves or license keys
- Automatic renewal of license keys
- Available on high-availability cloud server
- No additional cost

The requirements for “Managed DSLS” are:

- Support is active
- Access to Internet (HTTPS port) while Dymola is running

17.4 Dymola License Server on Linux

This section refers only to the Linux version of Dymola.

There are two possible license servers for Dymola on Linux. The default one is FLEXnet Publisher license server. As an alternative, the Dassault Systèmes License Server (DSLS) can be used.

17.4.1 FLEXnet Publisher license server

This section covers Linux-specific parts of the Dymola license server. For general items, e.g. background and how to set up the server using `lmtools.exe`, please see corresponding section on Dymola License Server on Windows.

Note! Dymola requires support of FLEXnet Publisher version 11.16.2.1. This version is part of the Dymola distribution for Linux.

The Linux license server for Dymola is located in a separate tar file.

To start the server, the dynasim and the lm* files need to be installed, for example in `/usr/local/bin`. The server is started with the command

```
lmgrd -c <path to license file> -l <path to logfile>
```

A check with `pg aguxf` should show two new processes, `lmgrd` and `dynasim`. The server status can be checked with `lmutil lmstat -a`. In case of problems the log file should be examined. Note also the general license server check at startup, see section “Testing the license server at startup” on page 1590.

Note that it is possible from Dymola 2021 to use the web based license server `lmadmin` instead of the command line based `lmgrd` for both Linux and Windows. Please see section “Using the web based license server manager lmadmin instead of lmgrd” on page 1589 for more details.

To start the license server automatically when the system is rebooted, please update e.g. `/etc/rc.d/rc.local` accordingly. Note that the license server needs not to run as “root”.

Note. The license server by default uses the ports 27000-27009. They can be configured if needed, e.g. if there are issues with firewalls.

Full details of FLEXnet license server installation can be found in the FLEXnet User’s Manual, which can be downloaded from www.flexera.com.

License borrowing on Linux

License borrowing is enabled by setting the environment variable LM_BORROW. The value must specify beginning and end dates of the borrowing period, as well as the vendor name "dynasim". The general format is:

```
LM_BORROW=<start date>:dynasim:<end date>
```

An example (using bash) which specifies the start date 10 November 2009 and the end date 12 November 2009 is:

```
export LM_BORROW=10-nov-2009:dynasim:12-nov-2009
```

After setting the environment variable LM_BORROW, Dymola must be restarted and the appropriate license options checked out before disconnecting from the license server.

The status of borrowing can be displayed in the Linux server using a status command. An example, for Dymola 2023, is:

```
/opt/dymola-2023-x86_64/bin/lmutil lmborrow -status
```

The command displays the names of borrowed features and the expiration dates.

Returning a license before expiration of borrowing (early return)

Currently borrowed licenses can be returned early when the computer is connected to the license server again.

The names of the features that are currently borrowed can be seen using the status command in the previous section. When returning, any of these names must be used in the return command below.

In order to do an early return, give a return command while Dymola is connected to the server. An example returning the license for Pneumatics Library, for Dymola 2023, is:

```
/opt/dymola-2023-x86_64/bin/lmutil lmborrow -return -c ~/.dynasim/
    dymola.lic DymolaPneumaticsLib
```

Whether the return was made can be seen using the status command in previous section.

A license returned to the license server cannot be checked out again until after approximately 2 minutes. If licenses are returned by e.g. exiting Dymola, but Dymola is restarted within approximately 2 minutes, the return is never performed.

17.4.2 Dassault Systèmes License Server (DSLS)

General

As an alternative to the FLEXnet license server on Linux, Dassault Systèmes License Server (DSLS) is now also supported for Dymola, for sharable and nodelocked licenses.

Some notes about installation and use:

- To generate a target (host) id to order a license key, start Dymola with the command `dymola.exe /DSLS` (without license) and then, in Dymola, use the command **Tools > License Setup > Details** to copy the target id.

- A nodelocked DSLS license key should be saved in `var/DassaultSystemes/Licenses`.
- The DSLS license server can be downloaded from the same location as the Dymola media. After downloading, you can install it as described in the DSLS documentation.
- To start Dymola with DSLS, use the command `dymola.exe /DSLS`
- To connect to an existing DSLS license server, use, in Dymola, the command **Tools > License Setup > Setup**. Type in the server name and press **OK**. The default port number can be used in almost any case.
- Dymola checks the contact to the license server every 5 minutes. If the connection is lost, increasingly severe messages are displayed. If the connection is down for 20 minutes, Dymola gives a final message before termination. Note that, even at this point, you are given the opportunities to save what has been modified, but the termination cannot be canceled.

Limitations:

- Borrowing of licenses (the Dymola command **Tools > License Setup > Borrow**) is not supported.
- Running FMUs that were exported without export license is not supported.

For more details, see the Knowledge Base article QA00000061268. Note that a DS Passport is needed to see this article.

Cloud service for license management (Managed DSLS)

Cloud service for license management, that is, the license server is managed for you by Dassault Systèmes, is supported from Dymola 2022x. The feature is sometimes referred to as "Managed DSLS". The advantages for you are:

- No need to manage serves or license keys
- Automatic renewal of license keys
- Available on high-availability cloud server
- No additional cost

The requirements for "Managed DSLS" are:

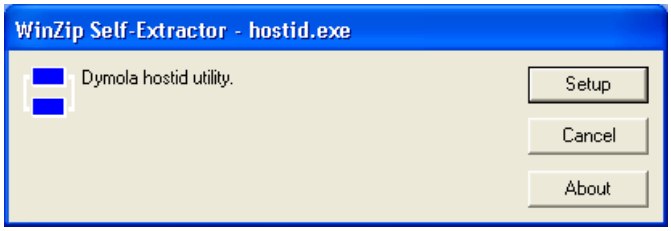
- Support is active
- Access to Internet (HTTPS port) while Dymola is running

17.5 Utility programs

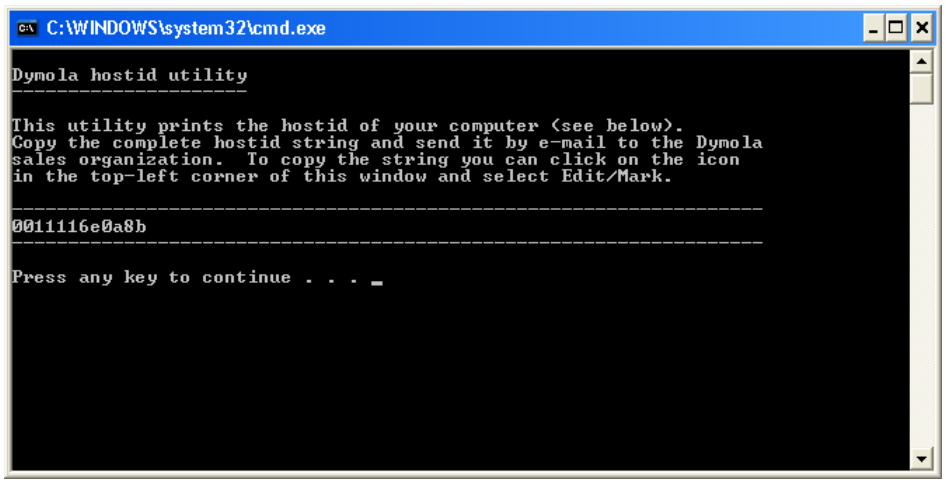
17.5.1 Obtaining a host id

To be able to easy find out the host id of a computer without having Dymola installed, a small file `hostid.exe` can be obtained from your Dymola distributor. (If Dymola trial version is installed, the host id can also be found using Dymola, please see section “Obtaining a host id” on page 1554.)

Executing this file (by double-clicking it or opening it), the following menu will be displayed:



Selecting **Setup** will display the following:



Clicking in the upper left corner and selecting **Edit > Mark** makes it possible to selecting the host id by dragging the cursor over it. Once selected, **Edit > Copy** will place the host id in the clipboard, from where it should be pasted into a mail to your Dymola distributor.

17.6 System requirements

17.6.1 Hardware requirements

- At least 2 GB RAM
- At least 400 MB disc space

17.6.2 Hardware recommendations

At present, it is recommended to have a system with an Intel Core 2 Duo processor or better, with at least 2 MB of L2 cache. Memory speed and cache size are key parameters to achieve maximum simulation performance.

A dual processor will be enough if not using multi-core support; the simulation itself, by default, uses only one execution thread so there is no need for a "quad" processor. If using multi-core support, you might want to use more processors/cores.

Memory size may be significant for translating big models and plotting large result files, but the simulation itself does not require so much memory. Recommended memory size is 6 GB of RAM.

17.6.3 Software requirements

Microsoft Windows

Dymola versions on Windows and Windows operating systems versions

Dymola 2023 is supported, as a 64 bit application, on Microsoft Windows 8.1 and Windows 10. Since Dymola does not use any features supported only by specific editions of Windows ("Home", "Professional", "Enterprise" etc.); all such editions are thus supported if the main version is supported.

Compilers

Please note that for the Windows platform, a Microsoft C/C++ compiler, or a GCC compiler, must be installed separately. The following compilers are supported for Dymola 2023 on Windows:

Visual Studio - Free editions:

Note. When installing any Visual Studio compiler, make sure that the option "C++/CLI support…" is also selected to be installed.

- Visual Studio 2012 Express Edition (11.0)
- Visual Studio 2015 Express for Windows Desktop Edition (14.0)
- Visual Studio 2017 Desktop Express **Note!** This compiler only supports compiling to Windows 32-bit executables
- Visual Studio Community 2017
- Visual Studio 2017 Build Tools **Notes:**

    - The recommended selection to run Dymola is the workload “Visual C++ build tools” + the option “C++/CLI support…”
    - Installing this selection, no IDE (Integrated Development Environment) is installed, only command line features
    - This installation is not visible as specific selection when later selecting the compiler in Dymola, the alternative to select is the same as for any Visual Studio 2017 alternative: **Visual Studio 2017/Visual C++ 2017 Express Edition (15)**.
    - For more information about installing and testing this compiler with Dymola, see www.Dymola.com/compiler.
- Visual Studio Community 2019
- Visual Studio 2019 Build Tools **Notes:**
    - The recommended selection to run Dymola is the workload “C++ build tools” + the option “C++/CLI support…”
    - Installing this selection, no IDE (Integrated Development Environment) is installed, only command line features
    - This installation is not visible as specific selection when later selecting the compiler in Dymola, the alternative to select is the same as for any Visual Studio 2019 alternative: **Visual Studio 2019/Visual C++ 2019 (16)**.
    - For more information about installing and testing this compiler with Dymola, see www.Dymola.com/compiler.

Visual Studio - Professional editions:

Note. When installing any Visual Studio compiler, make sure that the option “C++/CLI support…” is also selected to be installed.

- Visual Studio 2012 (11.0)
- Visual Studio 2015 (14.0)
- Visual Studio Professional 2017 (15)
- Visual Studio Enterprise 2017 (15)
- Visual Studio Professional 2019 (16)
- Visual Studio Enterprise 2019 (16)

GCC compilers:

The following versions have been tested (hence, at least the versions in that range should work fine):

- 32-bit MinGW, with GCC version 5.3, 6.3, and 8.2
- 64-bit MinGW, with GCC version 5.3, 7.3, and 8.1

Note – the GCC compilers have some limitations, and demand for add-ons during installation etc., please see section “GCC compiler” on page 1548.

Linux cross-compiler (WSL):

Dymola on Windows supports cross-compilation for Linux via the use of Windows Subsystem for Linux (WSL). The default WSL setup is 64-bit only and Dymola adopts this limitation. For more information, see "Cross-compilation for Linux on Windows" starting on page 1576.

Linux

Supported Linux versions and compilers

Dymola 2023 is supported on Red Hat Enterprise Linux 8.2, 64-bit, with gcc version 8.3.1, and compatible systems (For more information about supported platforms, do the following:

- Go to https://doc.qt.io/
- Select the relevant version of Qt, for Dymola 2023 it is 5.15. For earlier versions of Dymola, to find the relevant Qt version, see the corresponding Dymola Release Notes.
- Select Supported platforms)

Any later version of gcc is typically compatible.

In addition to gcc, the model C code can also be compiled by clang.

You can use a dialog to select compiler, set linker flags, and test the compiler by the **Verify Compiler** button, like in Windows. This is done by the command **Simulation > Setup**, in the **Compiler** tab.

You can however also still change the compiler by changing the variable `CC` in

```
/opt/dymola-2023-x86_64/insert/dsbuild.sh
```

Dymola 2023 is supported as a 64-bit application on Linux.

