

# **Dymola**

## Dynamic Modeling Laboratory

### User Manual

### Volume 1

September 2014

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

Document version: 17. Important additions/corrections compared with the previous Dymola documentation “March 2014” (doc. version 16) are marked in the margin.

© Copyright 1992-2014 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 Science Park  
SE-223 70 Lund  
Sweden

E-mail: <http://www.3ds.com/support>  
URL: <http://www.Dymola.com>  
Phone: +46 46 2862500

# Contents

<b>1</b>	<b>What is Dymola?</b>	<b>11</b>
1.1	Features of Dymola .....	11
1.2	Architecture of Dymola.....	12
1.3	Basic Operations .....	13
1.3.1	Simulating an existing model .....	13
1.3.2	Building a model .....	16
1.4	Features of Modelica.....	19
1.4.1	Background.....	20
1.4.2	Equations and reuse .....	20
1.4.3	Modelica history .....	20
<b>2</b>	<b>Getting started with Dymola</b>	<b>23</b>
2.1	Introduction .....	23
2.2	Simulating a model — industrial robot .....	24
2.2.1	Investigating the robot in modeling mode .....	24
2.2.2	Simulation.....	33
2.2.3	Other demo examples .....	37
2.3	Solving a non-linear differential equation.....	38
2.3.1	Simulation.....	41
2.3.2	Improving the model .....	46
2.4	Using the Modelica Standard Library .....	54
2.4.1	The Modelica Standard Library.....	54
2.4.2	Creating a library for components .....	62
2.4.3	Creating a model for an electric DC motor.....	64
2.4.4	Documenting the model.....	70
2.4.5	Testing the model .....	73

2.4.6	Handling the warnings .....	77
2.4.7	Creating a model for the motor drive.....	81
2.4.8	Parameter expressions .....	85
2.4.9	Documenting the simulation.....	90
2.4.10	Scripting .....	92
2.5	Building a mechanical model .....	94
2.6	Other libraries .....	99
2.6.1	Libraries available in the File menu by default .....	99
2.6.2	Libraries that can be added.....	101
2.7	Help and information .....	101
2.7.1	Reaching compressed information .....	101
2.7.2	Reaching more extensive information .....	103
<b>3</b>	<b>Introduction to Modelica .....</b>	<b>107</b>
3.1	Modelica basics.....	107
3.1.1	Variables.....	109
3.1.2	Connectors and connections .....	109
3.1.3	Balanced models.....	110
3.1.4	Partial models and inheritance.....	111
3.2	Acausal modeling.....	112
3.2.1	Background.....	112
3.2.2	Differential-algebraic equations .....	113
3.2.3	Stream connector support .....	114
3.3	Advanced modeling features .....	115
3.3.1	Vectors, matrices and arrays.....	115
3.3.2	Class parameters.....	116
3.3.3	Algorithms and functions .....	117
3.4	Hybrid modeling in Modelica .....	117
3.4.1	Synchronous equations .....	118
3.4.2	Relation triggered events .....	121
3.4.3	Variable structure systems .....	122
3.5	Initialization of models .....	125
3.5.1	Basics.....	126
3.5.2	Continuous time problems .....	126
3.5.3	Parameter values.....	130
3.5.4	Discrete and hybrid problems .....	131
3.5.5	Example: Initialization of discrete controllers.....	133
3.6	Standard libraries .....	136
3.6.1	Modelica Standard Library .....	136
3.6.2	Modelica Reference .....	137
3.6.3	Other libraries .....	137
3.7	References .....	137
<b>4</b>	<b>Developing a model .....</b>	<b>143</b>
4.1	Windows in Modeling mode .....	144
4.1.1	Dymola Main window .....	144
4.1.2	Edit window .....	146
4.1.3	Package and component browsers .....	154
4.1.4	Library window .....	158

4.1.5	Command window.....	158
4.1.6	Message window .....	159
4.1.7	Information browser .....	159
4.2	Basic model editing.....	161
4.2.1	Basic operations.....	161
4.2.2	Packages, models, and other classes .....	164
4.2.3	Components and connectors .....	168
4.2.4	Connections .....	177
4.2.5	Creating graphical objects .....	184
4.2.6	Changing graphical attributes .....	190
4.2.7	Programming in Modelica .....	193
4.2.8	Documentation .....	215
4.2.9	Support for national characters .....	233
4.3	Advanced model editing .....	235
4.3.1	State Machines.....	235
4.3.2	Splitting models.....	239
4.3.3	Components and connectors .....	246
4.3.4	Building templates using replaceable components .....	259
4.3.5	Parameters, variables and constants.....	261
4.3.6	Matching and variable selections.....	270
4.3.7	Using data from files .....	277
4.3.8	Display units .....	279
4.3.9	Changing graphical attributes .....	281
4.3.10	Viewing graphical attributes.....	282
4.4	Checking the model.....	283
4.4.1	Commands .....	283
4.4.2	Unit checking and unit deduction in Dymola .....	283
4.5	Editing model reference .....	289
4.5.1	Window settings .....	289
4.5.2	Default view of classes .....	290
4.5.3	Package browser details.....	290
4.6	Editor command reference – Modeling mode .....	291
4.6.1	Main window: File menu.....	291
4.6.2	Main window: Edit menu .....	308
4.6.3	Main window: Commands menu.....	323
4.6.4	Main window: Window menu .....	325
4.6.5	Main window: Help menu .....	328
4.6.6	Main window: Linear analysis menu.....	331
4.6.7	Context menu: Edit window (Icon layer).....	332
4.6.8	Context menu: Edit window (Diagram layer).....	332
4.6.9	Context menu: Edit window (Documentation layer).....	333
4.6.10	Context menu: Edit window (Modelica Text layer) .....	335
4.6.11	Context menu: Edit window (Modelica Text layer – mathematical notation) .....	340
4.6.12	Context menu: Edit window (Used Classes layer) .....	341
4.6.13	Context menu: Package browser .....	342
4.6.14	Context menu: Component browser .....	343
4.6.15	Context menu: Components .....	343
4.6.16	Context menu: Coordinate system boundary.....	350

4.6.17	Context menu: Connection while connecting .....	350
4.6.18	Context menu: Graphical objects.....	351
4.6.19	Graphical object menus: Line and fill style .....	353
4.6.20	Context menu: Variable declaration dialog; Value input field .....	355
4.6.21	Context menu: Parameter dialog; plot window .....	362
4.6.22	Context menu: Parameter dialog; parameter input field .....	362
<b>5</b>	<b>Simulating a model.....</b>	<b>369</b>
5.1	Windows in Simulation mode .....	370
5.1.1	Dymola Main window .....	371
5.1.2	Variable browser.....	372
5.1.3	Diagram layer window .....	373
5.1.4	Plot window.....	374
5.1.5	Animation window .....	376
5.1.6	Visualizer window.....	377
5.1.7	Command window.....	378
5.1.8	Script editor .....	380
5.1.9	Message window .....	382
5.1.10	Information browser .....	388
5.2	Model simulation .....	389
5.2.1	Basic steps .....	389
5.2.2	Variable browser interaction.....	392
5.2.3	Using the diagram layer.....	401
5.2.4	Plotting .....	404
5.2.5	Plot window interaction.....	405
5.2.6	Errors and warnings when translating .....	439
5.2.7	Defining Graphical Objects for the animation window .....	442
5.2.8	Animation window interaction .....	444
5.2.9	Using the command window .....	447
5.2.10	Documentation .....	452
5.2.11	Simulation settings .....	460
5.3	Editor command reference – Simulation mode .....	462
5.3.1	Main window: File menu.....	462
5.3.2	Main window: Simulation menu .....	466
5.3.3	Main window: Plot menu .....	486
5.3.4	Main window: Animation menu.....	497
5.3.5	Main window: Commands menu.....	503
5.3.6	Main window: Window menu .....	504
5.3.7	Main window: Help menu .....	504
5.3.8	Context menu: Variable browser – nodes.....	504
5.3.9	Context menu: Variable browser – signals .....	506
5.3.10	Context menu: Plot window .....	507
5.3.11	Context menu: Plot window – curve and legend .....	508
5.3.12	Context menu: Plot window – signal operators .....	509
5.3.13	Context menu: Plot window – text objects .....	510
5.3.14	Context menu: Table window.....	511
5.3.15	Context menu: Animation window.....	511
5.3.16	Context menu: Visualizer window .....	512

5.3.17	Context menu: Command window – Command log pane .....	513
5.3.18	Context menu: Command window – Command input line.....	515
5.3.19	Context menu: Script editor.....	516
5.3.20	Context menu: Message window.....	519
5.4	Dynamic Model Simulator .....	520
5.4.1	Overview .....	520
5.4.2	Running Dymosim.....	520
5.4.3	Selecting the integration algorithm.....	522
5.4.4	Dymosim reference.....	527
5.5	Scripting .....	529
5.5.1	Basic facts about scripting in Dymola .....	529
5.5.2	Scripting in Dymola – Functions/Script files .....	532
5.5.3	The possible content in a Modelica function or script file.....	533
5.5.4	What is a Modelica function? .....	536
5.5.5	Creating a Modelica function .....	537
5.5.6	Saving a function call as a command in the model.....	538
5.5.7	Editing a Modelica function .....	538
5.5.8	Executing a Modelica function call .....	539
5.5.9	Advanced use of functions.....	540
5.5.10	Pre-defined Modelica functions.....	540
5.5.11	What is a Modelica script file? .....	540
5.5.12	Creating a Modelica script file .....	541
5.5.13	Saving a script file as a command in the model.....	544
5.5.14	Editing a Modelica script file .....	544
5.5.15	Running a Modelica script file .....	548
5.5.16	The Dymola script editor .....	549
5.5.17	Some examples of Modelica script files .....	553
5.5.18	Predefined Modelica script files .....	556
5.5.19	Built-in functions in Dymola .....	557
5.6	Debugging models .....	583
5.6.1	Guidelines for model development.....	583
5.6.2	Error messages.....	585
5.6.3	Direct link in the error log to variables in model window .....	585
5.6.4	Event logging.....	588
5.6.5	Model instability.....	588
5.6.6	Output of manipulated equations in Modelica format .....	589
5.6.7	Checking for structural singularities.....	599
5.6.8	Bound checking for variables .....	600
5.6.9	Online diagnostics for non-linear systems.....	601
5.6.10	Diagnostics for stuck simulation .....	602
5.7	Improving simulation efficiency .....	604
5.7.1	Time of storing result .....	604
5.7.2	Simulation speed-up .....	605
5.7.3	Events and chattering.....	607
5.7.4	Debug facilities when running a simulation .....	608
5.7.5	Profiling.....	611
5.7.6	Inline integration.....	614
5.8	Handling initialization and start values .....	617

5.8.1	The continue command .....	617
5.8.2	Over-specified initialization problems.....	617
5.8.3	Discriminating start values .....	618
<b>6</b>	<b>Appendix — Installation.....</b>	<b>623</b>
6.1	Installation on Windows .....	624
6.1.1	Dymola as 32-bit and 64-bit application.....	624
6.1.2	Installing the Dymola software.....	624
6.1.3	Installing a C compiler .....	629
6.1.4	Installing the Dymola license file .....	632
6.1.5	Additional setup.....	635
6.1.6	Changing the setup of Dymola .....	643
6.1.7	Removing Dymola.....	644
6.1.8	Installing updates.....	644
6.2	Installation on Linux .....	644
6.2.1	Installing Dymola .....	645
6.2.2	Additional setup.....	645
6.2.3	Removing Dymola.....	646
6.3	Dymola License Server on Windows.....	647
6.3.1	Background.....	647
6.3.2	Installing the license server .....	647
6.3.3	License borrowing .....	651
6.4	Dymola License Server on Linux.....	656
6.5	Utility programs .....	657
6.5.1	Obtaining a host id.....	657
6.6	System requirements .....	658
6.6.1	Hardware requirements.....	658
6.6.2	Hardware recommendations .....	658
6.6.3	Software requirements .....	659
6.7	License requirements .....	660
6.7.1	General .....	660
6.7.2	License for Dymola – Simulink interface.....	661
6.7.3	License for real-time simulation.....	661
6.7.4	Licenses for exporting code.....	661
6.7.5	Licenses for executing/importing/using code .....	662
6.7.6	Licenses for libraries in the Dymola library menu .....	664
6.7.7	Licenses for libraries not in Dymola library menu .....	665
6.8	Troubleshooting .....	665
6.8.1	License file .....	665
6.8.2	Compiler problems .....	667
6.8.3	Simulink .....	669
6.8.4	Change of language .....	670
6.8.5	Other Windows-related problems.....	670
<b>7</b>	<b>Index .....</b>	<b>671</b>

# **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 acasual connections. Model libraries are available in many engineering domains.

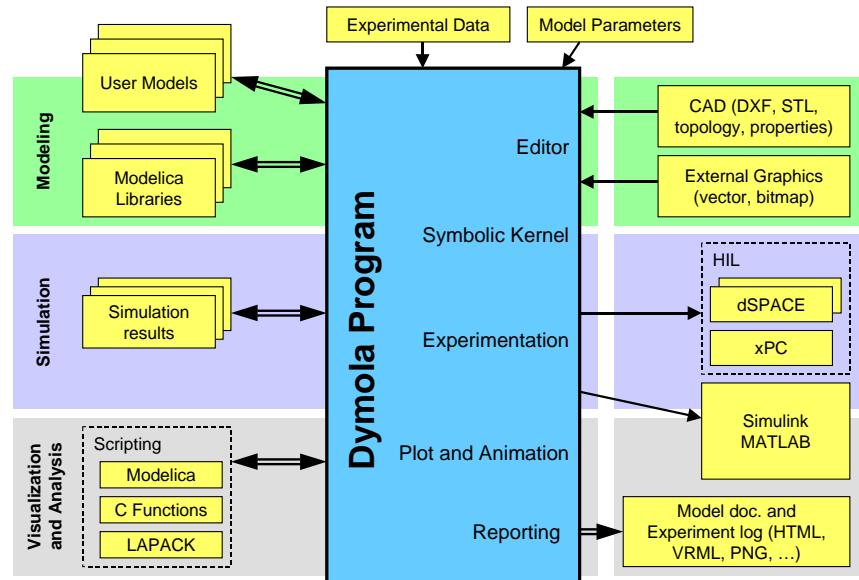
Dymola uses a new modeling methodology based on object orientation and equations. The usual need for manual conversion of equations to a block diagram is removed by the use of automatic formula manipulation. Other highlights of Dymola are:

- Handling of large, complex multi-engineering models.
- Faster modeling by graphical model composition.
- Faster simulation – symbolic pre-processing.
- Open for user defined model components.
- Open interface to other programs.
- 3D Animation.
- Real-time simulation.

## 1.2 Architecture of Dymola

The architecture of the Dymola program is shown below. Dymola has a powerful graphic editor for composing models. Dymola is based on the use of Modelica models stored on files. Dymola can also import other data and graphics files. Dymola contains a symbolic translator for Modelica equations generating C code for simulation. The C code can be exported to Matlab/Simulink and hardware-in-the-loop platforms.

Dymola has powerful experimentation, plotting and animation features. Scripts can be used to manage experiments and to perform calculations. Automatic documentation generator is provided.



## 1.3 Basic Operations

Dymola has three kinds of windows: Main window, Library window and Command window. The Main window operates in one of two modes: Modeling and Simulation.