Notes:

- 32-bit compilation might require explicit installation of 32-bit libc. E.g. on Ubuntu: `sudo apt-get install g++-multilib libc6-dev-i386`
- Dymola is built with Qt 5.15.0 and thereby inherits the system requirements from Qt. This means:
    - Since Qt 5.15 no longer supports embedding of the XCB libraries, these must now be present on the platform running Dymola. See the table in https://doc.qt.io/qt-5.15/linux-requirements.html for the list of versions of the ones starting with "libxcb". Note that the development packages ("-dev") mentioned outside the table are not needed.
    - The library `libxcb-xinput.so.0` and might require explicit installation.
- For FMU export/import to work, zip/unzip must be installed.

Notes on libraries

- Please note that you have to use the Optimization library version 2.x or higher to use multi-criteria design optimization on Linux; the older Design.Optimization package does not support multi-criteria design optimization on Linux.
- The library UserInteraction is not supported on Linux.

17.6.4 Dymola license server

FLEXnet Publisher license server

For a Dymola license server on Windows or Linux, all files needed to set up and run a Dymola license server on Windows or Linux, except the license file, are available in the Dymola distribution. (This includes also the license daemon, where Dymola presently supports FLEXnet Publisher version 11.16.2.1. This version is part of the Dymola distribution.)

Dymola 2023 and later.

Dymola 2023 and later use FLEXnet Publisher version 11.16.2.1 which supports at least the following operating systems:

Microsoft Windows 32-bit:

- Windows 10
- Windows Server 2019
- Windows Server 2016
- Windows 7 SP1

Microsoft Windows 64-bit:

- Windows 10
- Windows Server 2019
- Windows Server 2016
- Windows 7 SP1

Linux 32-bit:

- RedHat Enterprise Linux 6 and 7
- SUSE Linux Enterprise 11 SP4 and 12 SP3

Linux 64-bit:

- RedHat Enterprise Linux 6, 7, and 8[24]
- SUSE Linux Enterprise 11 SP4, 12 SP3, and 15

[24] For RedHat Enterprise Linux 8 (RHEL8) we have confirmed that the 64-bit license server works fine but not the 32-bit. To support also RHEL8, we have therefore added the 64-bit variant under bin64 in the Dymola distribution.

- Ubuntu 16.04, 18.04, and 18.10

Dymola 2018 – Dymola 2022x

Dymola 2018 up to, and including, Dymola 2022x, use FLEXnet Publisher version 11.14.0.2 which supports at least the following operating systems:

Microsoft Windows 32-bit:

- Windows 10
- Windows Server 2008, including SP1 and SP2
- Windows 8.1
- Windows 8
- Windows 7, including SP1
- Windows Server 2012 R2
- Window Server 2012

Microsoft Windows 64-bit

- Windows 10
- Windows Server 2008, including SP1, SP2, and R2
- Windows 8.1
- Windows 8
- Windows 7, including SP1
- Windows Server 2012 R2
- Window Server 2012

Linux 32-bit:

- RedHat Enterprise Linux 6 and 7
- SUSE Linux Enterprise 10, 11, and 12

Linux 64-bit:

- RedHat Enterprise Linux 6, 7, and 8[25]
- SUSE Linux Enterprise 10, 11, and 12

[25] For RedHat Enterprise Linux 8 (RHEL8) we have confirmed that the 64-bit license server works fine but not the 32-bit. To support also RHEL8, we have therefore added the 64-bit variant under bin64 in the Dymola distribution.

Dassault Systèmes License Server (DSLS)

As an alternative to FLEXnet Publisher license server, the Dassault Systèmes License Server (DSLS) can be used. Dymola 2023 supports DSLS R2022x. Earlier DSLS versions cannot be used.

Note that borrowing of licenses is not supported for this license server.

17.7 License requirements

17.7.1 General

This description covers the license requirements for different features; for description of the features themselves, please see relevant documentation in chapter "Simulation Environments", and the documentation for the libraries.

This description assumes that machines having Dymola installed have the Dymola **Standard Configuration** license.

Apart from the **Standard Configuration** license, there are other licenses for different categories of users:

- **Trial** license. This license has some limitations compared with the Standard Configuration license.
- Licenses for academia (**Academic Learn/Innovate**, **Student Learn/Innovate**) The Student licenses have some limitations compared to the Academic ones, for example, it is not possible using a Student license to run dymosim.m from Matlab/Simulink or dymosim.exe from a command window. You need any of the Academic licenses to do that.

Licenses can be node-locked (to a certain machine) or sharable (accessible from a license server). Note that node-locked and sharable licenses cannot be mixed.

17.7.2 License for Dymola – Simulink interface

To use the traditional connection between Dymola and Simulink (Dymola-on-Windows Mode), you need a **Simulink Interface** license in Dymola.

Note that this mode is not available for Linux. For Linux, see section "Import to Simulink"; the second part.

17.7.3 License for real-time simulation

In Dymola

To be able to enable inline integration, needed for real-time simulation, you must have the **Real-Time Simulation** license. (The **Source Code Generation** license includes this

functionality as well.) The **Real-Time Simulation** license is from Dymola 2017 FD01 included in the Dymola standard license.

On dSPACE and xPC Target platforms

To be able to use real-time simulation on dSPACE or xPC Target platforms, you must have

- The **Simulink Interface** license in Dymola to use the Dymola-Simulink interface.
- Any of the licenses **Real-Time Simulation** or **Source Code Generation**, depending on what you want to do:
    - If you only want to use models that use inline integration, the **Real-Time Simulation** license is sufficient. This license is from Dymola 2017 FD01 included in the Dymola standard license.
    - If you want to use other models as well, you need the **Source Code Generation** license.

17.7.4 Licenses for exporting code

The assumption is that the model is to be imported/executed on another machine than the one where it was created.

Exporting a Dymola model to a machine without a Dymola license

See the corresponding import section below, that also explains the export.

Exporting a model to Simulink

In Windows you can use the "Dymola-on-Windows" mode of the Dymola-Simulink interface (using DDE communication between Dymola and Simulink). In this mode, you can export a compiled model and execute it on another Windows computer. You don´t need any license to export the compiled model, but there might be requirements depending on how you want to use it – see the section about importing a model to Simulink below.

For other platforms (for example Linux) – but also optionally in Windows – you can use the "Import" mode of the Dymola-Simulink interface. You can export models to be imported in this mode. To do this, you must use the ExternalInterfaces library, the `ExternalInterfaces.Export.toSimulink` function. You need the **Simulink Interface** license to do this. Depending on how the model should be used, there are additional requirements, see section "Importing a model to Simulink" below.

Exporting a model for integration with Real-Time Workshop (dSPACE and xPC)

You can do this by using the ExternalInterfaces library, the `ExternalInterfaces.Export.toSimulink` function. You need to have both the **Simulink Interface** license, and the **Real-Time Simulation** license to do such an export. (If you have a **Source Code Generation** license, that one includes **Real-Time Simulation** as well.) The **Real-Time Simulation** license is from Dymola 2017 FD01 included in the Dymola standard license.

Exporting a model to FMU format

See the section “Importing an FMU” below; this section also explains the export.

Exporting a model to Binary

You can export a model to binary. This requires the license **Binary Model Export**. (The **Source Code Generation** license includes this functionality as well.) Such an exported model can be executed on a machine without having any Dymola license.

Exporting a model to Source Code (C code)

You can export a model to source code. This requires the license **Source Code Generation**. Such an exported model can be executed on a machine without having any Dymola license. Additional features are available, see the chapter “Simulation Environments”.

17.7.5 Licenses for executing/importing/using code

Executing a Dymola model on another computer

There are two cases:

- The model has been generated by a machine having a **Binary Model Export** license (or **Source Code Generation** license). In this case no Dymola license is required to execute the model.
- The model has been generated without any of the two licenses above. You can execute the model if the machine has a Dymola **Standard Configuration** license, and the environment variable DYMOLA_RUNTIME_LICENSE is set to the path where the license is located. (This is needed for the executable model to find the license file, see “Specifying the license key by the environment variable DYMOLA_RUNTIME_LICENSE” on page 1555.) **Note.** This alternative is not supported for the GCC compiler; see limitations in section “GCC compiler” starting on page 1548.

Import to Simulink

If you have used the “Dymola-On-Windows” mode to export a compiled model, you can use it in a number of ways:

- You can execute the model on a Windows computer without Dymola installation in two cases:
    - The model has been generated by a machine having a **Binary Model Export** license (or **Source Code Generation** license). In this case, no Dymola license is required to execute the model.
    - The model has been generated without any of the two licenses above. You can execute the model if the machine has a Dymola **Standard Configuration** license, and the environment variable DYMOLA_RUNTIME_LICENSE is set to the path where the license is located (see “Specifying the license key by the

environment variable DYMOLA_RUNTIME_LICENSE" on page 1555). **Note.** This alternative is not supported for the GCC compiler; see limitations in section "GCC compiler" starting on page 1548.

- You can import it in Simulink using the "Import" mode of the Dymola-Simulink interface. This requires that the model has been generated by a machine having a **Binary Model Export** license (or **Source Code Generation** license).

If you have exported a model using the using the ExternalInterfaces library, the `ExternalInterfaces.Export.toSimulink` function, you can import it using the "Import" mode of the Dymola-Simulink interface. Such an imported model can be compiled, simulated, and also downloaded to real-time platforms (the last option requires that it has been exported from a machine having the **Real-Time Simulation** license, or the **Source Code Generation** license). The **Real-Time Simulation** license is from Dymola 2017 included in the Dymola standard license.

Importing an FMU

You have three cases:

- The FMU is a Dymola FMU, and generated by a machine having a **Binary Model Export** license (or **Source Code Generation** license); in this case, you can import it and execute it on a computer without a Dymola license.
- The FMU is a Dymola FMU, and generated by a machine without any of the licenses above; in this case, you must fulfill two conditions to be able to execute the FMU: (**Note.** This alternative is not supported for the GCC compiler; see limitations in section "GCC compiler" starting on page 1548.)
    - The machine you import to must have a Dymola **Standard Configuration** license.
    - The environment variable DYMOLA_RUNTIME_LICENSE must be set to the path where the license is located. (This is needed for the dll of the FMU to find the license file.) For information about this variable, see "Specifying the license key by the environment variable DYMOLA_RUNTIME_LICENSE" on page 1555.
- The FMU is from another vendor. For this case, please see the documentation for that vendor; there are no additional requirements from Dymola.

17.7.6 Licenses for libraries in the Dymola library menu

A number of libraries in the Dymola library menu can be used in Dymola without any additional licenses, a number of libraries require additional licenses when working with more advanced features, and a number of libraries always require additional licenses.

Note:

- If you want to export code, the above requirements for export licenses apply as well.

Free libraries and packages in the Dymola library menu

- DataFiles
- Design.Calibration
- Design.Experimentation
- Design.Validation
- DymolaCommands
- DymolaModels
- FTire Interface Library
- Modelica Reference
- Modelica Standard Library
- Modelica_DeviceDrivers
- Modelica_LinearSystems2
- Modelica_StateGraph2
- ModelManagement
- Plot 3D
- SDF (Scentific Data Format)
- Thermal Management Demos (in the menu **File > Demos**; note that commercial libraries are needed to run the demo)
- Testing Library
- UserInteraction
- Vehicle Demos (in the menu **File > Demos**; note that commercial libraries are needed to run the demo
- VehicleInterfaces

Libraries in Dymola library menu that require additional licenses when for example running more advanced examples

- Optimization (and the older Design.Optimization) – requires **Design Optimization** license when running more advanced examples.

Libraries in Dymola library menu always requiring additional licenses

Libraries in the Dymola library menu not listed above always require additional licenses.

17.7.7 Licenses for libraries not in Dymola library menu

There are a number of libraries not in Dymola library menu, for example:

- Libraries can be downloaded from the homepage of Modelica Association (www.Modelica.org) – some are free, some require license.
- Third party vendors can use the license handling in Modelica to add license requirements on their libraries.

17.8 Troubleshooting

This is a common section for both Windows and Linux. If a problem only is applicable for e.g. Linux, it is stated.

Occasionally the installation will not succeed, or the program will not operate as intended after installation. This section will outline some of the problems that have been detected in the past.

17.8.1 License file

The license file used is not the one wanted

There are a number of standard paths where Dymola searches for a valid license. In an old invalid license is stored by mistake in one of those locations, that license might be tried instead of the correct one. Information about which license is currently in use by Dymola is given using the command **Tools > License Setup > Setup**. The path to that license is specified by `Filename` in that tab.

License file is not authentic

The error message "License file not authentic" indicates either an error in the license file, or a mismatch between your computer system and your license file.

- The license file is locked to your computer system, which means that you cannot execute Dymola from another computer.

- The license file format has been changed in Dymola 7.0 and later versions. If you also have older versions of Dymola installed, please check that you have a new license file as well.

Additional information

If there is some error in the license file or with the license server, Dymola presents a short error message by default. A more detailed description, including FLEXnet Publisher error codes, is produced if Dymola is started with the command line option `/FLEXlmDiag`. On Windows, start a command (DOS) window (using the command **Start > All Programs > Accessories > Command Prompt** in Windows) and issue the following commands (assuming Dymola 2023 64-bit is used on a 64-bit computer):

```
cd \Program Files\Dymola 2023\bin64
dymola.exe /FLEXlmDiag
```

On Linux the command will be:

```
dymola /FLEXlmDiag
```

The additional information is usually helpful in any correspondence with support.

License server

Correct operation of the FLEXnet Publisher license server should be verified with `lmtools.exe`, see "Installing the license server" on page 1585. The FLEXnet Publisher logfile provides additional information about the day-to-day operation of the server.

Always using the latest version of the FLEXnet Publisher license daemon `lmgrd.exe` is strongly recommended. It is guaranteed to be compatible with all earlier versions of FLEXnet Publisher.

License borrowing

Note that borrowing is not supported for the Dassault Systèmes License Server (DSLS).

Different versions of Dymola

There are limitations regarding license borrowing when borrowing is done in one version of Dymola, and using the borrowed license is used in another version of Dymola on the same PC.

For Windows, a license borrowed using Dymola 7.4 FD01 or older cannot be used by Dymola 2012 or newer without being connected to the license server.

For Linux, a license borrowed using Dymola 2012 FD01 or older cannot be used by Dymola 2013 or newer without being connected to the license server.

If access to e.g. both Dymola 7.4 FD01 and Dymola 2012 is required on a Windows PC, both versions must be used to borrow, by the following procedure:

- Initiate borrowing with any Dymola version.
- Open Dymola 7.4 FD01 (or older) and check out the required features.

- Open Dymola 2012 (or newer) and check out the required features.
- Validate by checking that there are two entries of all the required features in the **Details** tab, using the command **Tools > License Setup.**
- Disconnect from the network and validate that both versions can be run as expected.

License borrowing of 32/64-bit Dymola

When borrowing license, only the license of the Dymola version you run will be borrowed: and 64-bit and 32-bit Dymola are seen as different versions. For the few cases when you want to

- Borrow a license, AND
- run Dymosim.exe outside of Dymola, AND
- do not have export option;

we advise that you in 64-bit Dymola generate a 64-bit Dymosim.exe using the flag `Advanced.CompileWith64=2`. For more information about this flag, see the chapter "Simulating a model", section "Dynamic Model Simulator".

Sharable licenses

Please note that if a new session is started in Windows by using **Log Off > Switch User** the original user is still logged on and any Dymola program occupies a sharable license.

17.8.2 Compiler problems

Verify compiler button and error messages

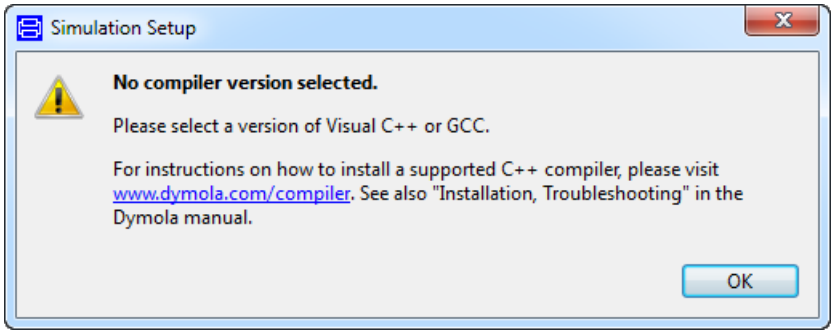
The compiler used to compile the C code generated by Dymola into executable code for simulation is set in the **Compiler** tab using the command **Simulation > Setup…**, see section "Selecting a compiler" on page 1549.

Some potential problems can be found by pressing the **Verify compiler** button in that tab. Any warning messages indicate problems that need to be resolved before translating a model. Pressing the button performs a number of tests:

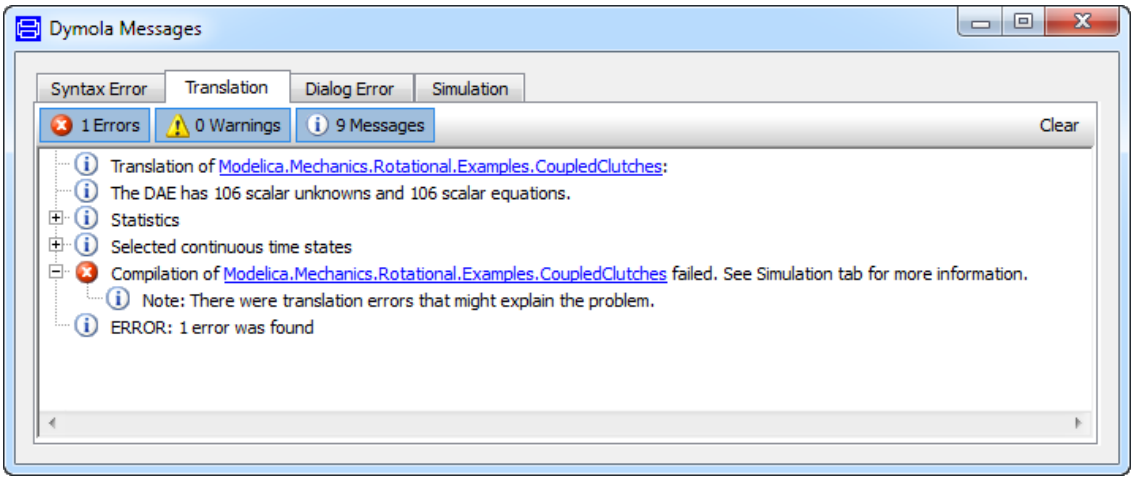
- Validates the DOS environment (for Windows) / shell (for Linux).
- Check the Dymola installation for runtime libraries.
- Verifies that the selected compiler directory contains a valid compiler.
- Validates that the compiler can compile by executing a small test model.
- (For Windows) Validates that an unsupported Visual Studio compiler version has not been selected when using the **Visual Studio Custom** alternative in the setup.

The compiler is tested for both 32-bit and 64-bit mode.

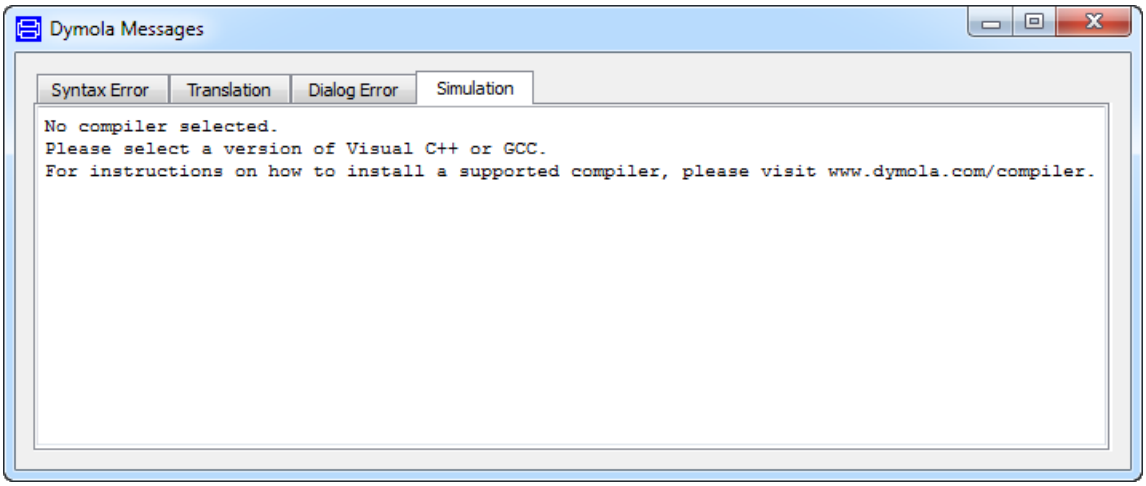
Error messages with information how to proceed (including a link to the web page described below) will be displayed, e.g. when no compiler is selected:



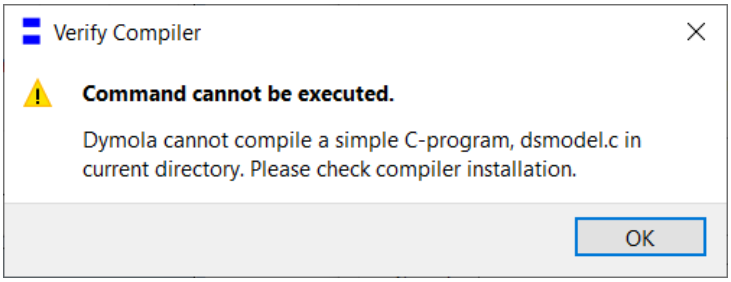
If no compiler has been selected (or installed), the corresponding information will also be displayed in the command log, in the **Translation** tab:



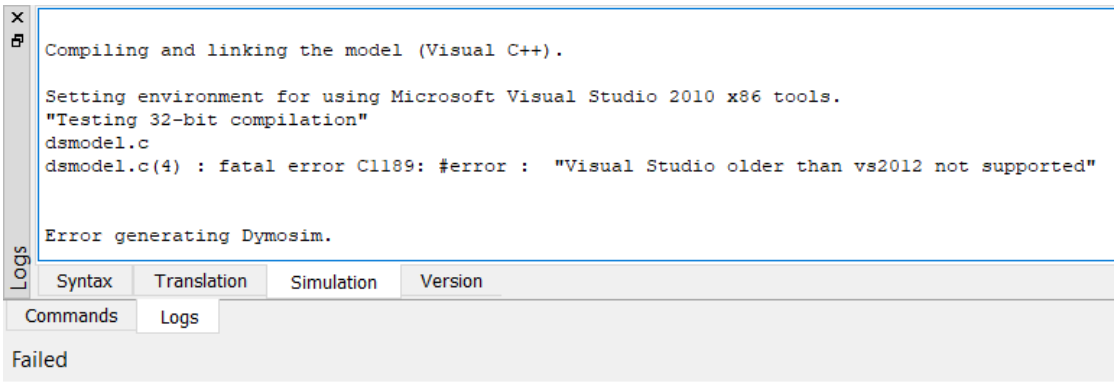
And also in the **Simulation** tab:



If an invalid compiler version has been selected when using the **Visual Studio Custom** alternative in the compiler setup, you will have the following message (in this example, a Visual Studio 2010 compiler has been selected):



In the command log the following message is given:



Dymola webpage for compiler issues

A web page is available for Dymola compiler issues: www.Dymola.com/compiler. This page contains the following information:

- Links to download compilers.
- Extract from the documentation concerning installation and troubleshooting of compilers.
- Additional documentation and information about the compilers.

A link to this page is presented when detecting any error using the **Verify compiler** button and in the build log when no compiler is installed/selected.

17.8.3 Simulink

If the Dymola-Simulink interface does not work, please check the following (some of which may sound elementary):

- You have a Dymola license that supports the Simulink interface. Note that Simulink support is a separate option.
- You have included the three directories `Dymola 2023\mfiles`, `Dymola 2023\mfiles\traj` and `Dymola 2023\mfiles\dymtools` in the Matlab path. These have to be included every time you want to use the Dymola-Simulink interface and it is a good idea to store the included paths in Matlab.
- You can find the interface in Simulink's browser as `Dymola Block/DymolaBlock` (if not, you have probably not included the directories, mentioned above, into the Matlab path).
- Make sure that you have a Visual Studio C++ compiler installed on your computer. Make sure that the Matlab mex utility has been configured to use that compiler (type `mex -setup` in Matlab to configure). Finally, test by trying to compile and link an example mex file, e.g. `matlab\extern\examples\mex\yprime.c`.
- You have created external inputs to the Dymola Block, and outputs from the Dymola Block, in a correct way. See also the chapter “Simulation Environments”, section “Dymola – Matlab interface”, subsection “Using the Dymola-Simulink interface”.
- You have compiled all Dymola models used in the model; otherwise, you will get an error message.
- If “Allow multiple copies of block” is unchecked, you should not copy the block. Unchecking it should only be done if you have a dSPACE system.