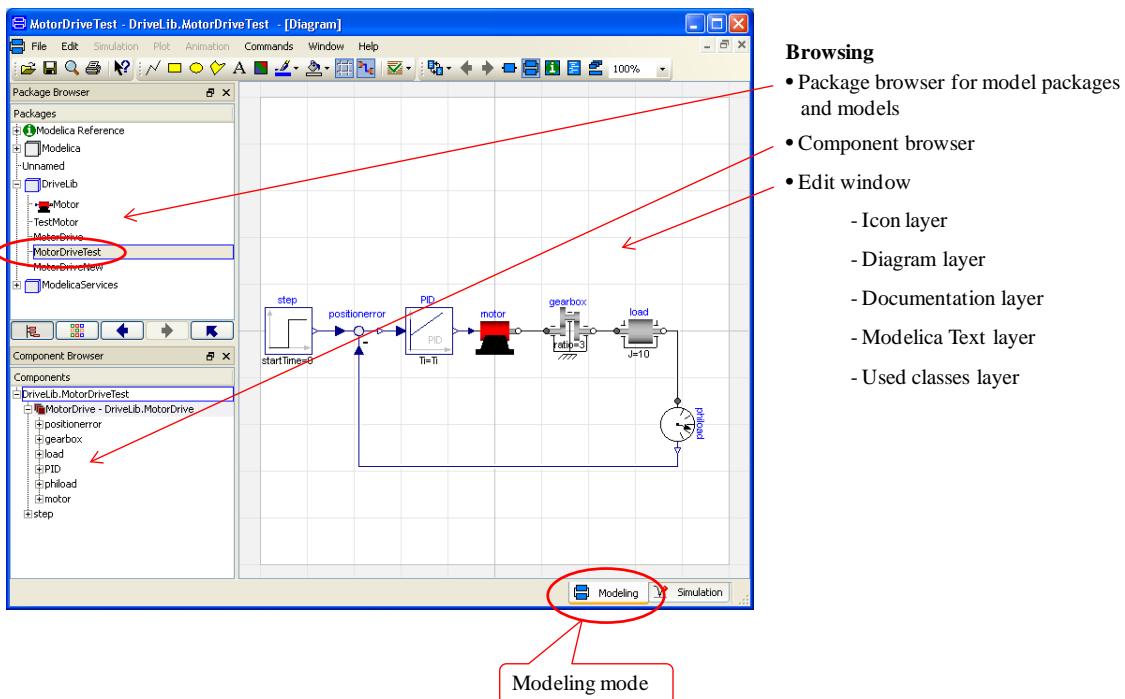
The Modeling mode of the Main window is used to compose models and model components.

The Simulation mode is used to make experiment on the model, plot results and animate the behavior. The Simulation mode also has a scripting sub-window for automation of experimentation and performing calculations.

### 1.3.1 Simulating an existing model

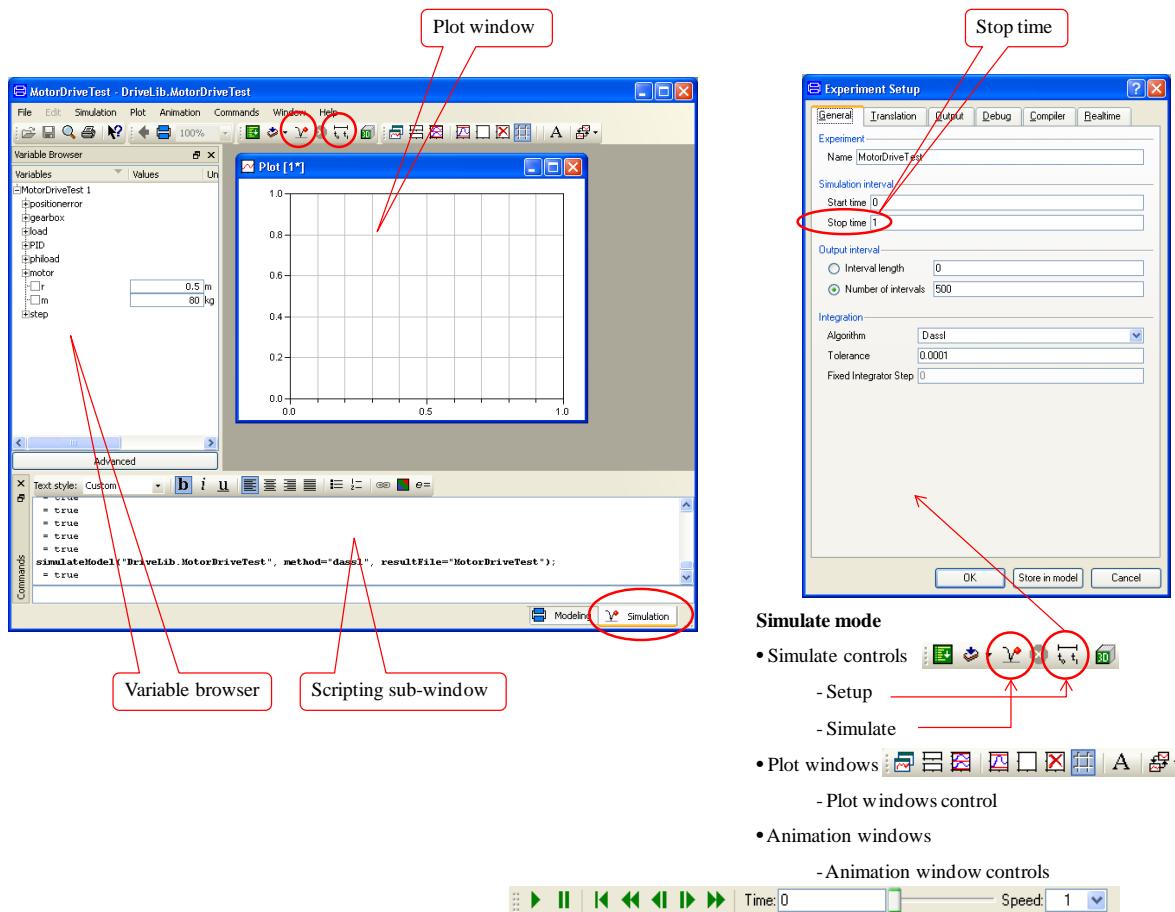
#### Finding the model

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

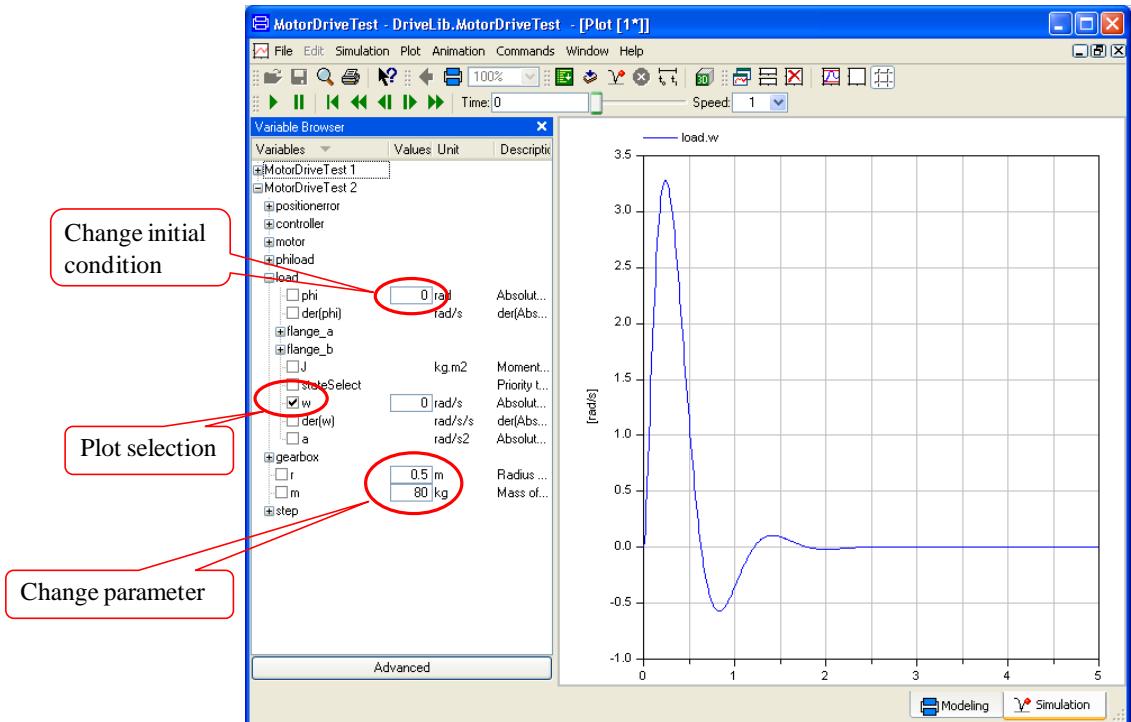


## Simulation Mode

The Simulation Mode is used for experimentation. It has a simulation setup to define duration of simulation, etc., plot windows, animation windows and variable browser.



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



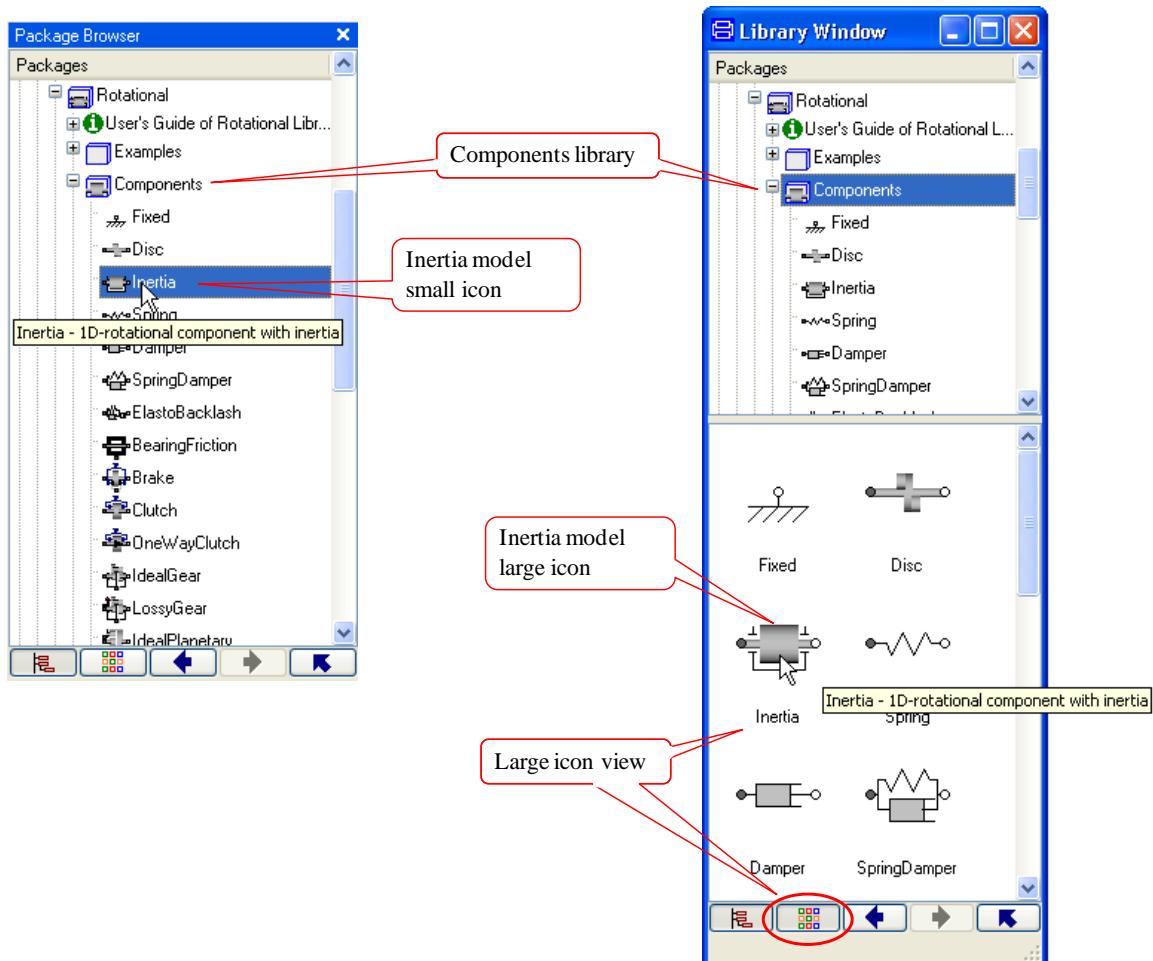
An animation window shows a 3D view of the simulated model. The animation can be run at different speeds, halted, single stepped and run backwards.

## 1.3.2 Building a model

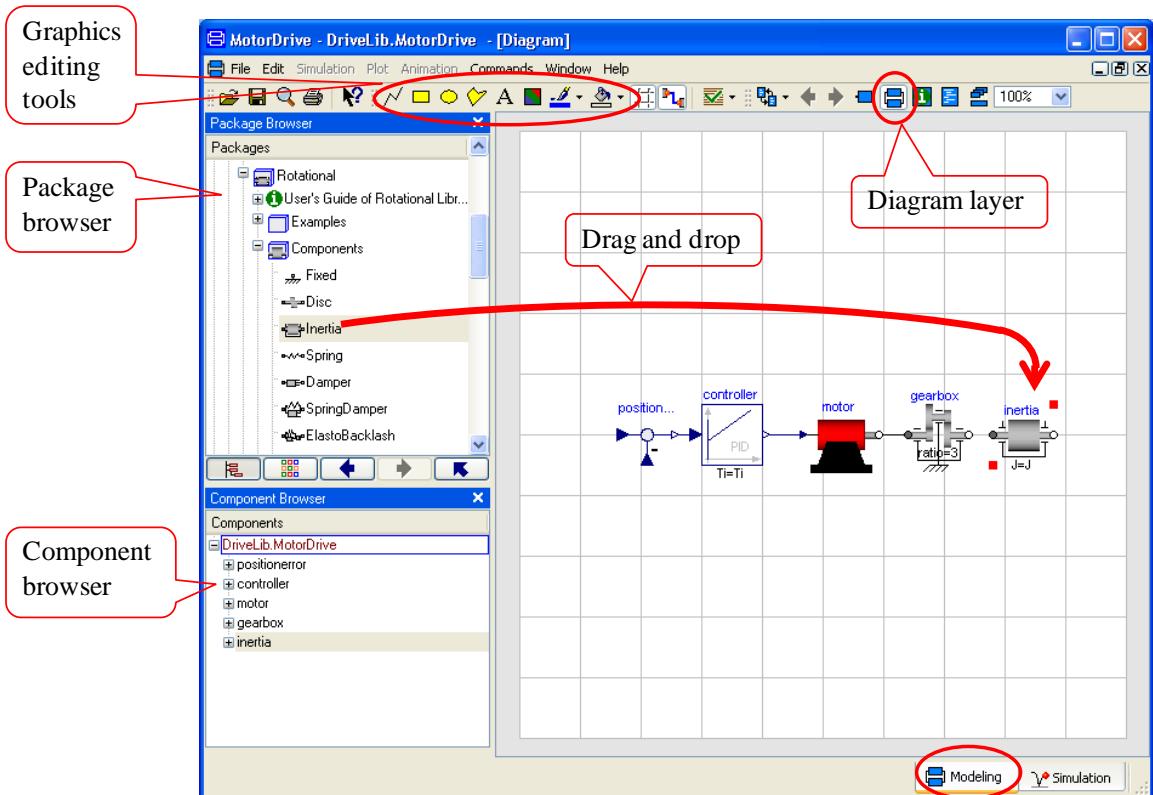
The graphical model editor is used for creating and editing models in Dymola. Structural properties, such as components, connectors and connections are edited graphically, while equations and declarations are edited with a built-in text editor.

### Finding a component model

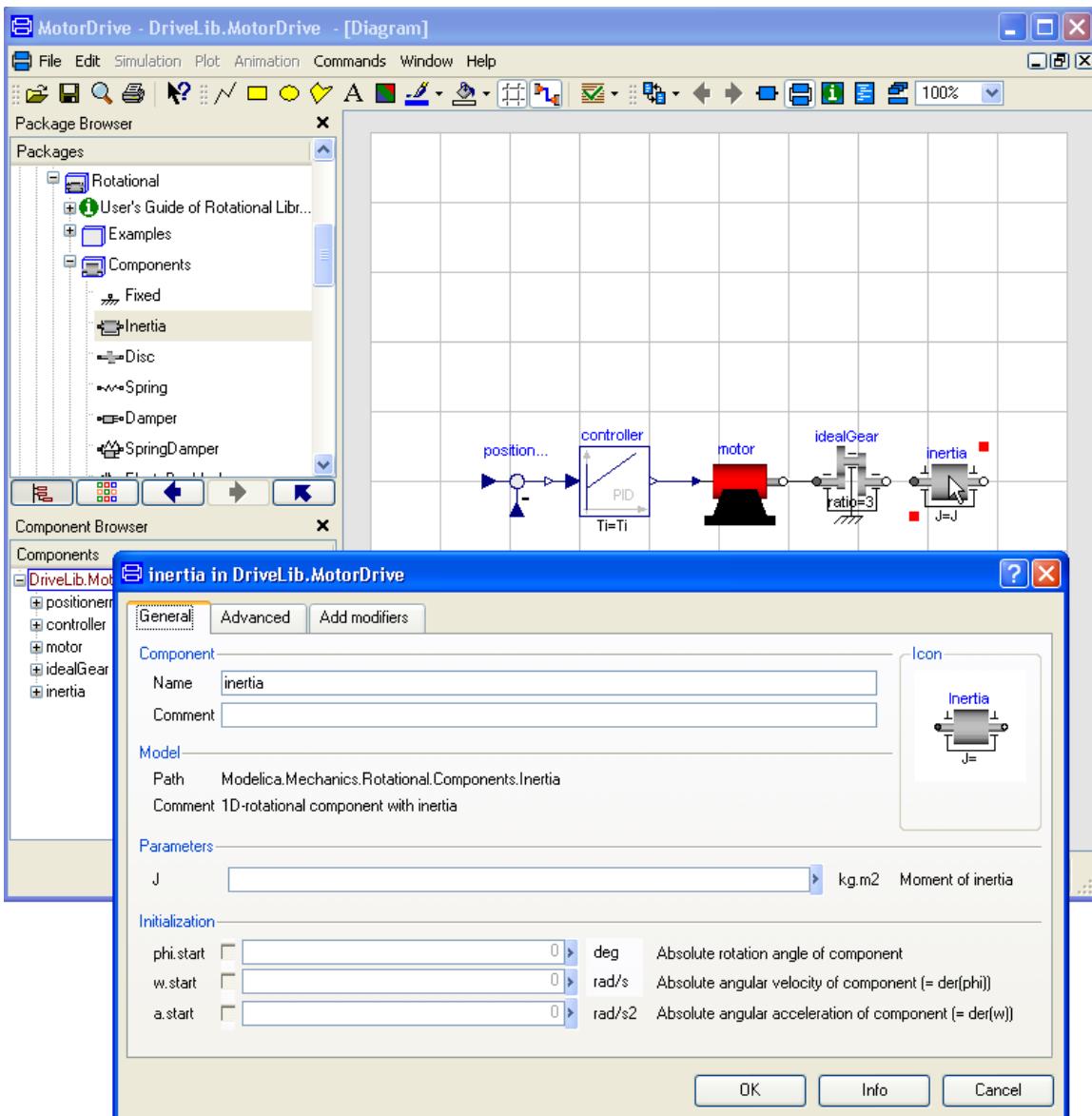
The package browser allows viewing and selecting component models from a list of models with small icons. It is also possible to get a large icon view.