Also, note that the parameterizations differ between blocks in the Modelica Standard Library and in Simulink. For example, the frequency of Simulink's Sine-block is measured in rad/s, which is commonly known as angular frequency and should thus be 2π times the frequency in the corresponding source in Modelica.

Only Visual Studio C++ compilers are supported to generate the DymolaBlock S-function. The LCC compiler is not supported.

17.8.4 Change of language

Dymola is available in Japanese. Sometimes the user wants to change the language after installation. Please see section “Language” on page 1556 on how to do this.

17.8.5 Other Windows-related problems

Starting the installation

The installation normally starts automatically when you insert the distribution DVD. If auto start has been disabled, please start `D:\setup.exe` (assuming your DVD drive is labeled D) from Windows Explorer by double-clicking on the file or use the **Start** button in Windows, select **Run**, enter `D:\setup.exe` and click **OK**.

Deep directory hierarchies

Compilation and simulation of the model may fail in a very deep directory hierarchy, if the length of the directory path exceeds 250 characters. This is caused by a bug in Microsoft software, and we are investigating ways to work around it.

Writable root directory

Due to a bug in some versions of the Microsoft runtime library, the root directory C:\ should be writable in order to store temporary files. If that is not the case, Dymola will create working files in the current directory, which are not automatically deleted.

18 Index

4

4K
using 4K screens, 1557

6

64-bit
dymosim, DDE, FMU, 892

A

about, 526
absolute tolerance, 896
acausal modeling, 154
active simulation model, 325
adaptive
homotopy, 1419
AddModelicaPath, 952, 1564
algebraic loops, 1425
nonlinear, 1024
algorithms, 159, 802, 821, 895
absolute tolerance, 896
Adams, 896
A-stable, 898
BDF, 896
Cerk23, 898
Cerk34, 898
Cerk45, 898
Cvode, 898
DAE mode, 899
DASSL, 898
dense output, 896
Dopri45, 898
Dopri853, 898
Esdirk23a, 898
Esdirk34a, 898
Esdirk45a, 898
Euler, 899
Explicit Euler (inline integration), 1023
Explicit Runge Kutta (inline integration), 1023
fixed order, 897
global error, 896
Godess, 898
Implicit Euler (inline integration), 1023
Implicit Runge Kutta (inline integration), 1023
LSODAR, 898
Mixed explicit/implicit Euler (inline integration), 1023
order, analyze, 649
Radau IIa, 898
relative tolerance, 895
Rkfix2, 899
Rkfix3, 899
Rkfix4, 899
Rosenbrock (inline integration), 1023
Sdirk34hw, 898
sparse solvers, 900
stability boundary, 897
step size, analyze, 649
stiff, 897
Trapezoidal method (inline integration), 1023
variable order, 897
variable step size, 896
alias variables

handling of alias names in error messages and other outputs, 757
listing of eliminated alias variables, 988
align
components automatically, 246
objects, 475
to gridlines, 516
alist, 422
analogFilter, 1144
analytic Jacobians, 1427
angle for elliptical arc, 263
animate, 872
together, 872
animated GIF, 496, 866
animation, 48, 596
animation of one result file, 758
animation of several result files together, 759
backstep, 861
continuous run, 862
export, 866
exporting high resolution images, 496
pause, 861
play, 860
reverse, 861
rewind, 861
save settings in script, 759
save settings without script, 759
setup, 861
step forward, 861
time reduced, 496
visual shapes, 597
window. *See* animation window
animation tab
3D view control, 866
animation [setup], 861
export animation, 866
fit view, 867
follow, 867
new [animation window], 860
new visualize 3D [window], 860
reset all, 867
reset view, 867
rotation center, 868
show history, 867
show trace, 867
time slider, 861
transparency, 868
transparent, 867
visible, 867
animation window, 46, 596
moving view, 49, 761
pan view, 49, 761
roll view, 49, 761
rotate view, 49, 761
scroll view, 49, 761
tilt view, 49, 761
zooming, 49, 761
annotation
protection, 1207
annotations, 274, 1479
display of, 537
expanding/collapsing, 275
graphical, 476
parameter, 382
update Modelica annotations, 1537
variable, 382
vendor-specific, 1423
API, 1238
model structure, 1238
area chart, 744
array
defining an array, 69
display constant arrays as matrices, 630
of connectors, 359
array of records in dialog, 1509
arrays, 1420
arrow keys, 219
assert, 1423
assignment, 159
interactive, 910
associate commands to a model, 793
A-stable, 898
auto-format, 285, 538
automatic differentiation, 1386
automatic manhattanize, 252
AVI (Audio-Video Interleaved), 496, 866
axis
selection of vertical axis, 846, 857

B

bar chart, 744
base class, 199, 301
basic primitives, 1444, 1456
batch execution, 933
Bessel filter, 1145
bifurcation warnings, 663
binary model export, 1335, 1336
dymosim DLL, 1336
license, 1334
XML interface, 1337
bitmap images, 268

black-box import using FMI, 1272
BLT partitioning, 1426
BMP
  graphical object, 268
borrowing
  general, 1591
  on Linux, 1598
  on Windows, 1592
bound parameter, 116
boundary, 197
bounds checking, 1000
bracket handling, 286
break, 1413
bring to front, 476
browser
  component, 35, 36, 202
  information, 215, 613
  package, 35, 202, 513
  undock, 202
  variable, 588, 869, 873
built-in commands, 937
built-in functions, 937
bus signal tracing, 606
Butterworth filter, 1145

C

C compiler, 1546
calculateNumberPrecision, 955
calibrate, 1094
calibrateSteadyState, 1158
calibration, 1093, 1111
  detrending signals, 1139
  free start values, 1114
  limiting signals, 1139
  measurement file formats, 1108
  model validation against measurement, 1115
  parameter. *See* parameter tuning
  preprocess data, 1135
  sensitivity, 1133
  setting up/executing task, 1095
  static, 1146
  steady-state. *See* static calibration
  validation of nominal model, 1100
Calibration package. *See* packages : Calibration
Call Function, 544
capture parameters
  as favorite, 228
  as model, 336
cascading style sheet, 497
Cascading Style Sheets, 320
cd, 952
change
  attributes, 479
  class, 347
  current working directory, 461, 952
characters
  national, 327
chattering, 1003, 1012
Chebyshev filter, 1145
check, 478, 491
  model - checkModel (built-in function), 940
  model - normal, 491
  model - pedantic, 491
  model - with simulation, 491
  Modelica text - syntax, 288
  of built-in functions, 492
  pedantic, 290
  stop checking, 492
check box in dialog, 1492
check of input data in dialog, 1508
checkCalibrationSensitivity, 1133
checking
  of conversion scripts, 1534
checkLibrary, 1216
checkModel, 940
circle sector, 263
  closure, 264
circles, 263
class
  add base class, 346
  change, 347
  change class of component, 345
  constraining, 393
  coverage, 1223
  description, 154, 303, 309, 479
  documentation, 95, 300
  extend, 111, 232
  lock, 335
  moving, 547
  parameters. *See* class - replaceable classes (local)
  renaming, 547
  replaceable classes (local), 158, 346, 385
  set component scale factor, 481
  unlock, 335
classDirectory, 952
classes
  creating, 233
cleanup
  cleanup a model developed step-by-step, 376
clear all, 458

clear log, 467
clearPlot, 955
clipboard, 495
closure of elliptic arc/circle sector, 264
code
  export. *See* code and model export
code and model export
  disabling export, 1335
  license options, 1334, 1335
code completion, 283
  SI units, 283
color, 271, 471, 473, 1449, 1452
  coding, 282
  defining custom colors, 472
  of plotted variables, 844
  palette, 472
  selecting, 471, 472
  true RGB, 473
color-coding
  color-coding of result file tables, 742
combo box in dialog, 1490
Comma Separated Values. *See* file extensions : .csv
command
  command line arguments for Dymosim, 904
  history, 780
  history window, 601
  input line, 599, 601, 768
  log, 599, 600, 766, 812
  log file, 600, 767
  log pane, 599, 600, 766
  window, 599, 812
command window. *See* command - window
commands
  built-in, 937
  convertClass, 1518
  convertClear, 1524
  convertElement, 1520
  convertModifiers, 1521
  post-processing, 787
comment
  attribute. *See* class description
comment out, 287
comment selection, 538, 889
comments, 278, 296
compareModels, 1229
compiler, 1546
  cross-compilation for Linux on Windows, 1546
  GCC (on Linux), 1581
  GCC (on Windows), 1546
  installation, 1546
  Microsoft Visual C++, 1546
  on Linux, 818
  setup, 817
  troubleshooting, 1613
component, 22
  browser, 35, 36, 202
  conditional, 363
  display subcomponents, 220
  double-clicking in the diagram layer, 517
  inner/outer, 1422
  library, 83
  path, 220
  reference, 119, 296
  renaming, 235
  replaceable, 347
  Shift+double-clicking in the diagram layer, 518
  Show Component, 220
  size, 198, 235, 480
  space and align automatically, 246
computer algebra, 156
conceal. *See* encryption
Concurrent Version System. *See* model management : version control
concurrent zooming. *See* plot window zooming
condition coverage, 1224
conditional
  components, 363
  connectors, 363
connection, 151, 219, 470, 490
  creating using Smart Connect, 248
  creating without using Smart Connect, 252
  displaying connector names in the diagram, 255
  hiding graphical connection, 260
  highlighting connections when clicking them, 254
  insert point, 554
  remove point, 554
connections, 247
connector, 151
  array, 359
  conditional, 363
  displaying connector names in the diagram, 255
  expandable, 355
  nested, 352
  overlapping, 353
  protected, 192
  stacked, 353
  stream, 156
  visibility, 192
connectors
  activate warnings for unconnected physical and input connectors, 998
constraining
  class, 393

clause, 393
context menu, 36, 219, 236
animation window, 880
command window - command input line, 883
command window - command log pane, 882
component browser, 543
components, 551
connection, 253
connection while connecting, 253, 555
connections, 554
connectors, 555
coordinate system boundary, 553
edit window Diagram layer, 530
edit window Documentation layer, 531
edit window Icon layer, 530
edit window Modelica Text layer, 533
edit window Used Classes layer, 541
graphical objects, 555
log window, 890
plot window, 874
plot window – curve and legend, 876
plot window - signal operators, 877
plot window - text objects, 878
script editor, 885
table window, 879
table window - value, 879
variable browser - signals, 873
variable browser nodes, 869
visualizer window, 882
continue line, 910
continue simulation, 827
continuous run of animation, 862
contour lines, 1468
ControlDesk, 1318
conversion
multiple conversion of libraries, 1536
offset, 425
conversion script, 1524
conversion scripts
checking of, 1534
coordinate system, 197, 480
boundary, 197
creatinga reference coordinate system, 1458
specification, 198
coordinateSystem, 1467
copy, 469, 487, 490
Copy Call, 545
copy to clipboard, 495, 701
corner radius, 263
Co-simulation
FMI for Co-simulation, 1268
CPU time, 806
create
link, 312, 776
plot window using built-in function, 955
table window using abuilt-in function, 959
createPlot, 955
createTable, 959
creating a model, 85
critically damped filter, 1145
crossing function, 162
CSS, 320, 497
current selection, 218
cursor
measurement cursor for plots, 715
position - display, 517
Custom Parameters group, 386
cut, 469, 487, 490
CVS, 1176, *See* model management : version control
CVS commands, 1171
CVS selection, 1175

D

DAE, 155, 1425
high index, 156, 1427
DAE mode, 802, 899
daemon
license, 1585
vendor, 1585
dark mode, 1558
data
use of data files, 420
data flow modeling, 155
data preprocessing for calibration, 1135
DataFiles package, 421
dataPreprocessing, 1135
DC motor, 72
DDE
Dymola commands, 1328
Dymosim DDE server, 1331
extended GUI, 1333
GUI, 1333
hot linking variables, 1333
requesting variables, 1332
setting parameters, 1332
simulator commands, 1331
DDE interface for Dymola, 1328
DDE server, 819, 892
debug monitor, 1013

debugging models, 812, 974
declaration
  conditional, 360
  derivative, 1387
  parameter, 376
  variable, 376, 484, 494
declarations, 55
default graphics, 270
delay-load, 1396
delete, 470, 487, 490
delete key, 220
deleting
  variable, parameter, or constant, 383
deleting objects, 220
demo, 34, 454
dense output, 896, 898
dependencies
  model, 1193
  plot, 633
deprecation
  warnings, 1424
der(...) operator, 52
derivative, 1385
derivatives
  partial, 1386
description of class, 154, 303, 309, 479
Design package. *See* packages : Design
Design Structure Matrix, 506
Design.Calibration, 1093
Design.Experimentation, 1043
detrending signals, 1139
DFT. *See* discrete Fourier transform
diagram layer, 49, 192, 469
  filtering, 222
  moving view, 217, 681
  working with diagram layer in the Simulation tab, 680
  zoom factor, 469
  zooming, 217, 682
diagram layer when simulating, 620
dialog
  array of records, 1509
  automatically constructed, 1480
  check box, 1492
  check of input data, 1508
  combo box, 1490
  edit buttons in dialog, 1496
  entry for start values, 1487
  formatting, 1495
  groups, 1483
  illustrations, 1493
  labels, 1488
  layout, 1488
  open files, 1496
  radio buttons, 1491
  select model, 1496
  tabs, 1483
differential-algebraic equations, 1425, *See* DAE
  high index, 1427
differentiation
  automatic, 1386
directory
  change current working directory, 461, 952
  current working directory, 461
  default working directory, 461, 952
  startup directory, 461, 523
discrete equations, 161
discrete Fourier transform. *See* Fourier transform
discretized plot, 631
display changed values in variable browser, 672
display unit
  select display unit for x-axis in plot when other variable than time, 714
  select display units in plot, 713
  select display units in variable browser, 624
  selecting display units in parameter dialog, 241
  setting imperial units as default for plots, 426
displayUnit, 241, 423, 438, 713
  prefixed, 242, 425
  reciprocal, 425
displayunit.mos, 423, 714
displayunit_us.mos, 423, 714
DLL
  dymosim, 1336
document, 938
documentation
  editor. *See* editor - documentation
  for web publication, 324
  generating HTML documentation, 320
  HTML (of classes), 300
  manuals, 137
  markdown, 322
  metadata, 320
  of class, 95, 300
  of model, 95, 300
  of simulation, 122, 771
  printable, 324
  templates, 324
documentation layer, 193, 300, 485
  content, 301
  edit, 309
  HTML source code, 304

toolbar, 302
documentation tab
copy, 487
cut, 487
delete, 487
duplicate, 487
find and replace, 487
formatted [documentation], 485
go to, 487
html, 486
info editor, 485
info source, 486
markdown, 486
meta data, 486
paste, 487
redo, 486
revision editor, 486
revision source, 486
select all, 487
undo, 486
double-clicking
a component in the diagram layer, 517
while drawing a connection, 252
dragging, 235
dres.mat. *See* Dymosim files : dres.mat
dsfinal.txt. *See* Dymosim files : dsfinal.txt
dsin.txt. *See* Dymosim files : dsin.txt
dslog.txt. *See* Dymosim files : dslog.txt
DSLS
on Linux, 1598
on Windows, 1596
dsmodel.c, 987, 1334
interfacing, 1340
dSPACE
ControlDesk, 1318
DS1005, 1316
DS1006, 1316
dym_rti_build, 1319
overrun situation, 1318
turnaround time, 1318
dsu.txt. *See* Dymosim files : dsu.txt
dummy derivative method, 1428
duplicate
class, 546
objects, 470, 487, 490
DXF files, 758
Dymola
32-bit application on Linux, 1581
32-bit application on Windows, 1542
64-bit application on Windows, 1542
latest Dymola release, 526
latest release available, 1579
Dymola license server
on Linux, 1597
on Windows, 1585
Dymola Main window, 187
DymolaBlock, 1303
Dymosim, 892
command line arguments, 904
DDE server, 892, 1330
dymosim DLL, 1336
API, 1337
Dymosim files, 892
dsfinal.txt, 943
dsin.txt, 893, 904, 943
dslog.txt, 893
dsres.mat, 893
dsu.txt, 893
dymtools, 894
dynamic state selection. *See* state selection : dynamic, *See* state selection
dynamic typing, 382

E

each, 578
echo
script commands, 932
echo script commands, 838
edit
angle for elliptical arc, 263
documentation (of class), 96, 309
documentation (of simulation), 122, 771
documentation layer, 309
equation or expression (in command log), 779
function call, 296
graphical annotations, 476
graphical objects, 261
HTML source code, 318
Modelica text layer, 271
text object, 265
edit annotation
transformation extent, 476
edit buttons in specialized GUI widget, 1496
Edit combined, 568
edit window, 35, 189, 218
editor
documentation (of class), 96, 309
documentation (of simulation), 122, 771
Modelica text, 271
efficiency

improving simulation, 1005
eFMI, 1293
eFMU, 1293
ellipses, 263
elliptic arc, 263
closure, 264
encapsulated, 479
encrypted file, 1196
Encrypted total model, 1213
encryption, 1195
Modelica files, 1197
Enhanced Metafile Format, 495
enumerations, 401
enumeratons, 1421
environment variables
CLASSPATH, 1350
CVSROOT, 1178
DYMOLA, 1582
DYMOLA_RUNTIME_LICENSE, 1335, 1555
DYMOLAPATH, 1582
DYMOLAWORK, 463, 1557
DYMOSIMGUI, 1333
DYMOSIMLOGDDE, 1334
DYMOSIMREALTIME, 1331
EDITOR, 1185
JAVA_HOME, 1396
LM_BORROW, 1598
MODELICAPATH, 1562, 1582
PATH, 1177, 1186, 1350, 1366
PYTHONPATH, 1366
SVN_SETUP, 1185
VISUAL, 1185
equality comparison, 1411
equation incidence, 640
equations
manipulated, 987
synchronous, 160
Erase Selected Objects, 1454
Erase window, 1449
eraseClasses, 953
EraseClasses, 1246
erasing graphical object, 1446, 1453
error messages, 603, 976
report function call in, 813
errors, 974
rules for avoiding, 974
evaluation
of parameters, 784
evaluation of parameters, 173
event, 162
events and chattering, 1012
iterations, 656, 806
iterations - (advanced) control of, 782
list possible event-triggering expressions, 985
logging, 985
logging interface. *See* event logging interface
noEvent operator, 1405
resolution - (advanced) control of, 782
event logging interface, 654
bifurcation warnings, 663
plot values and iterations, 660
show component in diagram, 661
show variable in code, 661
examples
add modifiers, 390
automatic differentiation, 1386
bounds checking, 1000
calibration - setting up/executing task, 1095
check if tuners can be calibrated, 1133
combining basic primitives, 1446
CVS file management, 1178
DC motor, 72
encrypted transfer function, 1198
experimentation case study - CoupledClutches, 1066
filtering signals, 1144
Furuta pendulum, 126
ideal diode, 164
initialization of discrete controllers, 175
limiting and detrending signals, 1139
model template - test-driven, 372
model template using replaceable components, 369
model validation, 1100, 1115
Monte Carlo Analysis, 1082
motor drive, 150, 155
motor model, 151
noise analysis, 1141
pendulum, 52, 171
perturb parameters (calibration), 1131
perturb parameters (experimentation), 1067
plot - createPlot, 958
plot - plot (built-in function), 961
plot - plotArray, 963
plot - plotText, 969
polynomial multiplication, 159
preprocess data for calibration, 1135
providing/concealing information, 1203
rectifier circuit, 165
result files in variable browser - keep between simulations, 677
results of translating Modelica models, 988
sampled data system, 160
scrambling, 1213
start values, 1026

stuck simulation, 1003
SVN file management, 1186
sweep one parameter (calibration), 1121
sweep one parameter (experimentation) - two variants, 1073
sweep two parameters (calibration), 1129
sweep two parameters (experimentation), 1080
exit
Dymola, 467
expandable connectors, 355
signal tracing, 606
expanding/collapsing
annotations, 275
components, 274
connections, 274
experiment (built-in function), 941
experiment setup. *See* simulation setup
experiment.StopTime, 1216
Experimentation package. *See* packages : Experimentation
export. *See* code and model export
animation, 496, 866
DSM (Design Structure Matrix), 506
graph, 503
HTML, 497
image, 495
model to other simulation environments, 796
plot as image, 959
settings, 525
SysML, 506
to clipboard, 495
UML, 504
export options. *See* code and model export - license options
need for, 904
exportAnimation, 972
exportDiagram, 973
exportDocumentation, 973
exportEquations, 973
exportIcon, 973
exporting models using FMI, 1251
exporting models with built-in numerical solvers, 1268
exportInitial, 941
exportSSP, 941
expression
interactive, 909
expressions
simplifying if-then-else expressions, 1009
extend, 111, 232
component with parameter values and other modifiers included, 233, 336
Extendable user interface
menus, toolbars and favorites, 1510
external function, 1391
annotations, 1393
including, 1391
link with library, 1392
external libraries
recommended location, 1395

F

fast sampling, 1003
fault-finding, 974, 1005, 1013
favorites packages, 227
file
import (alternatives), 420
file data
use of, 420
file extensions
.avi, 972
.AVI, 496, 866
.bmp, 268
.cab, 1559
.csv, 420, 562, 871, 1108, 1138
.emf, 495
.gif, 268, 866
.jpeg, 268
.mat, 420, 562, 871, 1108, 1138
.mo, 450, 456, 1330
.moe, 1197
.mof, 987
.mos, 714, 918, 1330
.msi, 1559
.png, 268, 495
.sdf, 420, 562, 790, 871
.svg, 495
.txt, 420, 562, 871
.wrl, 496, 867, 972
.x3d, 972
mos, 793
file formats
.x3d, 496, 867
.xhtml, 496, 867
file menu
clear all, 458
clear log, 467
demos, 454
exit, 467
export SSP, 458

import FMU, 453
import SSP, 453
libraries, 453
new, 447
new > duplicate class, 451
new > package, 450
new > script, 451
new > type (enumeration), 450
open, 452, 453
open > load, 452
open script, 453
options, 467
print, 464, 465
print preview, 465
recent files, 467
save, 455
save all, 457
save as, 457
save Encrypted File, 458
save Encrypted Total Model, 458
save log, 466
save total, 457
search, 458
working directory, 461
working directory - browse [for], 463
working directory - open in command prompt, 464
working directory - open in file browser, 464
working directory - use as startup directory, 464
file type associations, 1330
fill
color, 271
gradient, 473
pattern, 473
style, 473
filter
Bessel, 1145
Butterworth, 1145
Chebyshev, 1145
component browser content, 212
critical damped, 1145
diagram layer, 222
inherited components in the diagram layer, 226
package browser content, 208
variables in variable browser, 667
filtering signals, 1144
final, 578
find
bus signals (expandable connectors), 606
class or component, 458
components in diagram layer, 482
connctions (and connected components), 257
connections (and connected components), 483
text (inside a class), 534
usage of class, 458
Find and Replace, 534
fixed attribute of start value, 71, 168, 243
fixed order, 897
flags
displaying information for, 938
GUI for displaying and setting Boolean, integer, real, and string flags, 431
flat Modelica text, 196
FLEXnet, 1585
license server options file, 1595
flip
horizontal, 474
vertical, 474
flow prefix, 152
FMI, 1250
exporting models, 1251
FMI Kit for Simulink, 1292
for Co-simulation, 1268
importing black-box models, 1272
importing models, 1271
model exchange, 1250
specification for Co-simulation, 1250
specification for model exchange, 1250
validating FMUs, 1289
XML model description, 1250
FMI (Functional Mock-up Interface)
see FMU, 951
FMU, 1250
black-box export, 1257
export for multiple platforms, 1270
export model to FMU, 797, 951
exporting a model in DAE mode as FMU, 1263
exporting FMU's, 1251
exporting FMUs with settings, 1264
exporting FMUs with source code using sparse Jacobians, 1263
filtering signals when importing an FMU, 1277
FMU export from Simulink, 1292
generate Dymola result file in .mat format, 1251
import, 453
import FMU, 943
importing FMU's, 1271
importing FMUs with many inputs/outputs, 1283
including 'save total' of the model in the generated FMU, 1264
Linux cross-compiler, 1270
model identifier, 1263
multiple FMU´s, 1267
multiple instantiation of the same FMU, 1267
online tunable parameters, 1251

profiling, 1292
refresh FMU, 1288
re-import, 1288
specifying the max simulation run time per simulation, 1263
string parameters, 1264
unit handling, 1287
validating FMUs, 1289
FMU export for multiple platforms, 1577
font
for documentation, 310
size, 56, 286
font size, 267, 311, 776
base, 511
for documentation, 311
restrict minimum font size, 516
formatting
Modelica text, 285
formatting of dialog, 1495
Fourier transform, 1142
frame rate, 496
freeStartValues, 1114
frequency
variable frequency of output points, 783
frequency analysis, 1141
function, 159
derivative, 1385
derivative - smoothOrder, 1387
external, 1391
preInstantiate, 1424
function call
edit, 296
insert, 294, 539, 570, 578, 769, 884, 889, 906
Functional Mock-up Unit. *See* FMU
functions
built-in, 937
calibrate, 1094
calibrateSteadyState, 1158
checkCalibrationSensitivity, 1133
compiled, 916
dataPreprocessing, 1135
functional input argument to, 1423
perturbParameters (calibration), 1131
perturbParameters (experimentation), 1067
pre-compiled, 917
staticCalibrate, 1146
sweepOneParameter (experimentation), 1078
sweepParameter (calibration), 1121
sweepParameter (experimentation), 1074
sweepTwoParameters (calibration), 1129
sweepTwoParameters (experimentation), 1080
Furuta pendulum, 126