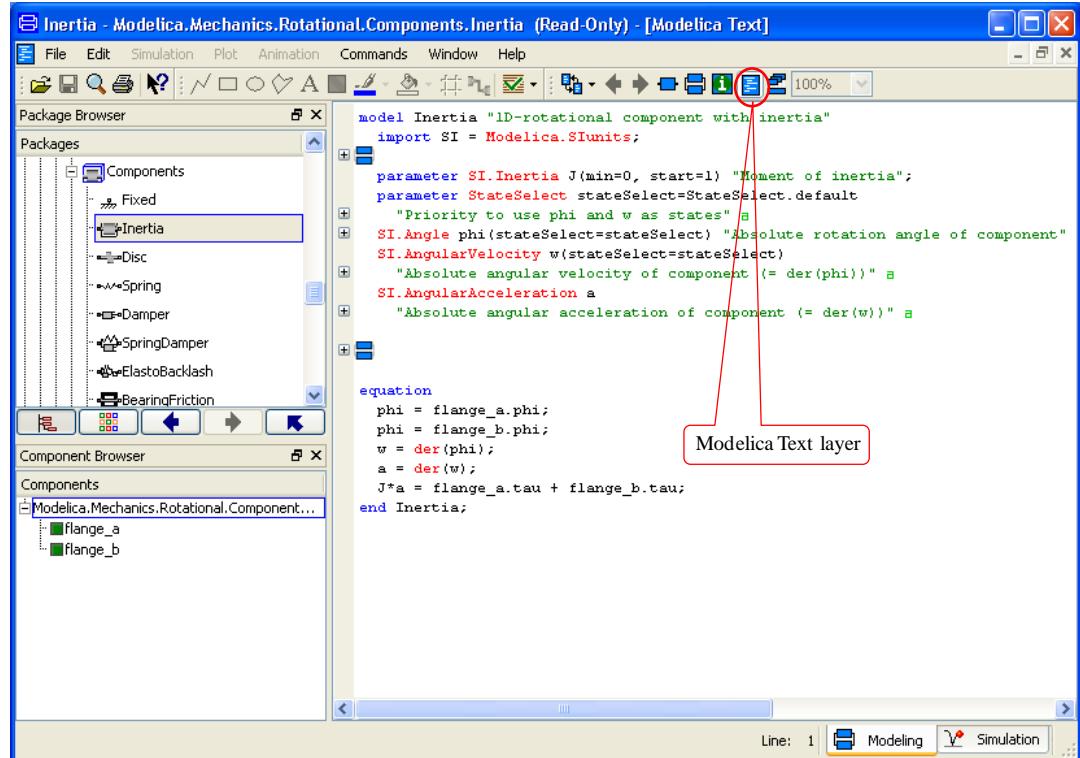
Components are dragged from the package browser to the diagram layer and connected. The component hierarchy is shown in the component browser.



By double clicking on a component, a dialog for giving the name of the component and its parameters is shown.



The component browser allows opening a component for inspection of the documentation or the model itself e.g. by looking at the underlying Modelica code which is shown in the Modelica Text layer. This is also the editor for entering Modelica code, i.e. declarations and equations for low level models.



## 1.4 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.

## 1.4.1 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.

## 1.4.2 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.

## 1.4.3 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>). Modelica is intended to serve as a standard format so that models arising in different domains can be exchanged between tools and users. It has been designed by a group of more than 50 experts with previous know-how of modeling languages and differential-algebraic equation models. After more than 50 three-days' meetings during more than 10 years, version 3.0 of the language specification was finished in September, 2007.

A paper by Hilding Elmquist: "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.

## **2 GETTING STARTED WITH DYMOLA**



# 2 Getting started with Dymola

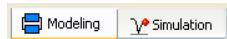
---

## 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 menu **Help > Documentation**. 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 101 for more information.

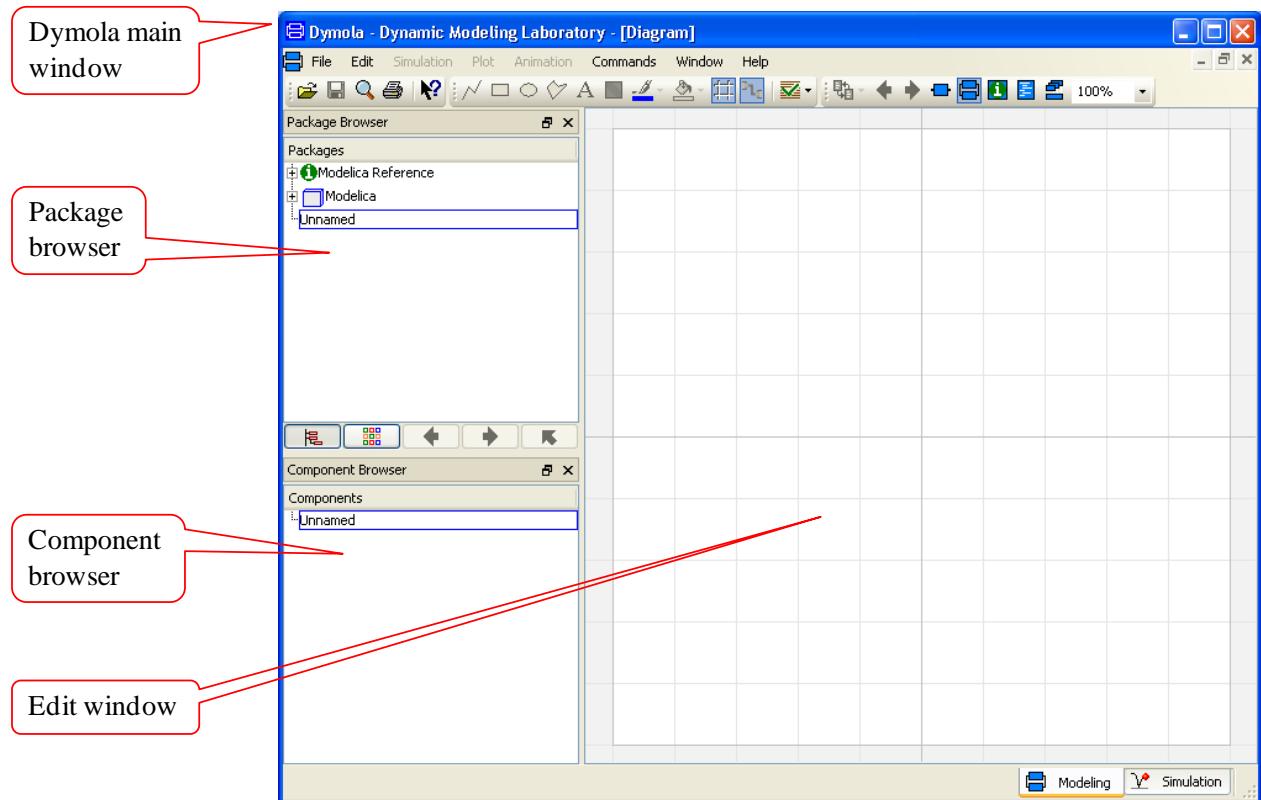
Start Dymola. The main Dymola window appears. A Dymola main window operates in one of the two modes:

- **Modeling** for finding, browsing and composing models and model components.
- **Simulation** for making experiments on the model, plotting results, and animating behavior.



Dymola starts in Modeling mode. The active mode is selected by clicking on the tabs in the bottom right corner of the Dymola window.

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.



---

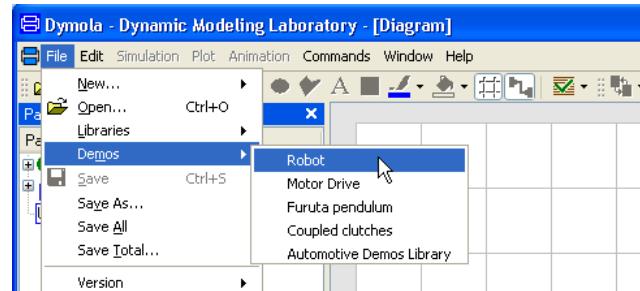
## 2.2 Simulating a model — industrial robot

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 38.

### 2.2.1 Investigating the robot in modeling mode

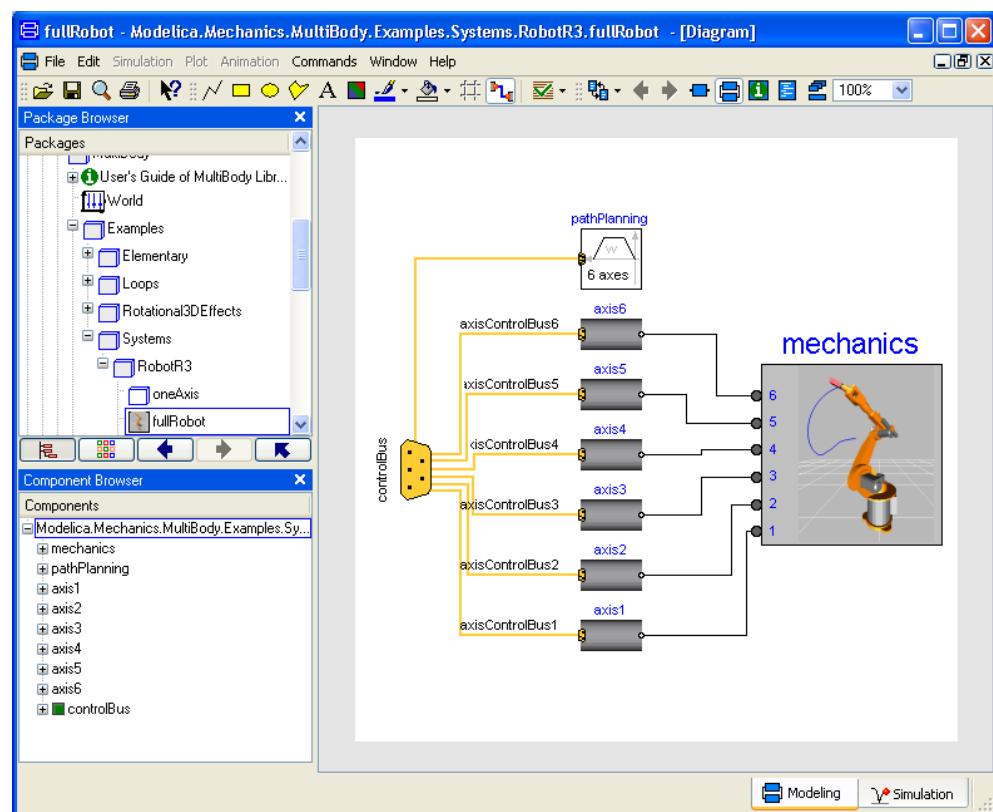
We will study a model of an industrial robot. To view the industrial robot model, use the **File > Demos** menu and select **Robot**.

**Opening a demo example.**



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

**The robot demo.**



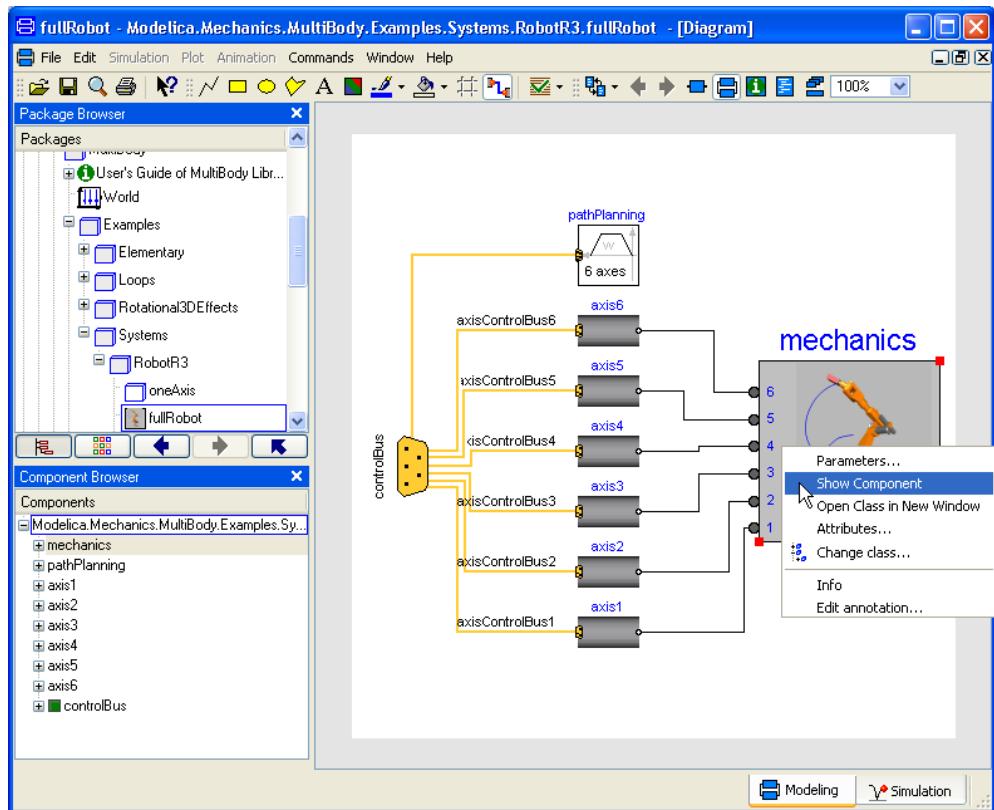
The package browser in the upper left sub-window displays the package hierarchy and it is now opened up with the robot model selected and highlighted. The model diagram in the edit window (the sub-window to the right) shows the top-level structure of the model. The model diagram has an icon for the model 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.

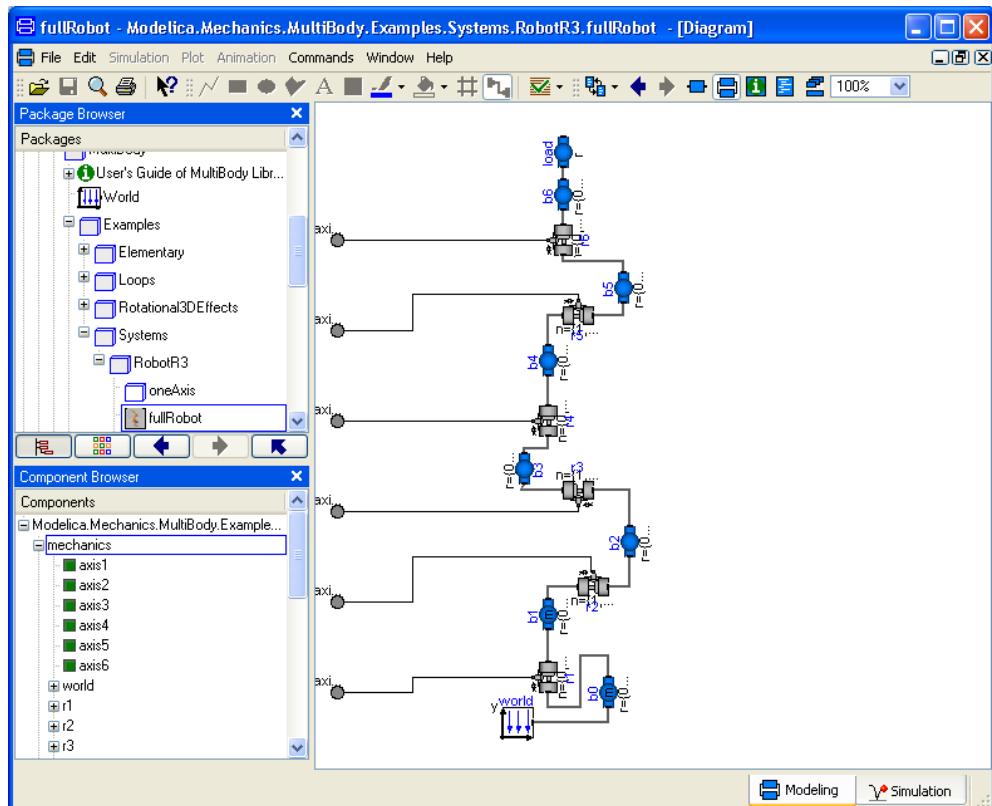


It is not necessary to select the robot component explicitly by first clicking with the left button on the mouse on it to access its menu. It is sufficient to just have the cursor on its icon in the edit window and right-click. The component browser also gives easy access to the robot component. Just 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.

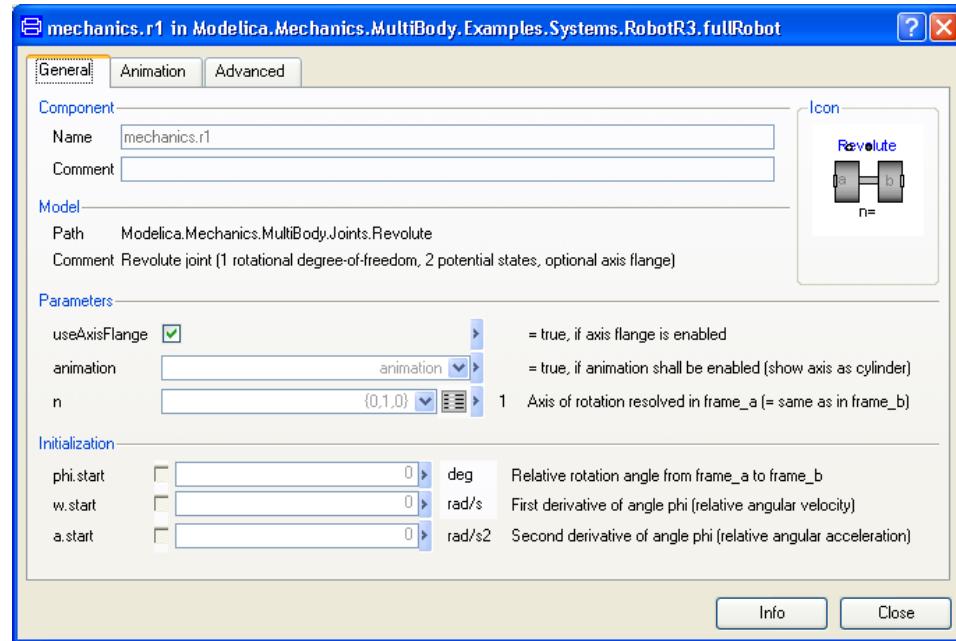
The edit window now displays the mechanical structure consisting of connected joints and masses. The component browser is opened up to also show the internals of the mechanics model component.

### The mechanical structure of the robot.



Double-click on, for example,  $r1$  at the bottom of the edit window. This is a revolute joint. The parameter dialog of that component appears. The parameter dialog can also be accessed from the right button menu. Double-clicking the left button selects the first alternative from the right button menu.

## Parameter dialog.

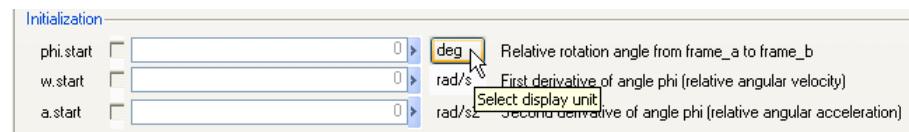


The parameter dialog allows the user to inspect actual parameter values. In demos the parameter values are write-protected to avoid unintentional changes of the demo example – then the dialog just has a **Close** button (and an **Info** button). When the parameter values can be changed there is one **OK** 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.

A parameter dialog may have several tabs. This dialog has the tabs: **General**, **Animation**, **Advanced** and **Add modifiers**. 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.

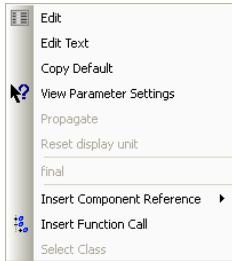
If prepared for, display units can be selected in the dialog. Units that have alternatives are marked by white background (here phi.start and w.start have selectable display units). By resting the cursor over such a unit a button is displayed,

## Selectable display unit.



and by clicking on that the selection can be made:

## Alternatives of selectable display unit.



**Initialization**

phi.start	<input type="text" value="0"/>	<input checked="" type="checkbox"/> rad	Relative rotation angle from frame_a to frame_b
w.start	<input type="text" value="0"/>	<input type="checkbox"/> deg	First derivative of angle phi (relative angular velocity)
a.start	<input type="text" value="0"/>	<input type="checkbox"/> rad/s <sup>2</sup>	Second derivative of angle phi (relative angular acceleration)

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.

**Parameters**

useAxisFlange	<input checked="" type="checkbox"/>	= true, if axis flange is enabled
animation	<input type="text" value="animation"/>	= true, if animation shall be enabled (show axis as cylinder)
n	<input type="text" value="({0,1,0})"/>	Axis of rotation resolved in frame_a (= same as in frame_b)

**Initialization**

phi.start	<input type="text" value="({1,0,0})"/>	deg	Relative rotation angle from frame_a to frame_b
w.start	<input type="text" value="({0,1,0})"/>	rad/s	First derivative of angle phi (relative angular velocity)
a.start	<input type="text" value="({0,0,-1})"/>	rad/s <sup>2</sup>	Second derivative of angle phi (relative angular acceleration)

**Info**

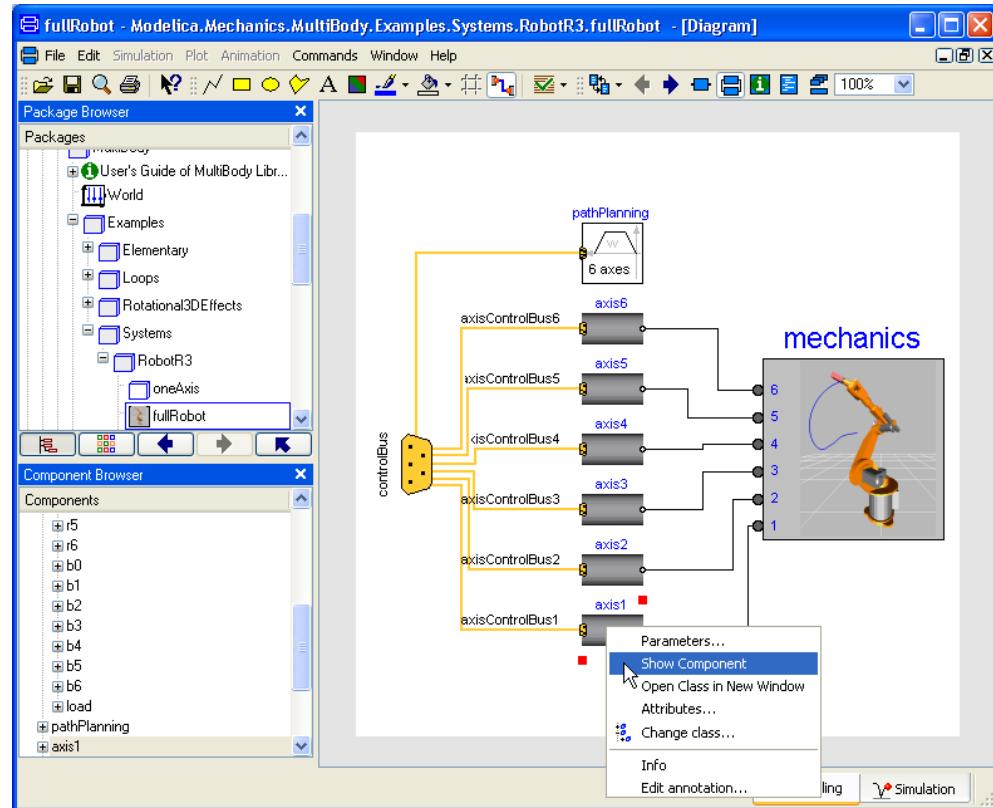
To learn more about the component, select **Info**. An information browser is opened to show the documentation of the revolute joint. Links in the document makes it easy to navigate to e.g. the description of the package containing the revolute joint. Now please close the browser and press **Close**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



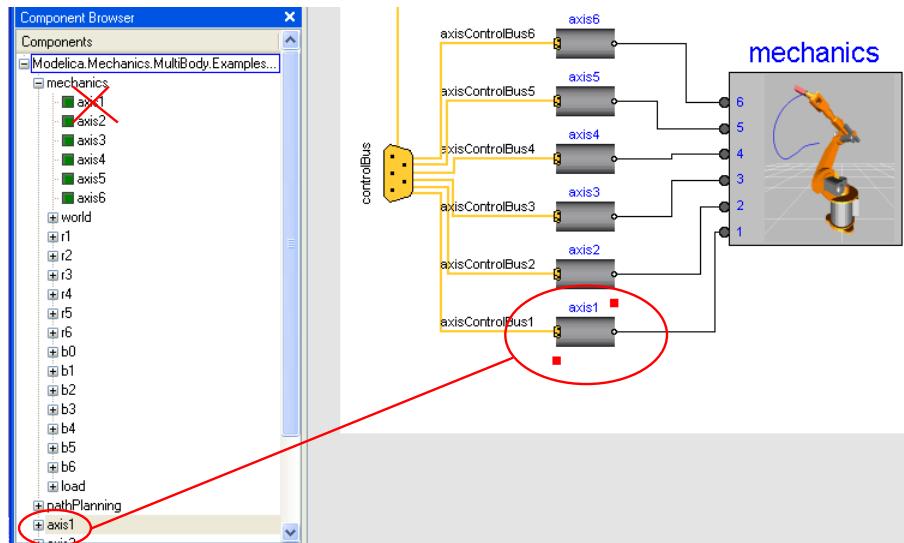
Let us now inspect the drive train model. There are several possible ways to get it displayed. Press the **Previous** button (the toolbar button with the bold left arrow) once to go to the robot model and then put the cursor on one of the axis icons and right-click. Please, note that robot.mechanics 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.**



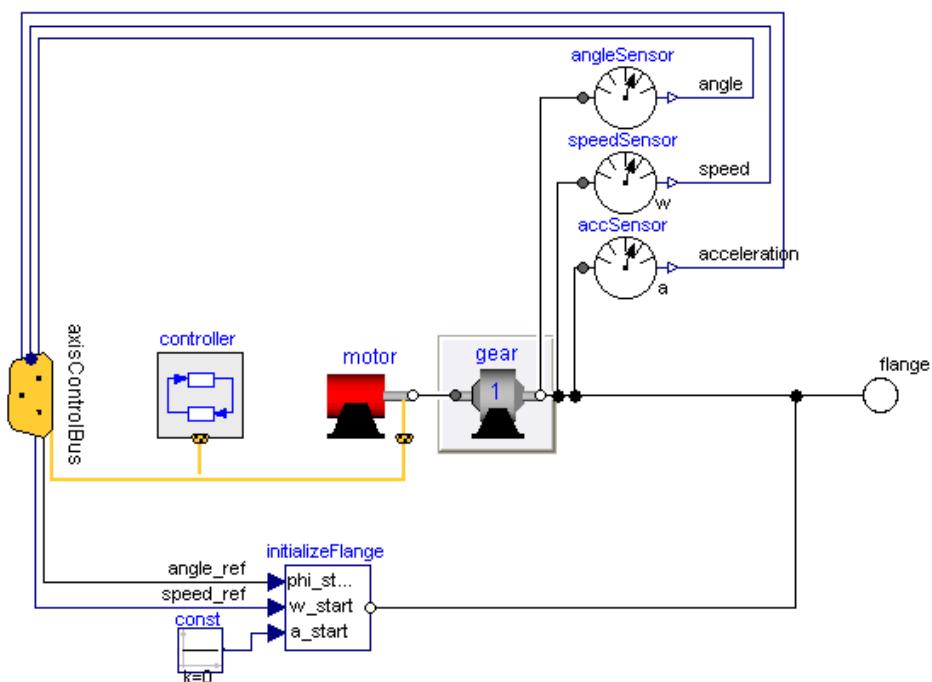
Another convenient way is to use the component browser. Put the cursor on top of the wanted component in the browser and right-click to get the context menu. Select **Show Component** as shown in the figure below. (In this case also care must be taken not to select any axes of the module mechanics. The component browser has been enlarged in the figure to illustrate this.) Since **Show Component** is the first menu option, double-clicking will yield the same result. Please recall that double-clicking on a component in the edit window pops up the parameter dialog (compare the two menus in the figure above and below!).

**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.**



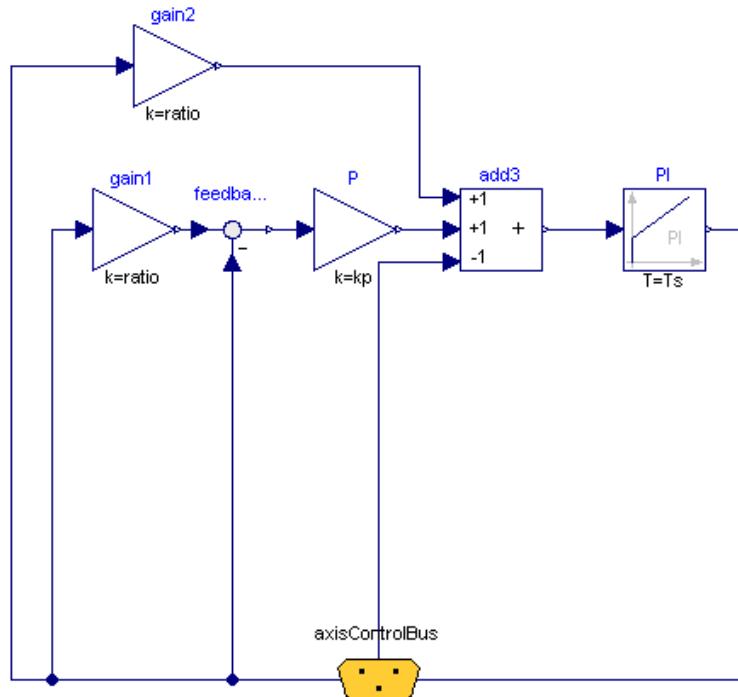
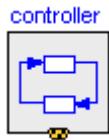
The drive train includes a controller. A data bus is used to send measurements and reference signals to the controller and control signals from the controller to the actuator. The bus for one axis has the following signals:

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.