G

GCC (on Linux), 1581
GCC (on Windows), 1546
generate
script, 833
GenerateBlockTimers, 1018
GenerateTimers, 1020
getClassText, 973
GetDymolaCompiler, 941
getExperiment, 943
getLastError, 953
GIF
animated, 496, 866
graphical object, 268
Git, 1190, *See* model management : version control
Git commands, 1171
Git selection, 1175
global ambient light, 1446
global error, 896
global settings
GUI for displaying and setting global flags, 431
Godess solvers, 898
graphic tab
insert, 484
graphical annotations, 476
graphical attributes, 271
graphical objects, 261, 596
default, 270
insert point, 556
remove point, 556
graphical properties, 1449
graphics. *See also* plot *and* Plot3D
basic primitives, 1444, 1456
color, 1449, 1452
contour lines, 1468
creating a reference coordinate system, 1458
erasing graphical object, 1446, 1453
inserting graphical object, 1446
pie chart, 1474
Plot3D. *See* Plot3D
rotation, 1450
scaling, 1450
surface plots, 1458
translation, 1450
Visualize 3D, 1443
graphics tab
align, 475

annotation, 476
Attributes, 479
back, 468
bitmap, 471
check, 477, 478, 491
connect mode, 482
connector names, 478
copy, 469
create local state, 484
cut, 469
delete, 470
diagram, 469
diagram filter, 484
draw group, 261
duplicate, 470
ellipse, 470
fill style, 473
find - components in diagram layer, 482
find connection, 483
fit window, 469
flip horizontal, 474
flip vertical, 474
forward, 468
icon, 469
line, 470
line style, 471
manhattanize, 474
order, 476
paste, 470
polygon, 470
recent, 468
rectangle, 470
redo, 469
rotate, 474
route connections, 474
select all, 470
smooth, 474
split model, 477
text, 471
toggle grid, 478
transition mode, 482
undo, 469
variable selections, 484
zooming, 469
grid, 197, 199, 219, 252, 262, 480
display, 478
gridlines, 199, 219, 478
grouping of record fields in dialog, 1483
guess values
for nonlinear equations during simulation, 997
GUI
user-defined, favorite models, 1513
user-defined, for data structures, 1479
user-defined, library-specific menus and toolbars, 1513
user-defined, menus and toolbars, 1510
GUI widgets
in general. *See* dialog
specialized, 1496
guidelines
for model development, 974

H

handles, 218
hardware-in-the-loop simulation, 26, 1313
help, 938
hiding variables, 1486
high index DAE, 156
high oscillatory modes, 1145
high resolution images, 495
highlight syntax, 55, 538
Highlight syntax, 282, 288
HIL, 990, 994, *See* hardware-in-the-loop simulation
history
command history, 780
homotopy, 1419
adaptive, 1419
debuggnig, 1419
operator, 1419
host id, 1554
HTML, 497
documentation of classes, 300
export, 497
external references, 503
generating HTML documentation, 320
hyperlinks, 300, 312, 776
Microsoft Word template, 499
online documentation, 498, 503
options, 497
setup, 497
hybrid model, 159
hybrid modeling, 159
hybrid system, 985
hyperlink, 312, 776

I

icon layer, 191, 469
moving view, 217
zooming, 217
identifiability. *See* parameter : identifiability
if-expression, 163

simplifying if-then-else expressions, 1009
if-then-else expression, 1439
illustrations in dialog, 1493
images
bitmap, 268
high resolution, 495
inserting images in parameter dialog, 393
inserting in command log, 777
inserting in diagram or icon layer, 268
inserting in documentation layer, 314
imperial units. *See* displayUnit_us.mos
setting imperial units as default for plots, 426
import
file data (alternatives), 420
FMU - importFMU (built-in function), 943
initial, 827
settings, 525
start values - importInitial (built-in function), 943
start values - importInitialResult (built-in function), 943
statement, 910
import FMU, 453
importFMU, 943
importing models using FMI, 1271
black-box models, 1272
importInitial, 943
importInitialResult, 943
importSSP, 944
importSSV, 944
incidence matrix, 640
incorrect license file format, 1556
indenting, 285
independent variable in plot, 629
index reduction, 1427
Info
button, 306
menu entry, 306
info button, 41
information
browser, 215, 613
initialization
steady state, 1037
initialization of models, 167, 1037
over-specified, 1024
initialized, 944
inline integration, 821, 1021, 1314
advanced options, 1324
algorithms, 1023, *See* algorithms
input
setting inputs from file, 422
insert
equation or expression (in command log), 777
equation or expression (in documentation layer), 315
function call, 294, 769, 884, 906
graphical object, 1446
image in command log, 777
image in diagram or icon layer, 268
image in documentation layer, 314
link, 312, 776
Modelica text into MS PowerPoint, 325
statement, 294
table in documentation, 316
Instability, 986
installation
environment variables, 1582
license daemon, 1585
license server on Linux, 1597
license server on Windows, 1585
Linux, 1581
remote on Windows, 1559
troubleshooting, 1611
windows, 1542
instance
hierarchy, 220
integer
short integer, 565
integration algorithms. *See* algorithms
integration methods. *See* algorithms
interface variables, 405
intergrator
order, analyze, 649
step size, analyze, 649
interpretMainStatic, 1400

J

Jacobian, 168, 171, 608, 987, 1426
using analytic ODE Jacobian, 1010
Java interface, 1350
calling Java functions from Modelica, 1396
calling Modelica functions from Java functions, 1398
interpretMainStatic, 1400
Java interface (older version), 1396
JavaScript interface, 1375
JPEG
graphical object, 268

K

keyword
redeclare, 158

L

labels in dialog, 1488
language setting, 1556
latest Dymola release, 526
layer, 189
  diagram, 192
  diagram when simulating, 620
  documentation, 193
  icon, 191
  Modelica text, 194
  used classes, 196
layout of dialog, 1488
least squares, 1140
library, 83
  annotation, 1393
  checking, 1215
  creating, 227
  customizing the File > Libraries menu, 1561
  documentation, 300
  libraries used, 904
  licensing, 1210
  list of libraries, 179
  management, 507
  managing libraries location, 1562
  menu, 453
  migration, 1517
  Modelica libraries newer than the ones in Dymola distribution, 1560
  Modelica libraries older than the one in Dymola distribution, 1560
  multiple conversion, 1536
  on-demand installation of downloaded libraries, 1565
  shortcut to, 453
  window, 213
  working with pre-release, 1534
libraryinfo.mos, 1567
license
  binary model export, 1334
  borrowing - general, 1591
  borrowing - on Linux, 1598
  borrowing - on Windows, 1592
  daemon, 1585
  demands, 1606
  early return - on Linux, 1598
  early return – on Windows, 1594
  error message, 1611
  expiration - settings, 1572
  export options, need for, 904
  incorrect file format, 1556
  node-locked, 1554
  not found or invalid - settings, 1572
  preventing checking out - in Linux, 1584
  preventing checking out – in Windows, 1572
  real-time simulation, 1334
  requirements, 1606
  runtime concept, 1556
  runtime, need for, 904
  server
    on Linux, 1597
    on Windows, 1585
  server options file, 1595
  sharable, 1552
  source code generation, 1334
licensing
  of library, 1210
lighting
  ambient lighting, 866
  extended lighting, 866
limiting signals, 1139
line, 262
  arrow style, 473
  color, 271
  style, 471
  style of plotted variables, 845
  thickness, 473
  thickness of plotted variables, 845
line (y = a*x+b), 1140
linear analysis menu
  bode plot, 527
  full linear analysis, 527
  linearize, 527
  poles, 527
  poles and zeros, 527
  root locus, 527
Linear Systems library, 1145
linearize, 827, 945
  Co-simulation FMUs, 1285
linearizeModel, 945
link
  create, 312, 776
  in error log to variables in model window, 984
  insert, 312, 776
  local, 313, 777
Linux
  compiler, 818
  cross-compilation for Linux on Windows, 1546
  Dymola as 32-bit application, 1581
  Dymola license server, 1597
  installation, 1581
Lissajous curve, 1458
list, 945
listfunctions, 945

load (data files). *See* import
local
  link, 313, 777
local class reference, 296
local host id, 1554
lock classes, 335
log
  command, 599, 600, 766, 812
  command log file, 767
  simulation, 608
  translation log, 604
log window, 603
logging events, 654, 985

M

main window
  model layer toolbar, 529
  toggle button for tab information level, 529, 869
  windows menu, 528
manhattanize, 92, 252, 474
manipulated equations, 987
markdown, 322
marker style
  of plotted variables, 845
matching
  of variables, 404
mathematical notation
  command log, 772
  Modelica text layer, 278
Matlab, 24, 893, 1302, *See also* Simulink
  Dymosim m files, 902
  importing Dymola result files to Matlab, 894
  importing Matlab files to Dymola, 563
  loading Matlab files to Dymola, 420
  mex, 1302
  m-file utilities, 1302
  path, 1302
  saving result in Matlab format, 420, 562, 871
matrix
  defining a matrix, 69
  editor, 561
  equations, 157
  handling of a large matrix, 569
  incidence matrix, 640
matrix dependencies
  handling of large matrix dependencies for FMU export, 1263
matrix T, 1449, 1450
max run time, 800
mean value, 1141
measurement cursor, 715
media
  media propagation, 366
mesh function, 1458
message window. *See* log window
metadata, 320
methods. *See* algorithms
m-file, 902
  dymbrowse, 894, 903
  dymget, 894, 903
  dymload, 894, 903
  load2DTable, 1313
  loaddsin, 1313
  loadNDTable, 1313
  save2DTable, 1313
  saveNDTable, 1313
  setfromdsin, 1313
  setParametersFDsin, 1313
  tcomp, 903
  tcut, 903
  tder, 903
  tdiff, 903
  tfigure, 903
  tinteg, 903
  tload, 894, 903
  tloadlin, 903
  tnhead, 903
  tnindex, 903
  tnlist, 894, 903
  tplot, 894, 903
  tplotm, 903
  trajectory, 902
  trange, 903
  tsame, 903
  tsave, 895, 903
  tzoom, 903
Microsoft
  Excel, 871
  PowerPoint, 325
  Visual C++, 1546
  Windows installation, 1542
  Word 2003, 495
  Word 97, 495
  Word HTML template, 499
migration to a new library, 1517
mixed-mode integration, 1021
mode
  pedantic compiler mode, 290, 806
  simulation, 18
model, 149
  active simulation model, 325

associate commands, 793
calibration. *See* calibration
cleaning up a model developed step-by-step, 376
comparison, 1229
creating, 232
debugging, 974
dependencies, 1193
documentation, 95, 300
export. *See* code and model export
extend, 232
hybrid, 159
instability, 986
management. *See* model management
metadata, 320
packages. *See* packages
partial, 153
replaceable, 158
split model, 338
structure, 1238
submodel, 338
templates, 369
test-driven templates, 372
model description
XML model description for FMI, 1250
model editor, 149
model exchange using FMI, 1250
model identifier. *See* FMU : model identifier
model layer
in simulation, 590
model layer toolbar, 529
model management, 1215
check package, 1215
encryption, 1195
model structure, 1238
version control, 521
model tabs, 200
model view window, 49, 590, 620, 679
Modelica, 25, 149, 272
modelica URI scheme, 269, 314
Synchronous, 1418
Modelica layer, 489
Modelica path, 1562
save changes of, 523
Modelica Reference, 179
Modelica Standard Library, 72, 73, 151, 178, 520
newer than the one in Dymola distribution, 1560
older than the one in Dymola distribution, 1560
Modelica text, 194, 272, 273
editor, 272
flat. *See* flat Modelica text
Modelica text layer, 272, 273
bracket handling, 286
code completion, 283
color coding, 282
comment out, 287
comments, 296
component reference, 296
context menu, 282
features, 281
font size, 56, 286
formatting, 285
function calls, 294
indenting, automatic, 285
inserting statements, 294
local class reference, 296
mathematical notation, 278
navigation, 278, 287
syntax check, 288
tooltip for short description of functions, 284
variable declaration, 291
MODELICAPATH, 1562
modeling, 149, 187
acasual, 154
hybrid, 159
MODELISAR, 1250
ModelManagement.Structure.AST, 1239
ModelManagement.Structure.Instantiated, 1246
modifiers, 238, 244
add modifiers, 390
Monte Carlo analysis, 1043, 1082
accumulated probability, 1087
automatic selection of set of bins, 1085
density of probability, 1087
fixed parameters, 1085
number of draws, 1086
observed variables, 1085
random distributions available, 1088
uncertain parameters, 1083
MonteCarloAnalysisScatter, 1088
mouse button
left, 218, 219
right, 219
moving
animation view, 49, 761
class, 547
diagram layer view, 217, 681
icon layer view, 217
objects, 219
plot view, 703
plot window, 102
visualizer view, 1448
multi-core support, 1006