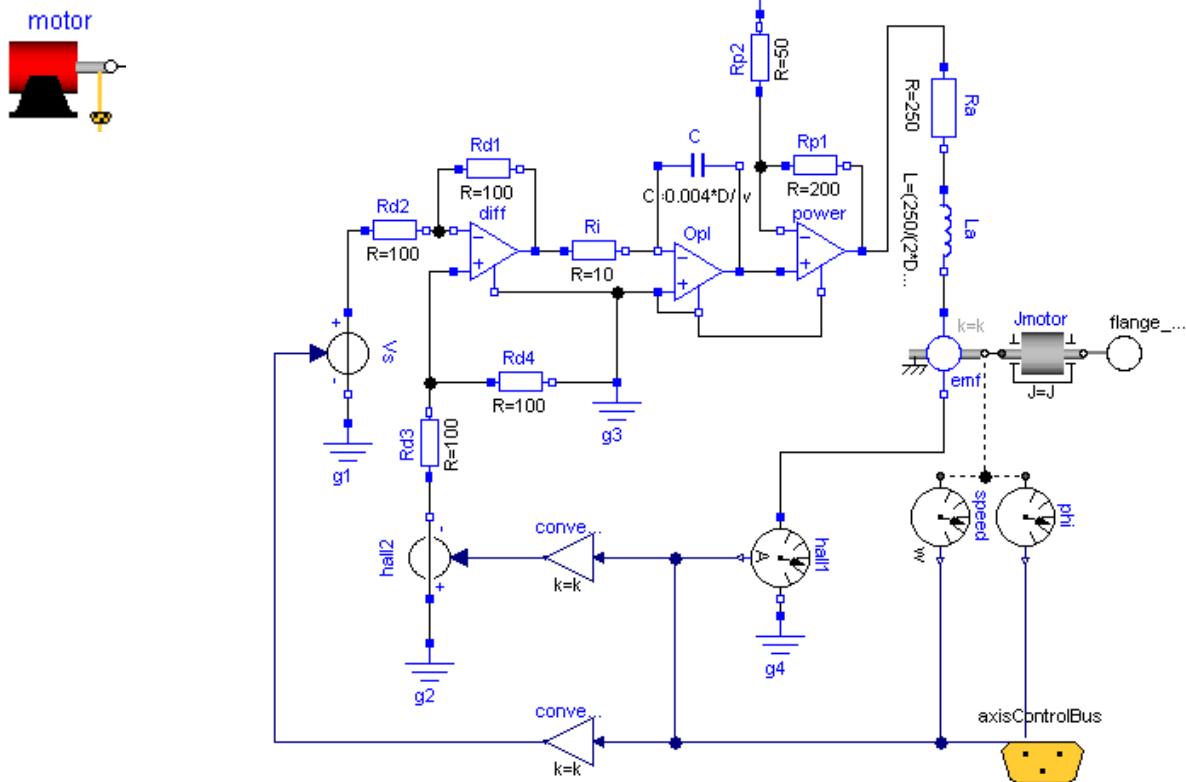
**The robot controller in the robot drive train.**



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.

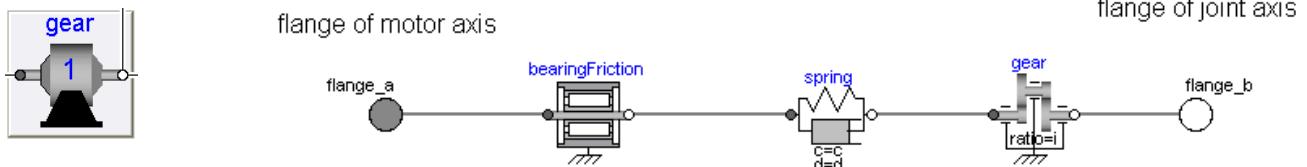
The motor model consists of the electromotical force, three operational amplifiers, resistors, inductors, and sensors for the feedback loop.

### The robot motor.

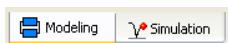


View the component gear in a similar way as for the other components. It shows the gearbox and the model of friction of the bearings and elasticity of the shafts.

### The robot gearbox.



## 2.2.2 Simulation



Let us simulate the robot model. To enter the Simulation mode, click on the tab **Simulation** at the bottom right of the main window. Please note that selecting Simulation mode is not the same as simulating the robot; it however gives the possibility doing so – the needed menus are available in this mode.

## Simulating/manipulating with given values

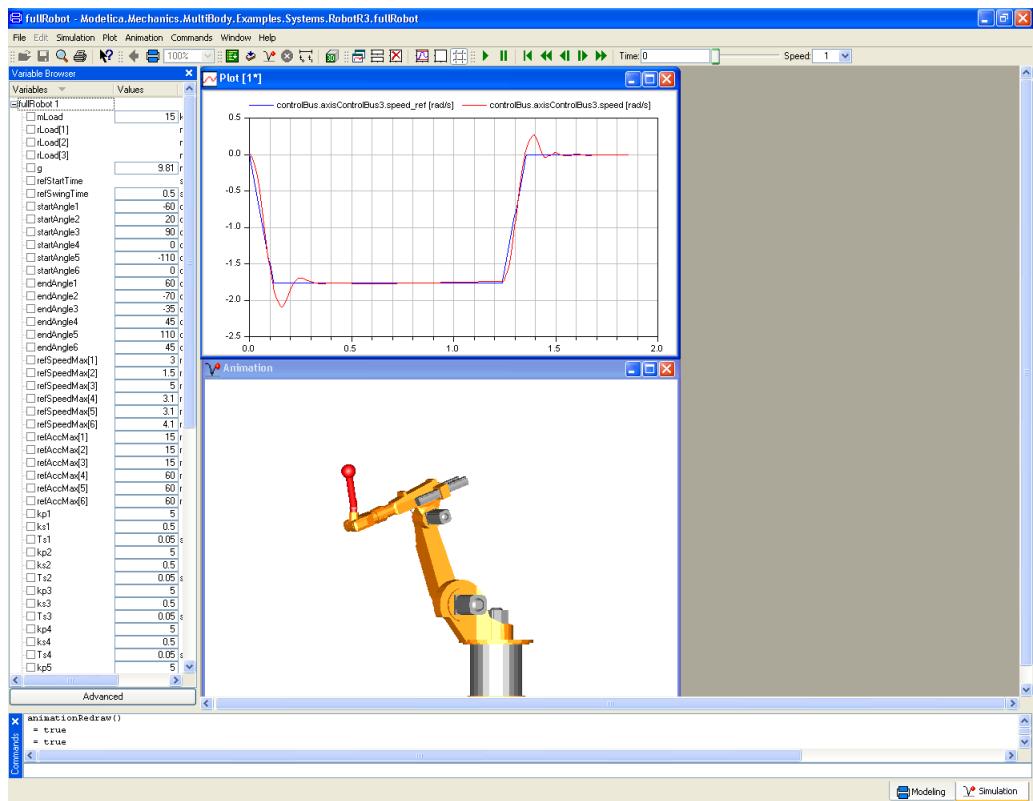
The Simulation menu contains commands to setup and run simulations. Shortcut buttons are also available. However, the demo has also been prepared with a command script that will simulate the model. The script is executed selecting **Commands > Simulate**. Please use that command.

### 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. After maximizing the Dymola main window it will look the following:

### Animated 3D view of the robot.



Let us start to animate the robot. Start the animation by selecting **Animation > Run** or clicking the **Run** button on the toolbar (the leftmost button)

#### Animation toolbar.



This toolbar contains the usual buttons of running, 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)

Please note that the Animation window can be maximized in the main window in the usual way.

If the Animation window by mistake is deleted, a new can be displayed using the command **Animation > New Animation Window**.

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.

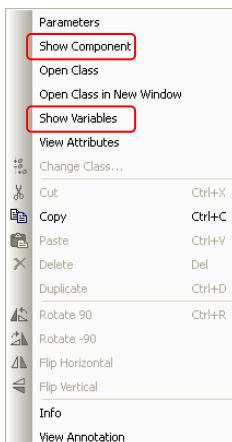
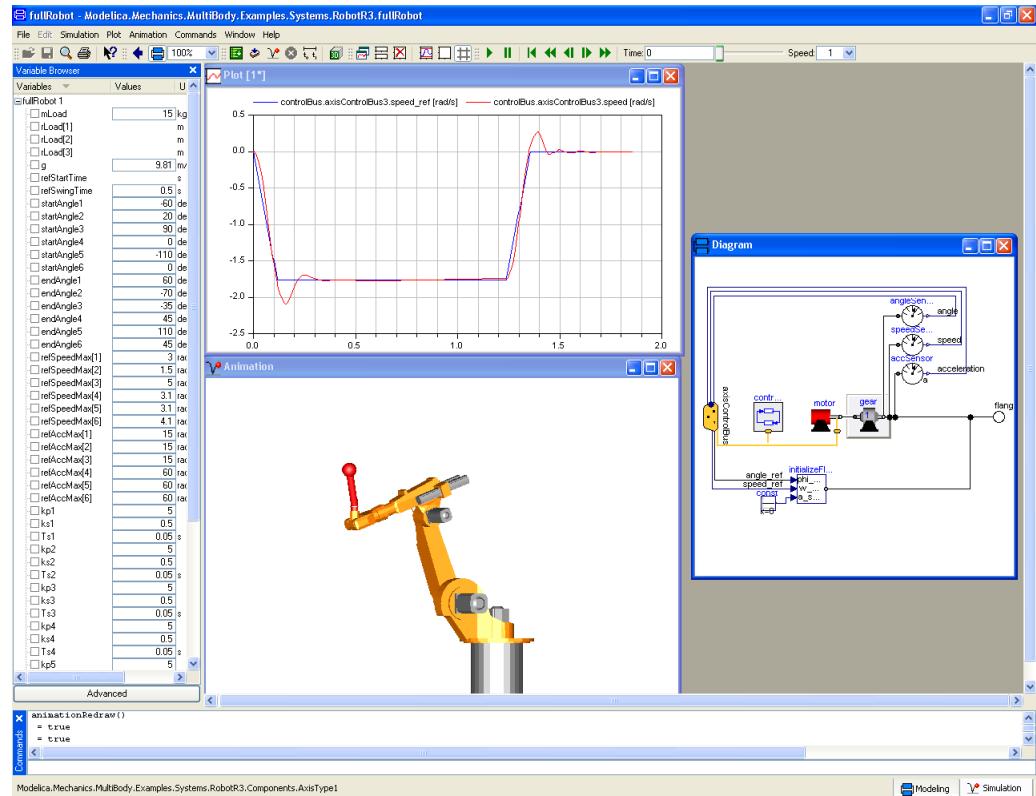
Let us now look at the plot window. The plot shows the speed reference and actual speed of the third joint.

A very convenient way to display the variables for a certain component that is of interest is to use the diagram layer of the edit window also in Simulation mode. The diagram layer 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.



Push the **Diagram** layer button in the diagram layer toolbar to show the diagram. The result might look like:

### The diagram layer in Simulation mode.



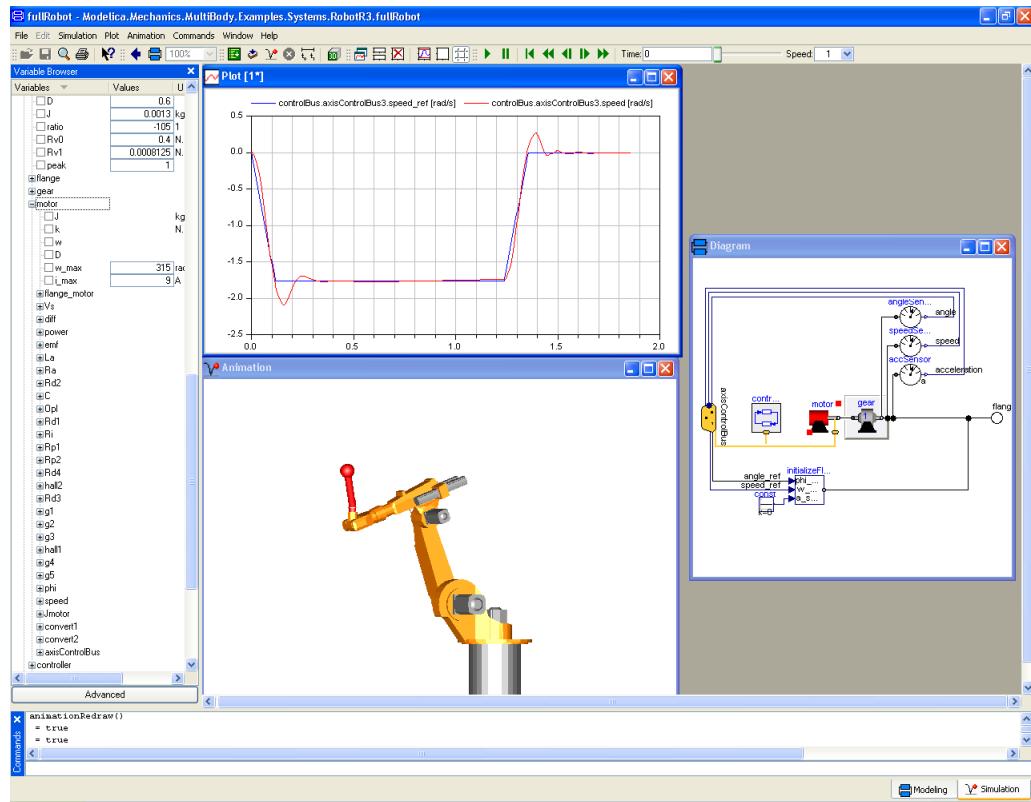
Now the diagram layer is visible. You can now navigate exactly the same way as in the Modeling mode – right-click on the component you want to look into and select **Show Component** to look into it.

You go back using the **Previous** button in the diagram layer toolbar

If you want the diagram layer to show more of an overview; e.g. the whole robot, you have to go back to Modeling mode and change what is displayed in the Edit window. Then you can go back to Simulate mode.

When having reached the interesting component, right-click on it and select **Show Variables**, which will open and highlight the selected component instance in the variable browser. In the figure below, the variables of the motor in Axis 1 is 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 84 for more information about this possibility.

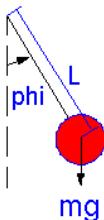
## 2.2.3 Other demo examples

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 Automotive Demos Library. It contains several demos, and the relevant demo has to be opened first by expanding the package “Examples” in the package browser by clicking the + before it, and then double-clicking on the relevant demo to open it. Please note that to run any of the Automotive Library 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

$$J \cdot \text{der}(w) = -m \cdot g \cdot L \cdot \sin(\phi)$$

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

$$J = m \cdot L^2$$

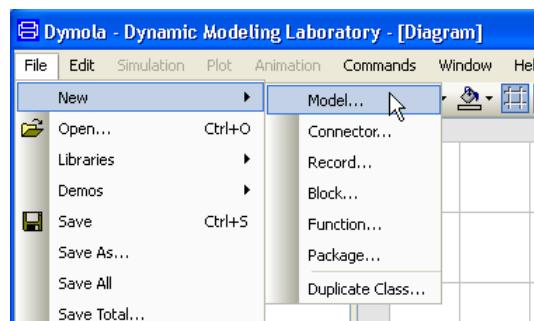
The variable  $w$  is the angular velocity and  $\text{der}(w)$  denotes the time derivative of  $w$ , i.e., the angular acceleration. For the angular position we have

$$\text{der}(\phi) = w$$

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

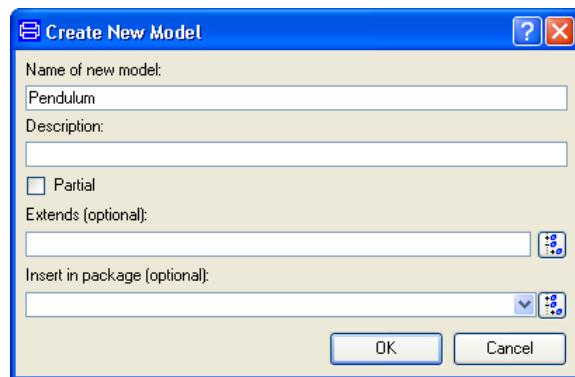
Click on the tab **Modeling** at the bottom right. Then select **File > New Model**.

**The first step to  
create a new model.**



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

**The dialog to name a new model component.**

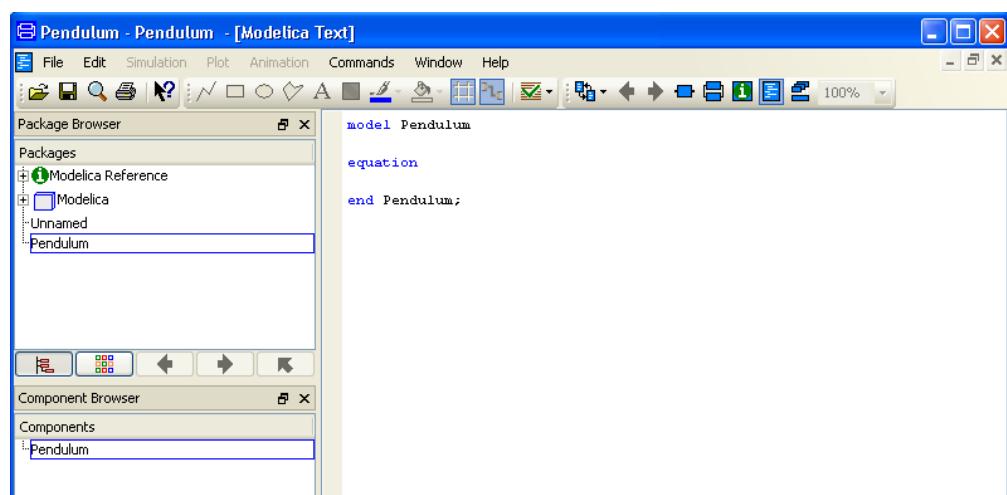


Click **OK**. You will then have to **Accept** that you want to add this at the top-level. You should in general store your models into packages, as will be described later.

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.



**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;

```

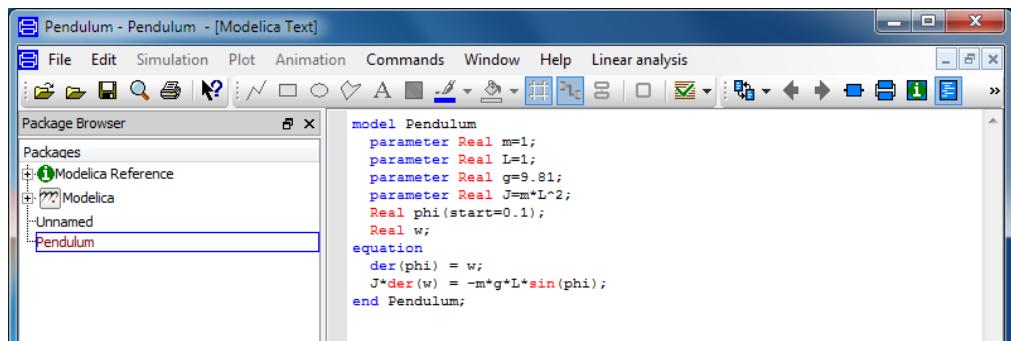
New text will be syntax highlighted (color coded) as you type, except types; e.g. Real. To get also types color coded, right-click and select **Highlight Syntax** or press **Ctrl+L**. Apart from implementing color codes for types, the command will also give a message if the syntax is not correct. It 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:

<b>blue</b>	keywords
<b>red</b>	types, operators etc.
<b>black</b>	values, variables, parameters etc.
<b>green</b>	comments

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

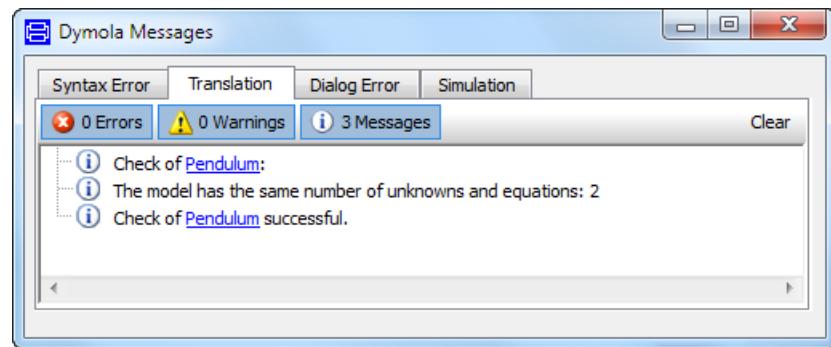
### Declaration of parameters, variables and equations.



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 or by selecting the command **Edit > Check** or by using the **F8** function key. The result of a check of the above text looks the following:

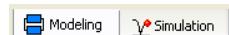


### Check of model.



The model is now ready to be saved. Select **File > Save**. Call the file pendulum and keep the file extension to .mo and place it in a working directory.

### 2.3.1 Simulation

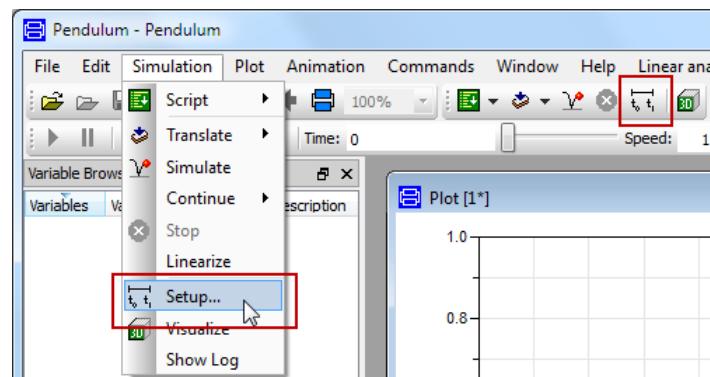


Now it is time to simulate. To enter the simulation mode, click on the tab **Simulation** at the bottom right of the main window. The simulation menu is now activated and new tool bar buttons appear.

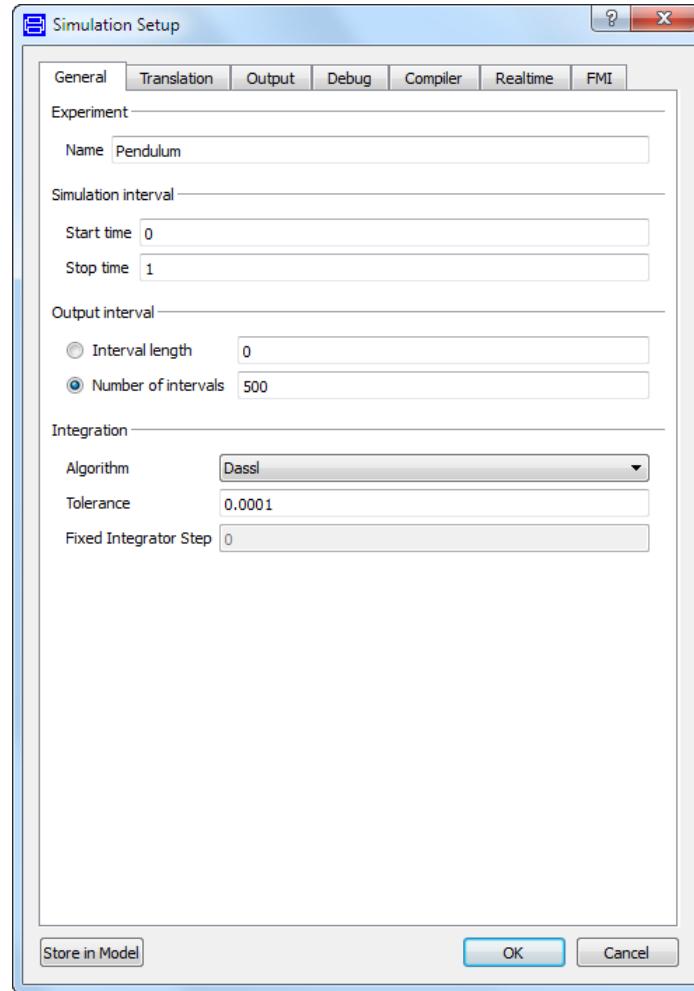


To set up the simulation select **Simulation > Setup...** or click directly on the **Setup** toolbar button.

#### Selecting Setup in the Simulation menu.



**The Simulation Setup menu.**

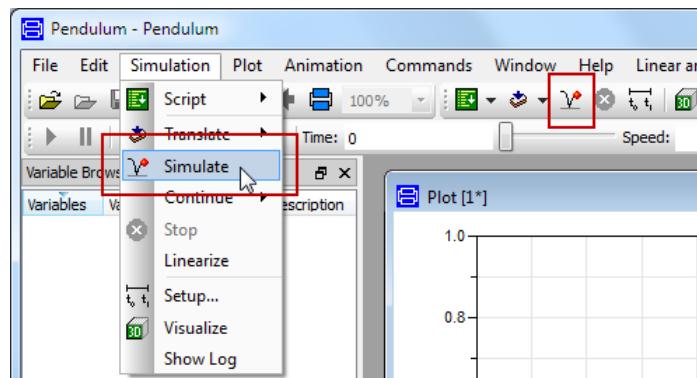


Set the **Stop time** to 10 seconds. Click **OK**.



To run the simulation select **Simulation > Simulate** or click directly on the **Simulate** toolbar button.

**Selecting Simulate in the Simulation menu.**

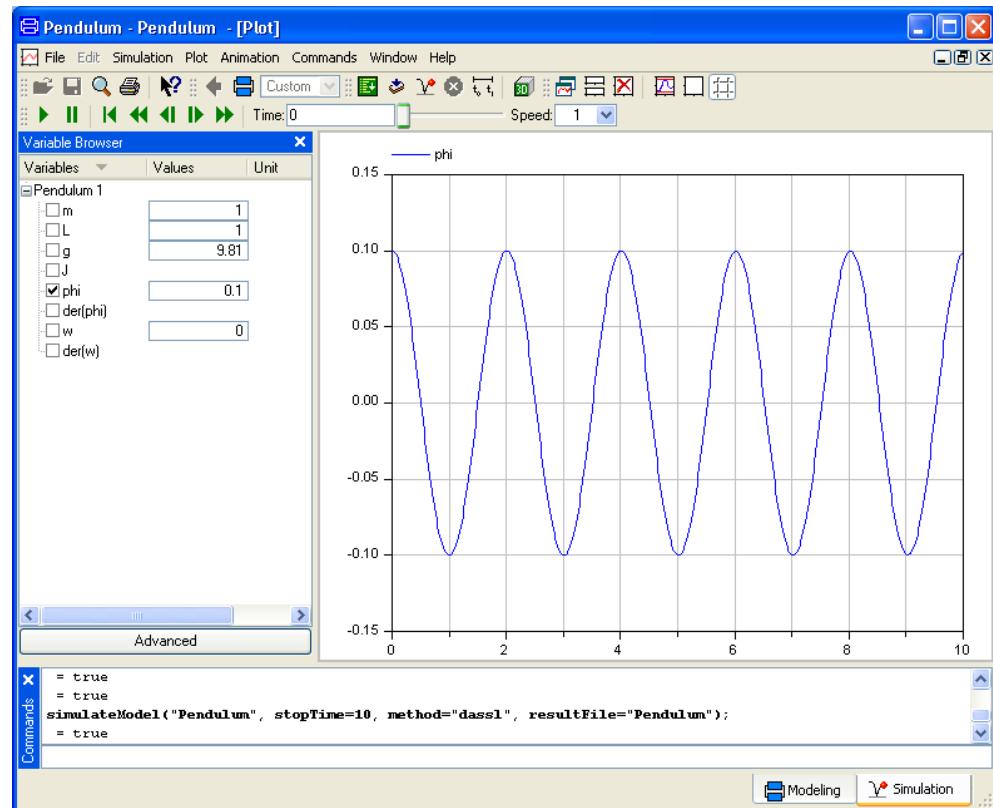


Dymola first translates and manipulates the model and model equations to a form suitable for efficient simulation and then runs the simulation. (You may explicitly invoke translation yourself by selecting **Simulation > Translate** or click on the **Translate** toolbar button.)

You will get a warning that the initial conditions are not fully specified. (The warning can be seen in the Translation tab of the Message window that will pop.) 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 close the Message 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.

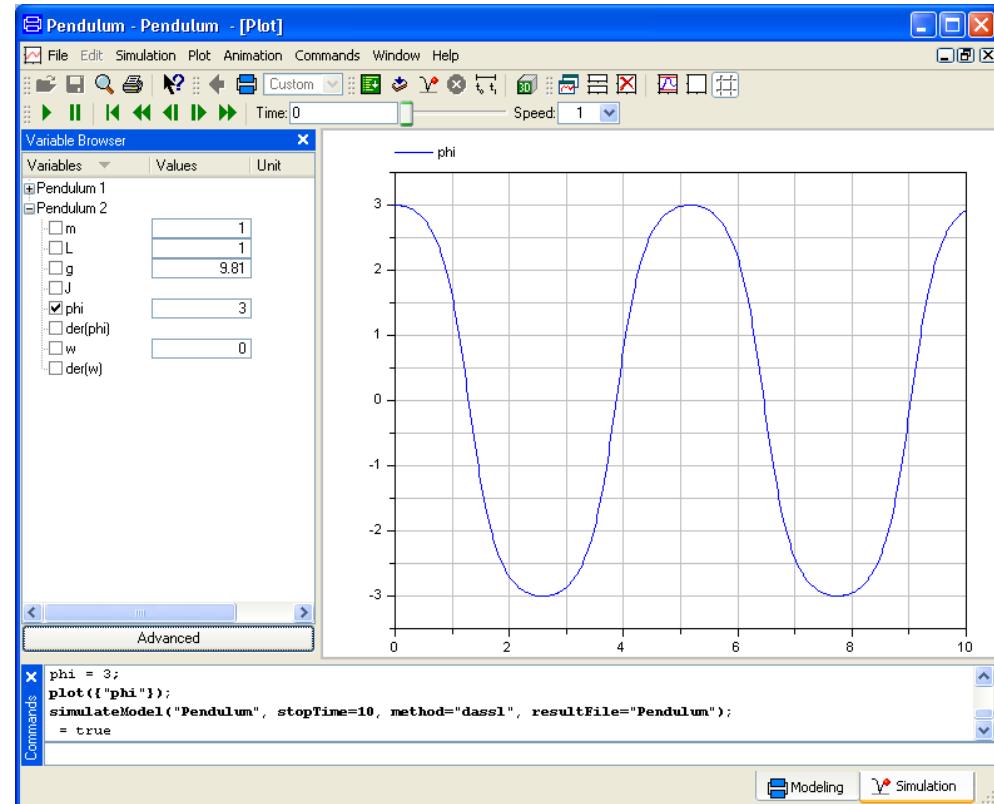
## Plotting the angle.



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  $\phi$  and click on the **Simulate** tool button.

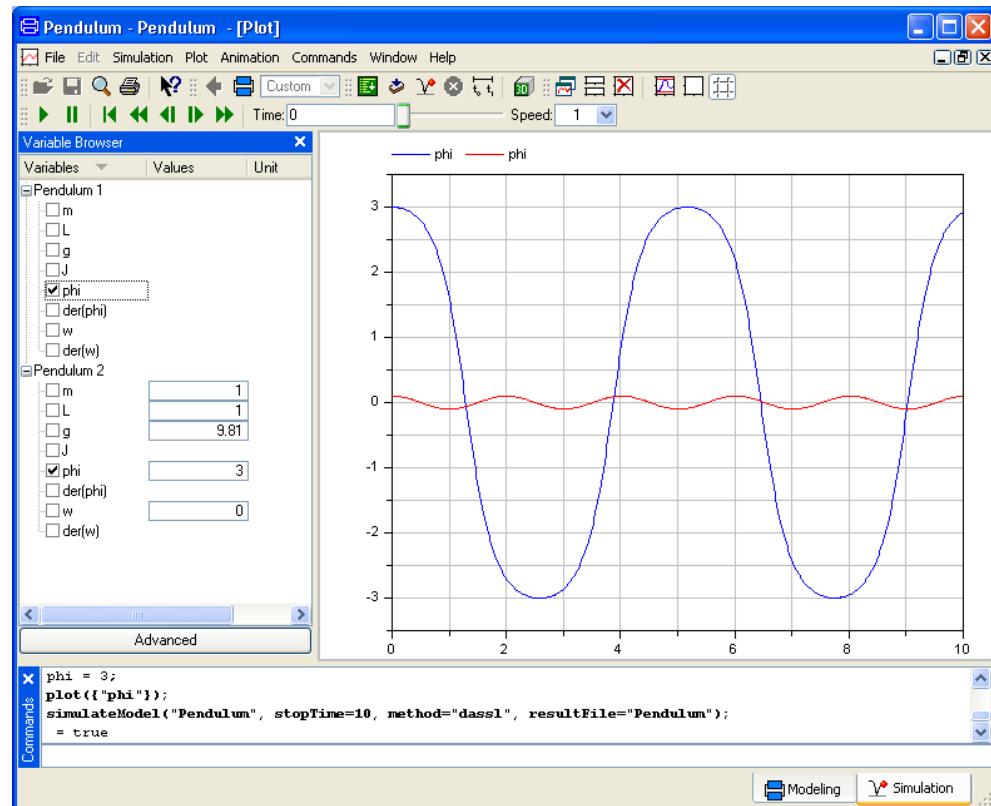
You can also change start values and parameters by typing in the command window; e.g. type  $\phi=3$  (followed by carriage return) for a change of  $\phi$ .

**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** to open Pendulum.mo.)



Press the **Modelica Text** toolbar button (the second rightmost tool button) to show the Modelica text layer.

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:

```

model Pendulum

equation
  der(phi) = w;
  J*der(w) = -m*g*L*sin(phi);
end Pendulum;