N

national characters, 327
negated curve, 700
nested connectors, 352
new
  enumeration type, 450
  model, connector etc., 448
  package, 450
  script, 451
new script, 832
New variable, 484, 494
node-locked license, 1554
noEvent(...) operator, 163, 1405
noise analysis, 1141
nonlinear, 1024
nonlinear equations, 1433
  guess values during simulation, 997
non-linear systems, 1001
nowindow, 934
numeric integration
  analyze, 649

O

ODE, 156, 1427
open all, 870
open script, 453, 833
opening of files in specialized GUI widget, 1496
openModel, 946
operator overloading, 1418
order, 897
ordering of objects, 476
ordinary differential equations, 1427, *See* ODE
output interval, 801
  variable output interval, 783
overlapping connectors, 353
overloading, 1418
over-specified initialization problem, 1024

P

package
  browser, 35, 202, 513
  creating, 227
  favorites, 227
  hierarchy, 35
  local, 277
  Plot3D. *See* Plot3D
  storing of, 450
packages
  Calibration, 1043, 1093
  Design, 1043
  Experimentation, 1043
pan
  animation view, 49, 761
  visualizer window, 1448
Pantelides´ algorithm, 1428
parallel code support
  general, 1006
  general tips, 1008
  handling of external functions, 1008
  parallel Jacobian update, 1008
parameter, 172, 237
  alias elimination, 1432
  annotations, 382
  calibration. *See* parameter tuning
  class parameters. *See* class - repaceable classes (local)
  declaration, 154, 376
  dependency, 1121
  dialog, 38, 238
  error, 376
  estimation. *See* parameter tuning
  evaluation, 173, 784
  evaluation - logging of, 785
  finding unused parameters, 492
  identifiability, 1133
  MonteCarloAnalysis. *See* Monte Carlo analysis
  perturb (calibration), 1131
  perturb (experimentation), 1067
  propagation, 116
  record, saving of, 395
  record, searching, 397
  sensitivity, 1094, 1121
  string parameters, 393
  structural, 784
  sweep, 798
  sweep (calibration), 1121
  sweep (experimentation), 1074
  sweepOneParameter (experimentation), 1078
  sweepTwoParameters (calibration), 1129
  sweepTwoParameters (experimentation), 1080
  tuning. *See* parameter tuning
  type prefix. *See* type prefix
  values, 172
  varying of, 1043
  warning, 376
parameter dialog
  activation of dialog entry for start values, 386
  advanced use, 385
  editing - advanced, 393

possible modifiers, 385
presentation of enumeration values, 386
parameter tuning, 1111, *See also* calibration
direct calculation, 1157
tuners, 1093, 1114, 1115
parameters
tunable - declaring tunable parameters as fixed, 1258
parametric curve, 742
parametric surface, 1458
parametric surface matrices, 1458
partial derivatives, 1386
partial models, 153
paste, 470, 487, 490
special, 325
path
component, 220
Path system variable, 1177
pedantic check, 290
pedantic compiler mode, 290, 806
pendulum, 52, 171
perturbParameters
calibration, 1131
experimentation, 1067
picking objects, 218
pie chart, 1474
platforms
non-Windows, 1340
plot. *See also* graphics
(built-in function), 959
add a pointer, 1473
an overview of functionality, 690
area chart, 744
bar chart, 744
Boolean signals, 699
create plot window using built-in function, 955
default plot style, 1466
dependencies, 633
dependencies - after failed initialization, 1027
discretization, 631
displaying a table instead of curves, 740
displaying curves negated, 700
displaying signal operators, 722
enumeration signals, 699
export as image, 959
external files, 746
from diagram layer, 687
from large result files, 691
HTML, 746
insert current plot commands in script, 930
intersection of surfaces, 1470
matlabplot, 746
menu. *See* plot menu
parametric curves, 742
plot expressions, 736
plot window interaction, 692
Plot3D. *See* Plot3D
plotly, 746
PNG, 746
predefined plots, 692
projected on the xy plane, 1469
Python scripts, 746
save plot as image, 691
save settings between sessions, 691
save settings in script, 692
save settings without script, 692
scripting, 692
select independent variable, 629
selecting multiple curves, 739
setup menu. *See* plot setup menu
surface, 1458
SVG, 746
window, 46, *See* plot window
window options, 101
plot menu
setup. *See* plot setup menu
plot setup menu, 694
plot tab
[draw] text, 856
[line] color, 857
close all, 840
delete [diagram], 841
delete [diagram] column, 858
delete [diagram] row, 858
display component, 858
display unit, 858
duplicate diagram, 841
erase content, 841
generate discretized plot, 841
generate plot in command window, 841
generate table window from plot, 841
independent variable, 842
insert [diagram] column to the left, 858
insert [diagram] column to the right, 858
insert [diagram] row above, 858
insert [diagram] row below, 858
left/right vertical axis selection, 857
line style, 857
line thickness, 857
marker style, 857
measurement cursor, 841
merge [diagram] below, 859
merge [diagram] right, 859
move [diagram] down, 859
move [diagram] left, 859

move [diagram] right, 859
move [diagram] up, 859
negated curve, 858
new diagram, 840
new table [window], 840
pause, 842
play, 842
plot expression, 856
rescale [diagram], 841
rescale all, 840
reset [window] layout, 840
reset layout, 859
signal operator, 856
split [diagram] horizontally, 859
split [diagram] vertically, 859
sync pan/zoom, 841
time slider, 842
toggle grid, 841
plot window, 591
2D layout, 695
changing displayed unit of signals, 712
changing time unit of x-axis, 711
color-coding of result file tables, 742
create tables from curves, 742
diagrams, several in 2D layout, 695
diagrams, several in one column, 694
displaying a table instead of curves, 740
displaying signal operators, 722
drag and drop of curves between plots and tables, 719
drag and drop of curves between plots/diagrams, 718
duplicate diagram to new plot window, 700
insert plot and corresponding command in command window, 753
inserting texts, 751
interaction, 692
locking, 719
measurement cursor, 715
moving view, 703
multiple y-axes, 720
parametric curves, 742
plot expressions, 736
rescale. *See* plot window zooming
selecting multiple curves, 739
time sliding window, 711
zooming. *See* plot window zooming
plot window zooming
horizontal (time) zooming, 707
rescale a diagram, 710
rescale a signal. *See* plot window zooming
rescale all diagrams in a plot window, 710
using the Range tab to zoom, 709
vertical zooming, 707
zooming in by dragging, 704
zooming in/out using Ctrl, 704
zooming out using context menus, 710
zooming with maximum y scaling, 705
Plot3D, 1443
insertLabel, 1444
insertPointer, 1444, 1473
insertPrimitive, 1444
plotBarGraph, 1444, 1468
plotData, 1465
plotHistogram, 1444
plotLines, 1444, 1468
plotPieChart, 1444
plotPoints, 1444, 1468
plotStem, 1444, 1468, 1473
plotSurface, 1444, 1468
plotTriangularizedSurface, 1444
styleData. *See* styleData
Plot3D.Examples.Surfaces.surfaceDemo, 1463
plotArray, 962
plotArrays, 963
plotDiscretized, 963
plotDocumentation, 963
plotExpression, 964
plotExternal, 964
plotHeading, 964
plotMovingAverage, 965
plotParametricCurve, 965
plotParmetricCurves, 965
plotRowColumnLabels, 965
plotScatter, 965
plotSignalDifference, 966
plotSignalOperator, 967
plotSignalOperatorHarmonic, 968
plotText, 969
plotTitle, 970
PNG
graphical object, 268
polygon, 262
polynomial multiplication, 159
post-processing
commands, 787
pre(...) operator, 161
PrecisonTiming, 1017
preprocess data for calibration, 1135
pre-release
of library, 1534
pretty print, 285, 538
print, 465
print preview, 465
printable documentation, 324

printPlot, 970
profiling, 1017
  basic profiling, 1018
  fine grained profiling, 1020
  profiling of function calls, 1021
profiling of FMUs, 1292
Program Files, 1543
progress bar, 617
propagate, 575
  media propagation, 366
  parameter, 575
  selecting GUI tab for propagated parameter, 388
propagation of parameters, 116
protected, 479
  variables - storage of, 786
protecting variables, 1486
protection group, 1196
Python interface, 1365

Q

quick access toolbar, 447
quit
  Dymola, 467

R

radio buttons in dialog, 1491
ReadModelicaFile, 1246
read-only
  encrypted models, 1197
real-time simulation, 1021, 1313
  export restrictions, 1314
  license, 1334
  options, 820
recent
  files, 467
  models, 468, 489
record. *See* also parameter : record
  searching, 397
rectangle
  rounded corners, 262
rectangles, 262
redeclare keyword, 158
redo, 469, 486, 490
reference files, 1216
refresh
  refreshing simulation results, 678
refresh FMU, 1288
regression testing, 1216, 1218
regular expressions, 461, 668
re-import FMU, 1288
relative tic mark, 674
relative tolerance, 895
release
  latest Dymola release available, 1579
removePlots, 970
removeResults, 971
removing graphical object. *See* erasing graphical object
rename
  classes, 547
  component, 235
replace
  text, 534
replaceable
  classes (local). *See* class - replaceable classes (local)
  components, 347
  variable, 578
replaceable model, 158
report generator, 1376
RequestOption, 974
rescale
  plot. *See* plot window zooming
reset
  settings, 525
reshaping objects, 198, 219
resolution
  using 4K screens, 1557
result file in variable browser
  refreshing simulation results, 678
result files in variable browser
  closing result files, 675
  default handling, 674
  display changed values, 672
  exporting selected simulation results, 678
  exporting simulation results, 678
  opening additional result files, 674
  selecting what result files to keep, 675
RGB, 473
robot demo, 34
roll
  animation view, 49, 761
  visualizer window, 1448
root finder, 898
root model, 203
rotate, 474
  animation view, 49, 761
  visualizer view, 1448

rotation, 1450
rounded corners, 262
route connections, 474, 554
run script, 46, 791, 833
RunScript, 926, 946
runtime
  concept, 1556
  license, need for, 904
run-time. *See* runtime