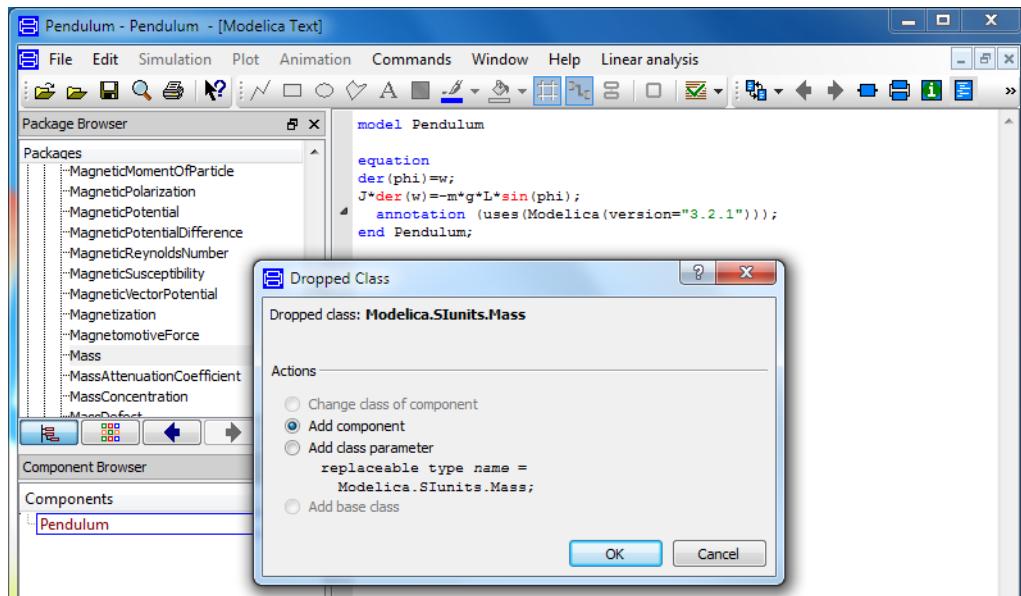
```

Open Modelica.SIunits in the package browser by first expanding the Modelica package by clicking on the + in front of it and then expanding “SIunits” by clicking on the + 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 39 using the physical types available in Modelica.SIunits. The mass is the first one to be declared. To find “Mass” in the package browser, click on the package “SIunits” (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 clicking on the header “Packages” of the package browser that will sort the physical types in alphabetical order.)

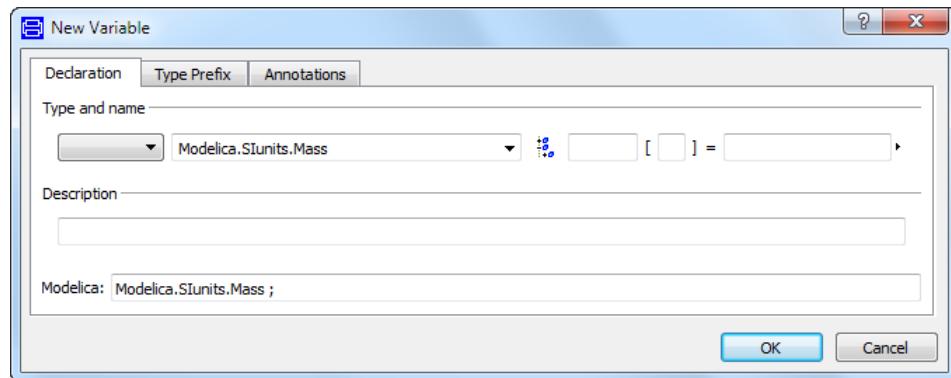
Once “Mass” is found, drag it to the component browser. The following menu will pop up:

#### Adding a physical type.

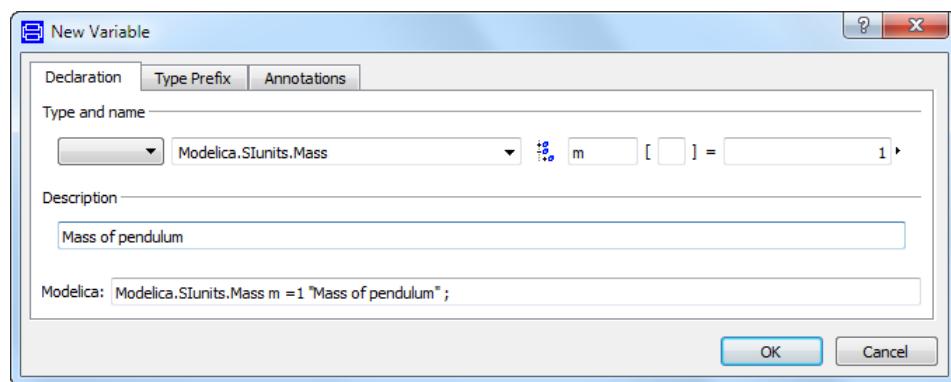


The choice to add a component is pre-selected. Click **OK**. A menu to declare a variable pops up:

#### Declaring a variable.

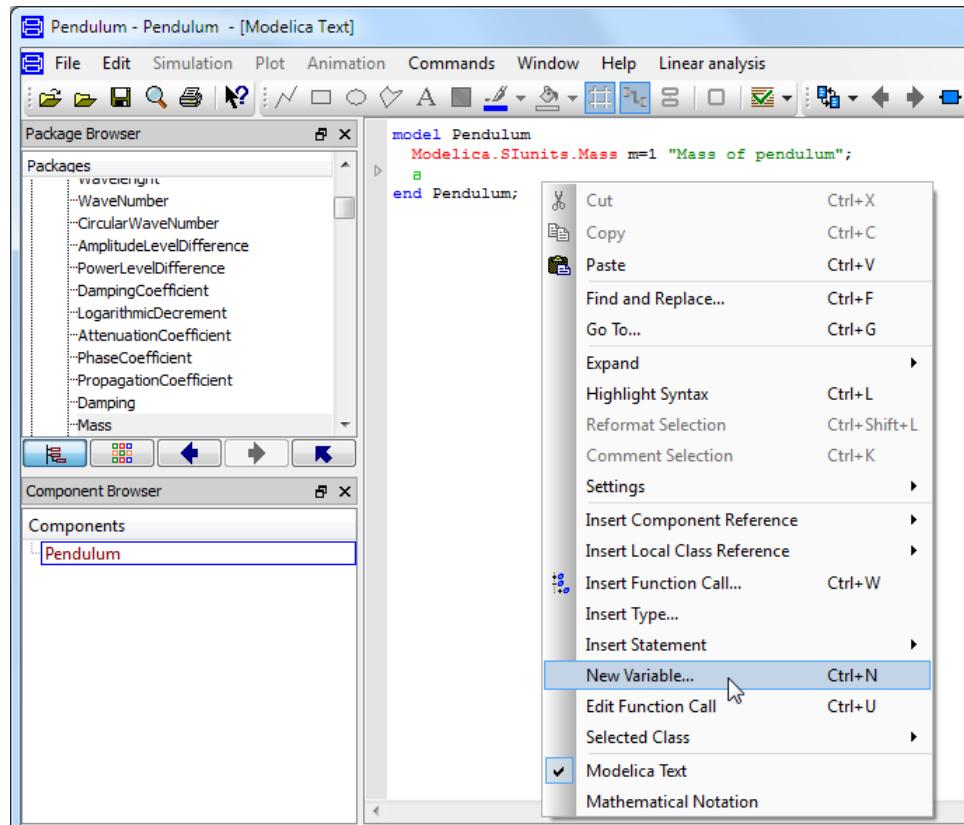


Now we can specify the type prefix (parameter), the name, value and description etc. (By keeping the cursor on top of an input field for a while a tooltip text pops up explaining the field.) Complete the description in the following way:

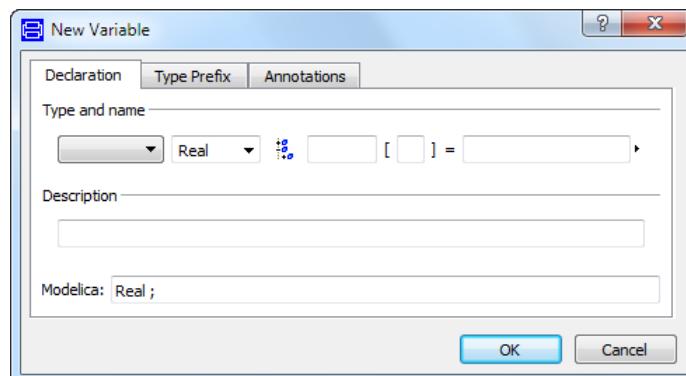


Click **OK** and the text appearing in the bottom row is inserted into the Modelica text window.

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

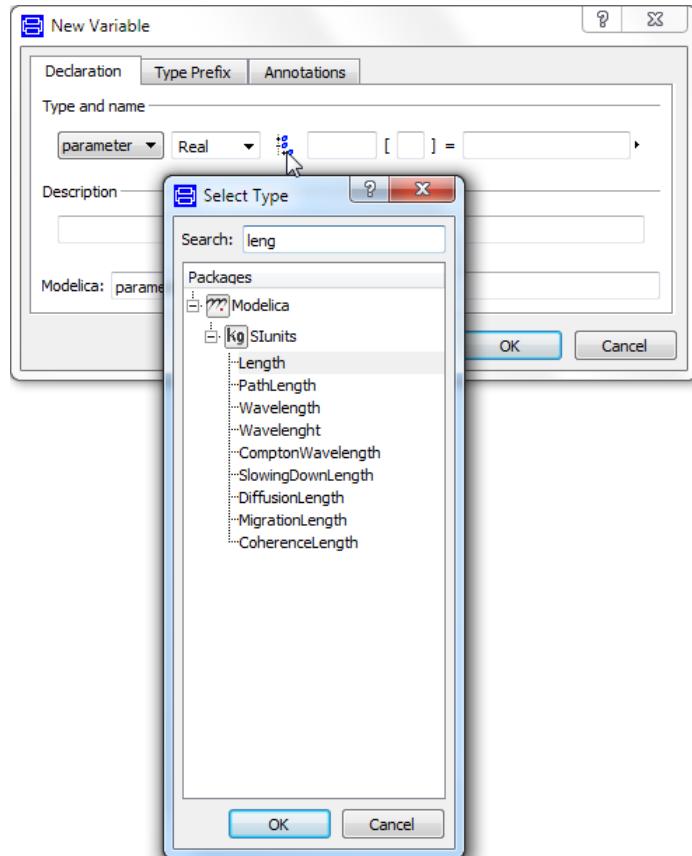


The result is the declaration dialog, just as before:



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:



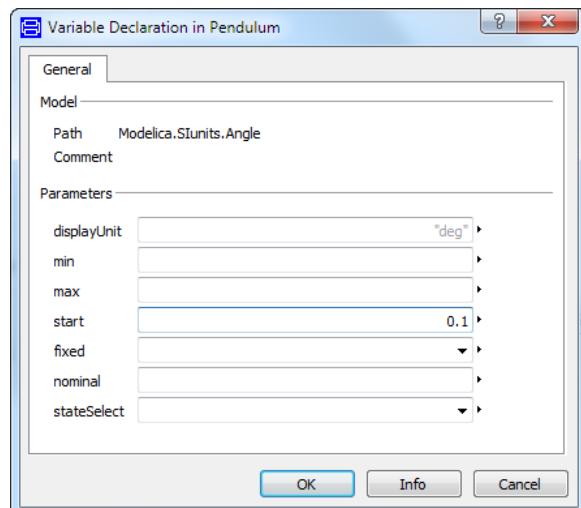
This was exactly what we wanted. It is sufficient to click **OK** to get the type, since it is preselected. (If we wanted another of the displayed types, we could have selected that before clicking **OK**.)

It should be noted that the way above is available also when the Modelica Text layer is not displayed; the command **Edit > Variables > New Variable...** is always available.

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.

```
model Pendulum
  parameter Modelica.SIunits.Mass m=1 "Mass of pendulum";
  parameter Modelica.SIunits.Length L=1 "Length 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;
```

The “+” in the margin to the left of the text and the 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 “+” or on the 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 command **Edit > Variables** can be used to select the variable that should be edited.

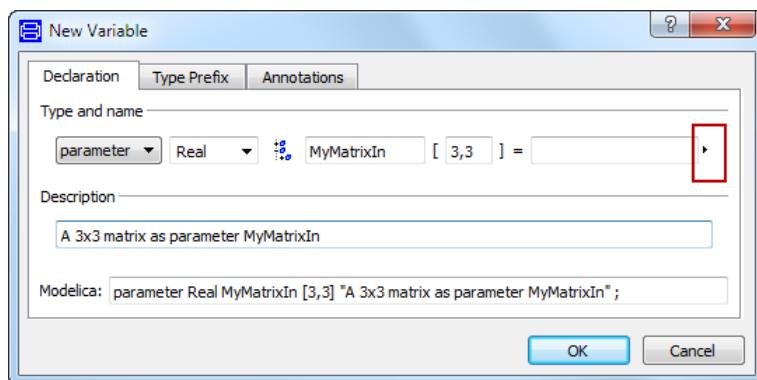
(Another way of inserting pre-defined physical quantities is to recognize that these are types. Types can be inserted in the code by right-clicking to pop the context menu and then selecting **Insert Type**. The menu that pops enables searching by typing in names. The advantage is that all types in all 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 **Edit > Variables** and select the new variable)



Note (outside the example) the ease of defining arrays/matrices using **Edit > 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.

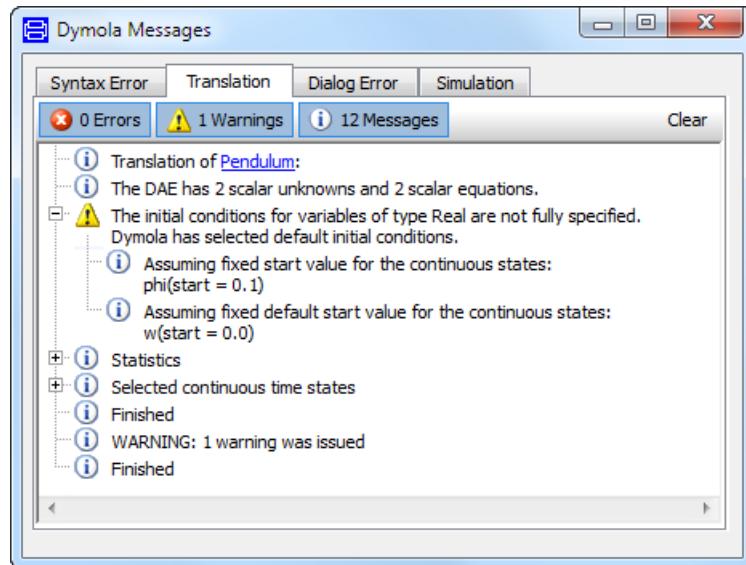
#### **Example of defining a 3 x 3 matrix Real parameter.**



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.

### **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 Message window and scroll to the top the following can be seen (when you have expanded “the warning” node by clicking on the “+” 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 manual “Dymola User Manual Volume 2”, 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:

```

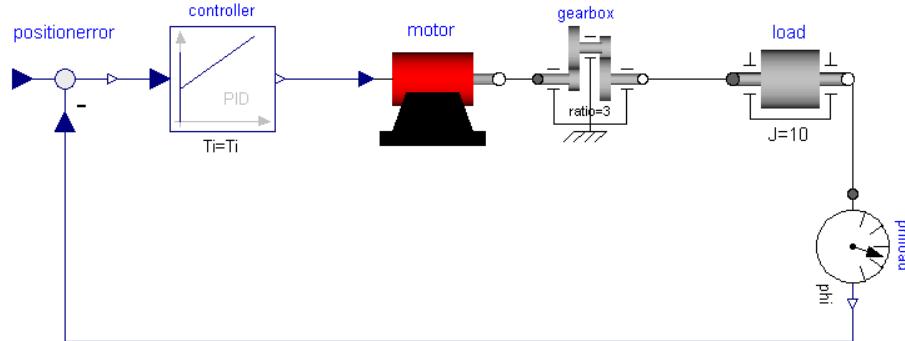
model Pendulum
  parameter Modelica.SIunits.Mass m=1 "Mass of the pendulum";
  parameter Modelica.SIunits.Length L=1 "Length of the pendulum";
  parameter Modelica.SIunits.Acceleration g=9.81 "Gravity of acceleration";
  parameter Modelica.SIunits.MomentOfInertia J=m*L^2 "Moment of inertia";
  Modelica.SIunits.Angle phi(start=0.1, fixed=true) "Pendulum angle";
  Modelica.SIunits.AngularVelocity w(start=0, fixed=true) "Angular velocity";
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.**



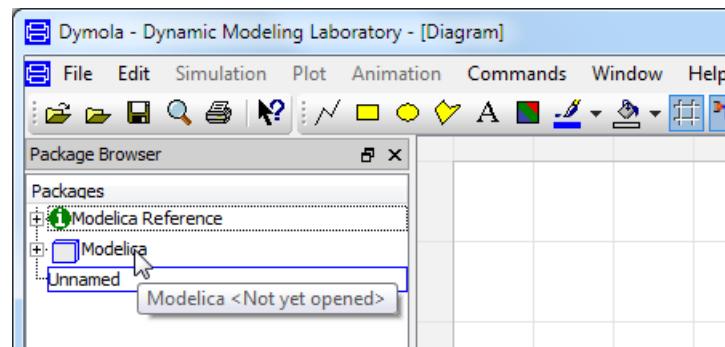
As when building a real system, there are several approaches. One extreme approach is to build the system from scratch. However, it is often a difficult and time-consuming task. Another approach is to investigate if the system already is available on the market or if there is some product that easily 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

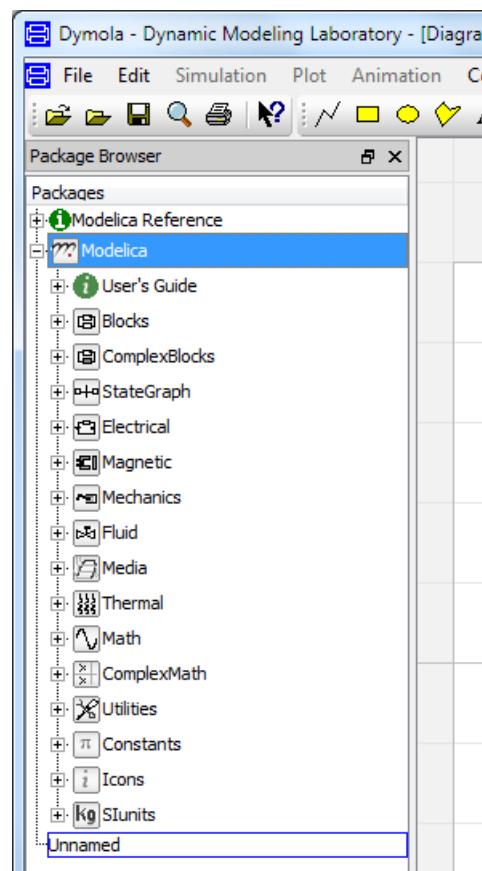
We will now have a look at the Modelica Standard Library to see what is available and how we access the model components and their documentation. To open the library, double-click on Modelica in the Package browser.