S

save
  animation setup, 833
  Encrypted File, 458
  Encrypted Total Model, 458
  parameter record, 395
  plot setup, 833
  settings, 522, 833
  start values to model, 625
  variables, 833
save all, 457
save as, 457
  results, 895
save log, 466
save model, 467
save script, 833
save total, 457
savelog, 953
SaveModel, 1246
saveSettings, 953
SaveTotalModel, 1246, *See* also Save Total… (in GUI)
scaling, 1450
Scientific Data Format (SDF), 420
scrambling, 1212
script, 46, 123
  change to script directory before execution, 838
  echo script commands, 838
  edit customized setup file startup.mos, 837
  editor, 926
  files, 904
  functions, 909
  generating a, 833
  insert current plot commands in script, 930
  insert output settings in script, 932
  insert translation settings in script, 931
  new, 832
  open, 453, 833
  path - getting the path to a currently executed script, 936
  run, 791, 833
  save, 833
  startup script, 1559
script editor, 926
  cut, 837
  delete, 837
  duplicate, 837
  paste, 837
  replace, 838
  select all, 838
script editor tab
  change to script directory, 838
  copy, 837
  echo [script] commands, 838
  edit startup, 837
  generate script, 833
  go to, 838
  new script, 832
  open script, 833
  run script, 833
  save script, 833
  save script as, 833
scroll
  animation view, 49, 761
  visualizer view, 1448
SDF (Scientific Data Format), 420, 790
SDF Editor, 422
search
  bus signals (expandable connectors), 606
  class or component, 458
  component browser content, 212
  components in diagram layer, 482
  connctions (and connected components), 257
  connections (and connected components), 483
  package browser content, 208
  text (inside a class), 534
  usage of class, 458
selecting object
  in visualizer window, 1448
selecting objects, 218, 470, 487, 491
selection of model in dialog, 1496
selections
  of variables, 404
semiLinear, 1439
send to back, 476
set corner radius, 263
setClassText, 947
SetDymolaCompiler, 946
setting parameters, 62
settings
  custom setup file, 1570

disable saving changing of settings in script, 525
export, 525
import, 525
prevent reading and saving of settings, 525
prevent saving of settings, 525
reset, 525
save settings, 522
setup
animation, 861
compiler, 817
plot. *See* plot setup menu
simulation, 799
setup file, 523
custom setup file, 1570
customized .mos setup file, 524
S-function MEX block, 1303
sharable licenses, 1552
shell command
dymosim, 904
shift key, 218, 252
Shift+double-clicking
a component in the diagram layer, 518
short integer, 565
shortcut
to demos, 454
to libraries, 453
shortcuts
list of shortcuts, 138
to (ribbon) tabs, 190
to model (class) layers, 189
Show Variables, 621
SI units, 437
signal
detrending, 1139
filtering, 1144
limiting, 1139
signal operators, 722
definitions, 733
signalOperatorValue, 971
Simlink Real-Time, 1321
simplify
if-then-else expressions, 1009
simulate, 797
model - simulateExtendedModel (built-in function), 946
model - simulateModel (built-in function), 947
model - simulateMultiExtendedModel (built-in function), 948
simulate model, 59
simulateExtendedModel, 946
simulateModel, 947
simulateMultiExtendedModel, 948, 949, 950
simulation, 46, 614
active simulation model, 325
documentation, 122, 771
efficiency, 1005
interval, 801
log, 608
max run time, 800
max run time for individual simulations, 800
results. See result files in variable browser
setup, 57, 799, *See also* simulation setup
static, 949
time unit, 803
simulation analysis
equation incidence, 640
event log, 654
numeric integration, 649
plot dependencies, 633
timers, 666
simulation mode, 18
simulation setup, 799
compiler tab, 817
debug tab, 812
FMI import tab, 825
general tab, 800
output tab, 809
real-time tab, 820
translation tab, 805
simulation seutp
FMI export tab, 823
simulation tab
add command, 793
animation time and speed [toolbar], 831
backstep, 831
continue, 827
import initial, 827
linearize, 827
load result, 829
new animation [window], 830
new plot window, 830, 839
new table window, 830
new visualize 3D [window], 830
organize commands, 795
pause, 831
play, 831
reverse, 831
rewind, 831
run script, 791
save in model, 829
setup, 799
show log, 831
simulate, 797
simulation analysis, 830

steady state, 797
step forward, 831
stop, 797
sweep parameters, 798
translate, 796
visualize, 830
working with the diagram layer, 680
Simulink, 24, *See also* Matlab
DymolaBlock, 1303
external input and output, 1310
FMI Kit for Simulink, 1292
FMU export from Simulink, 1292
Generate Result, 1306
graphical interface, 1302
implementation notes, 1312
parameters and initial values, 1303
parameters and initial values, 1313
troubleshooting, 1616
singularity
structural, 998
size (built-in function), 960
smooth, 473
smoothOrder, 1386, 1387
snapshots
periodic snapshots during simulation, 985
solution of nonlinear equations, 1433
source code generation, 1335, 1339
license, 1334
source code generation features for normal translation, 1340
translateModelExport, 1339
space
components automatically, 246
sparse solvers, 802, 900
specification
FMI for Co-simulation, 1250
FMI for model exchange, 1250
split model, 338
SSP, 1294
export, 1297
import, 1295
SSV
import, 1297
stability boundary, 897
stacked connectors, 353
stand-alone application, 1335, 1336, *See* also StandAloneDymosim
StandAloneDymosim, 1335, 1340
compiling and linking, 1341
declare_, 1344
dsblock, 1342
FindEvent_, 1344
GetDimensions, 1344
NextTimeEvent, 1344
trouble-shooting, 1345
start values
discriminating, 1024
fixed, 71, 168, 243
free start values, 1114
save to model, 114, 625
starting Dymola, 31
startup script, 1559
startup directory. *See* directory : startup directory
startup file
customized startup file, 524
state event, 163, 1405
state machines, 331, 1418
state selection, 561, 1402
attribute in parameter dialog, 386
dynamic, 604, 814, 1403
logging, 814
statements
inserting, 294
stateSelect, 561
StateSelect, 1402
static calibration, 1146
static simulation, 949
staticCalibrate, 1146
steady state
initialization, 1037
solver interface, 797
stop simulation when steady state is reached, 1033
steady-state calibration. *See* static calibration
step size, 896
(advanced) control of, 782
stiff, 897, 898
stop
Dymola, 467
simulation, 797
store
as one file, 479
in model, 799, 803, 807, 811, 815
of protected variables, 786
stream connector, 156
string
manipulation, 912
parameters, 393
string variables in models, 1422
structural
parameters, 784
singularities, 998

structurally incomplete, 999
structurally singular, 1426
style, 1449
style checking, 1225
  as the Check > Style command, 1228
styleData, 1466, 1467
  Level Curves, 1467
  Surface, 1467
  Vector Field, 1467
  Water Fall, 1467
subcomponents
  display subcomponents, 220
  Show Component, 220
submodel, 338
Subversion. *See* model management : version control
SUNDIALS suite of numerical solvers, 1268
surface matrices, 1458
surface plots, 1458
SVN, 1184, *See* model management : version control
SVN commands, 1171
SVN selection, 1175
sweep parameters, 798
sweepManyParametersScatter, 1082
sweepOneParameter
  experimentation, 1078
  experimentation with new GUI, 1044
sweepParameter
  calibration, 1121
  experimentation, 1074
  experimentation with new GUI, 1044
sweepTwoParameters
  calibration, 1129
  experimentation, 1080
  experimentation with new GUI, 1044
sweepTwoParametersMulti, 1082
symbolic processing, 1425
symbols
  =, 910
synchronize values in the variable browser with the simulation time, 670
synchronous equations, 160
Synchronous Modelica, 1418
syntax
  check, 288
  errors, 538
syntax check, 55
SysML
  export, 506
system requirements
  hardware, minimum, 1601
  hardware, recommended, 1601
  software, Linux, 1603
  software, Windows, 1601
System Structure and Parametrization (SSP), 1294
system variable
  Path, 1177

T

T matrix, 1449, 1450
tab information level, 529, 869
table
  insert in documentation, 316
table handling
  efficient code generation for large tables, 1432
  import/export files, 420
  interpolation, 909
  Matlab routines, 1312, 1313
  n-dimensional, 422, 1312, 1313
  plot, 564
  script, 909
table window
  create a table winow using a built-in function, 959
TableND block, 422
TableND package, 1312, 1313
tabs
  Tools, 495
tabs in dialog, 1483
tearing, 1426
temperature
  absolute, 425
  difference, 425
templates, 369
terminate
  Dymola, 467
test
  case, 1216
  suite, 1216
text
  "null-extent" text object, 265
  alignment, 312
  bold, 311
  bullet list, 312
  font, 310
  font size, 311
  indent - decrease, 312
  indent - increase, 312
  italic, 311
  minimize font size, 267, 516
  numbered list, 312
  size. *See* font size

strings, 265
style, 310
subscript, 311
superscript, 311
texts in plots, 751
underline, 311
text tab
back, 489
check, 491
check normal, 491
check pedantic, 491
check with simulation, 491
copy, 490
cut, 490
delete, 490
duplicate, 490
find and replace, 493
flat Modelica, 490
forward, 489
go to, 493
insert, 494
mathematical notation, 489
model check - stop checking, 492
Modelica, 489
paste, 490
recent, 489
redo, 490
select all, 491
undo, 490
used classes, 489
variable selections, 494
textString, 1458
thread-safe
external functions, 1008
tilt
animation view, 761
visualizer view, 1448
tolerance, 802
toolbar
documentation layer, 302
quick access, 447
top left, 447
tools tab
[export] graph, 503
[export] graph > DSM, 506
[export] graph > export..., 506
[export] graph > UML, 504
[export] HTML…, 497
about, 526
copy to clipboard, 495
Dymola Website, 526
export animation, 496
export image, 495
export to clipboard, 495
help documents, 526
library management, 507
options, 511
version, 526
Tools tab, 495
tooltip
for short description of functions, 284
for x- amd y-axis in plots, 703
tooltips
copy to clipboard, 701
dynamic, 218
dynamic tooltips for signals in plot, 700
for legends in plot, 702
plot window, 700
top left toolbar, 447
tracing
bus signals (expandable connectors), 606
in a script, 932
transform matrix
view, 1446
translate
model – export to other simulation environments, 796
model - translateModel (built-in function), 950
model to export - translateModelExport (built-in function), 950
model to FMU - translateModelFMU (built-in function), 951
translate model, 59, 615
FMU, 797
normal, 796
translateModel, 950
translateModelExport, 950, 1339
translateModelFMU, 951
translation, 1450
log, 604
translation statistics, 1216
traversing
models before translation, 1238
translated models, 1246
troubleshooting
compiler, 1613
language, 1617
license borrowing, 1612
license file, 1611
license server, 1612
other windows problems, 1617
Simulink, 1616
trouble-shooting
stand-alone application, 1345

tunable parameters
declaring as fixed, 1258
tuners. *See* parameter tuning : tuners
tuning. *See* parameter tuning
type prefix, 379
causality, 381
connector members, 381
dynamic typing, 382
final, 381
protected, 381
replaceable, 381

U

UML
export, 504
undo, 469, 486, 490
undock, 202, 765
unit
adding unit to output of block, 360
checking, 437, 440
conversion. *See* displayUnit
deduction, 437, 441
display. *See* displayUnit
unlock classes, 335
unused parameters, 492
update
Modelica annotations, 1537
upgrading Dymola, 1556
URI, 312
modelica URI scheme, 269, 314
URL, 776
used classes
layer, 196
used classes layer, 489
user-defined GUI. *See* GUI
UTF-8, 327

V

variability type. *See* type prefix
variable
alias. *See* alias variables
annotations, 382
attribute fixed, 243
browser, 588, 869, 873
changes, display in variable browser, 672
declaration, 291, 376, 484, 494
declaration dialog, 379
hidden, 1486
in Modelica, 151
independent variable in plot, 629
interactive, 910
interface variable, 405
matching, 404
predefined, 912
protected, 1486
replaceable, 578
selections, 404
store of protected, 786
type prefix. *See* type prefix
variable browser context menu
animate, 872
animate together, 872
open all, 870
simulation analysis, 872
variable order (algorithms), 897
variable step size, 896
variable structure systems, 164
variables, 951
vector
field, 1458
vendor-specific annotations, 1423
verifyCompiler, 951
version, 481
version control. *See* model management : version control
version management, 1167
view
transform matrix, 1446
View Parameter Settings, 573
visual
modeling, 597
reference, 862
Visualize 3D, 1443
window number, 1446
visualize simulation, 830
visualizer window, 598, 1445
moving view, 1448
pan, 1448
roll, 1448
rotate view, 1448
scroll view, 1448
selecting object, 1448
tilt view, 1448
zooming, 1448
VRML, 496, 867

W

warnings, 70, 104, 116, *See* also errors
widgets. *See* GUI widgets
  adding GUI widgets in parameter dialog, 393
window
  animation, 46, 596
  command, 599, 812
  command history, 601, 780
  Dymola Main, 31, 187
  edit, 35, 189
  grid, 197
  information browser, 215, 613
  library, 213
  log window, 603
  model view, 49, 590, 620, 679
  number (Visualize 3D), 1446
  plot, 46, 591
  types, 186, 584
  undock, 765
  variable browser, 588
  visualizer, 598, 1445
window menu
  cascade windows, 528
  close all, 528
  log window, 528
  new command window, 529
  new Dymola window, 528
  new library window, 528
  tile windows, 528
Windows
  Dymola as 32-bit application, 1542
  Dymola as 64-bit application, 1542
  Dymola license server, 1585
windows menu, 528
Wireframe, 1468
working directory. *See* directory : current working directory
workspace, 768

X

X3D
  as HTML/JS, 496, 867
  as pure XML, 496, 867
XML
  model description for FMI, 1250
XML interface, 1337
xPC Target, 1321

Z

zoom factor
  diagram layer, 469
zooming
  animation window, 49, 761
  concurrent. *See* plot window zooming
  diagram layer, 217, 469, 682
  fit window, 469
  icon layer, 217
  plot window, 102, *See* plot window zooming
  visualizer window, 1448