**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

- **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.)
- **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 99.
- **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).
- **SIunits** with about 450 type definitions with units, such as Angle, Voltage, and Inertia based on ISO 31-1992.

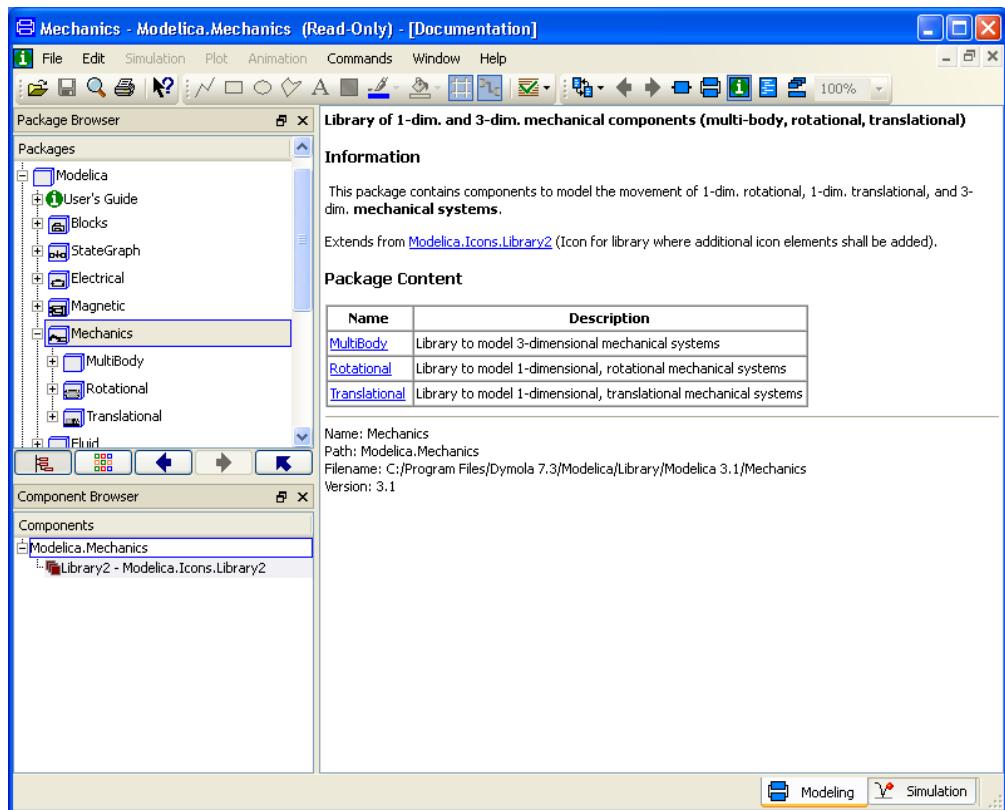
To get documentation for the entire Modelica Standard Library, place the cursor on Modelica, right-click and select **Info**. An information browser is directed to an HTML file containing documentation for Modelica. This documentation has been generated from the Modelica description of the library. There 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 Mechanics in the Package browser. The documentation layer of the library will be shown. (If not, click on the  in the upper toolbar.)

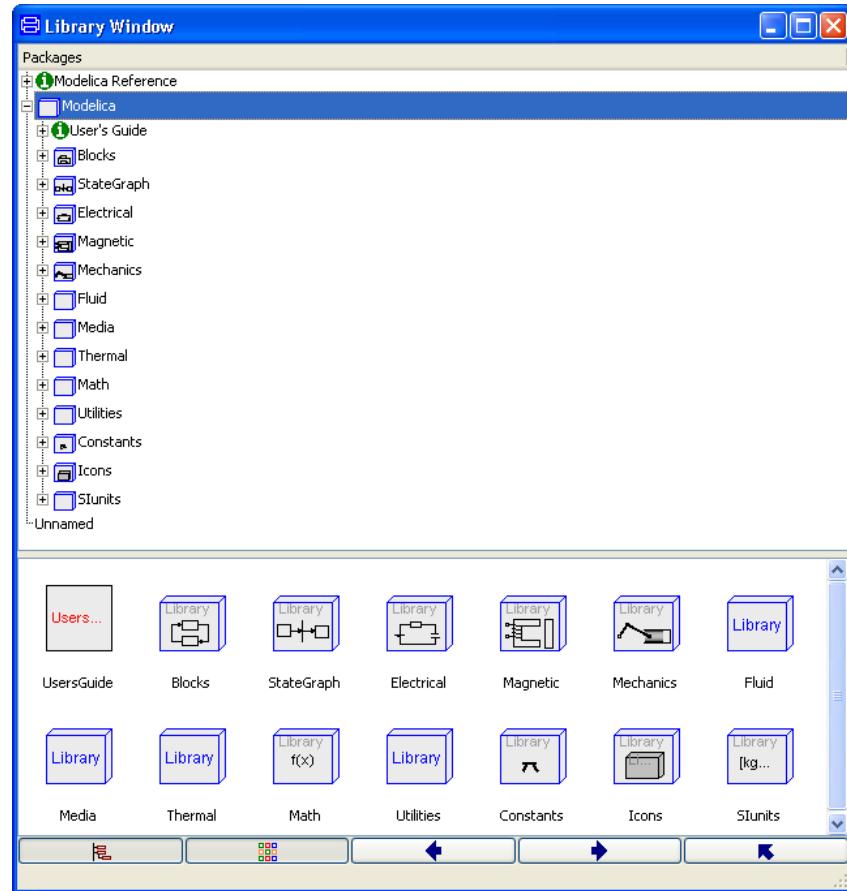
### Opening Modelica.Mechanics.



To get documentation on Modelica.Mechanics (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 toolbar to select **Window > 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:

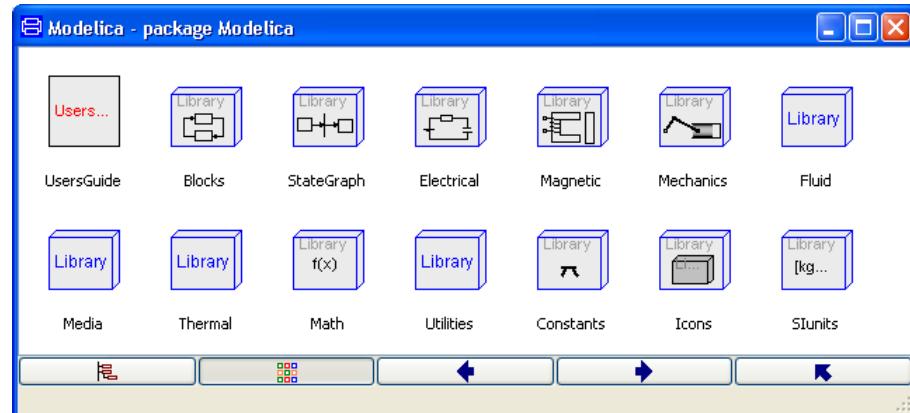
**A library window displaying the Modelica Standard Library.**



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.

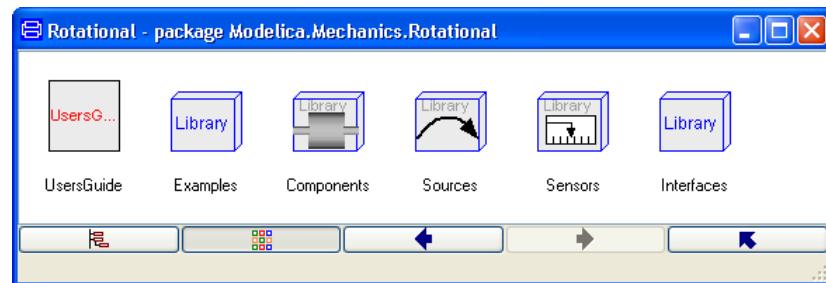
**A library window displaying the components of the Modelica Standard Library.**



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.

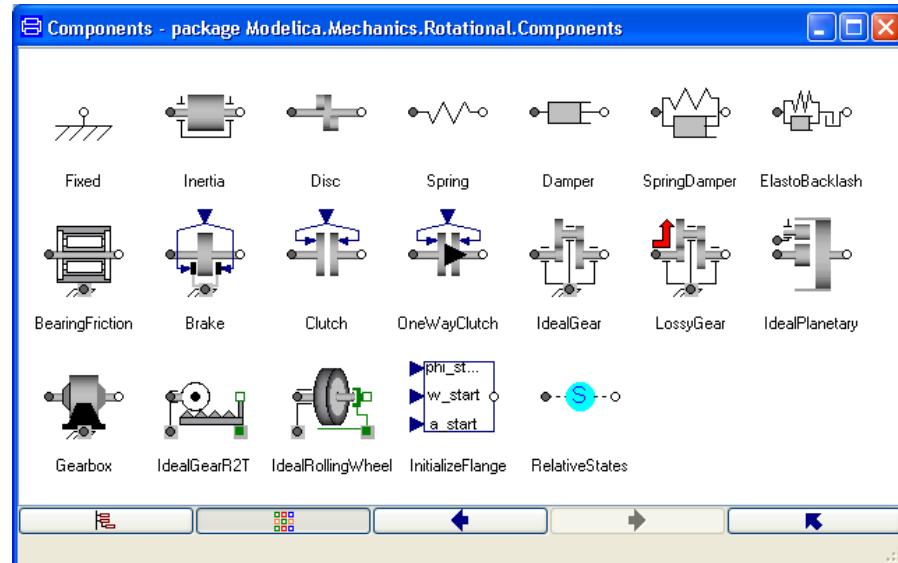
**The rotational mechanics library window.**



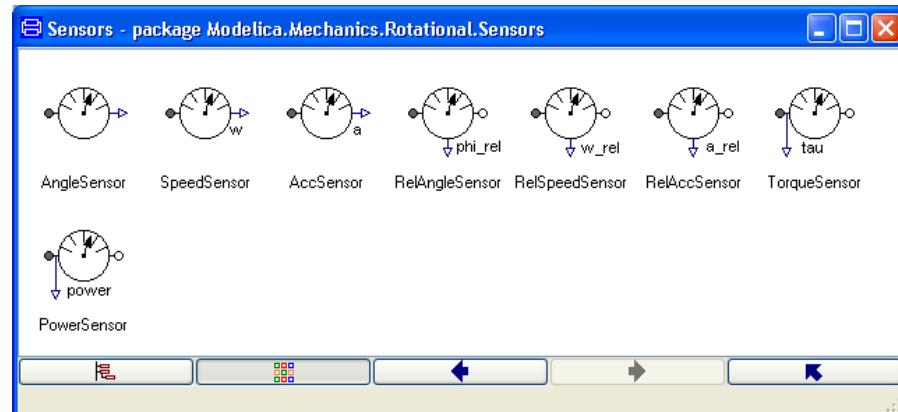
The Info for Modelica.Mechanics.Rotational contains important information on the package content.

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.**



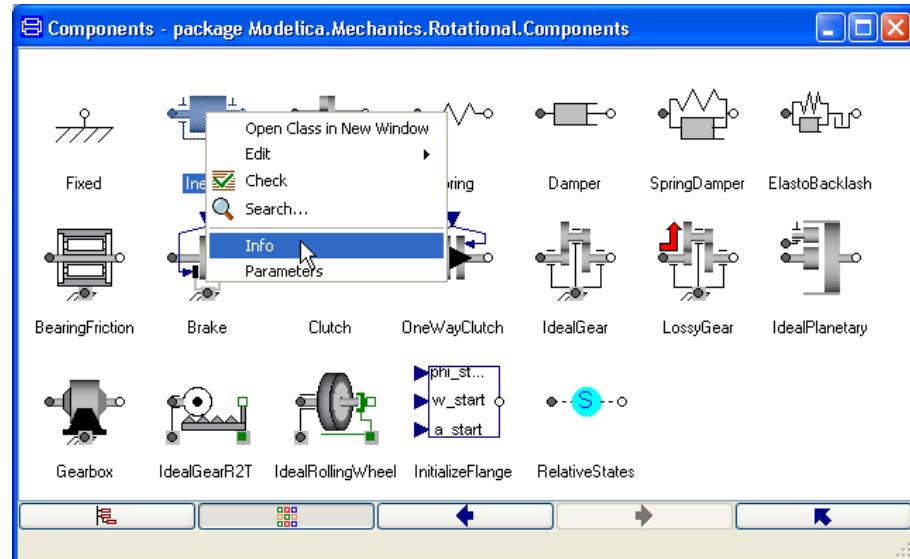
**The Sensors library.**



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

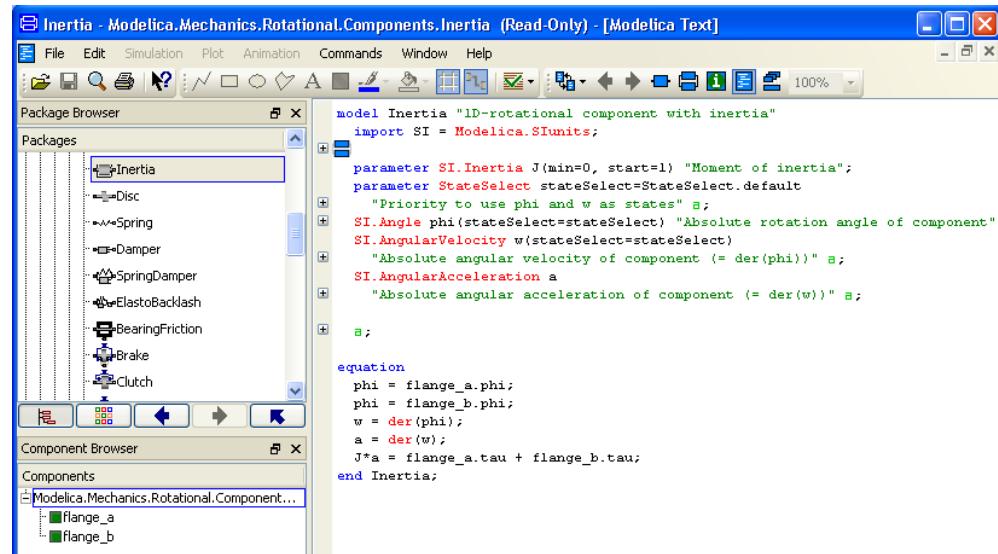
Right-click on Inertia for a context menu. Select **Info** to get documentation for the model.

**The context menu for a component.**



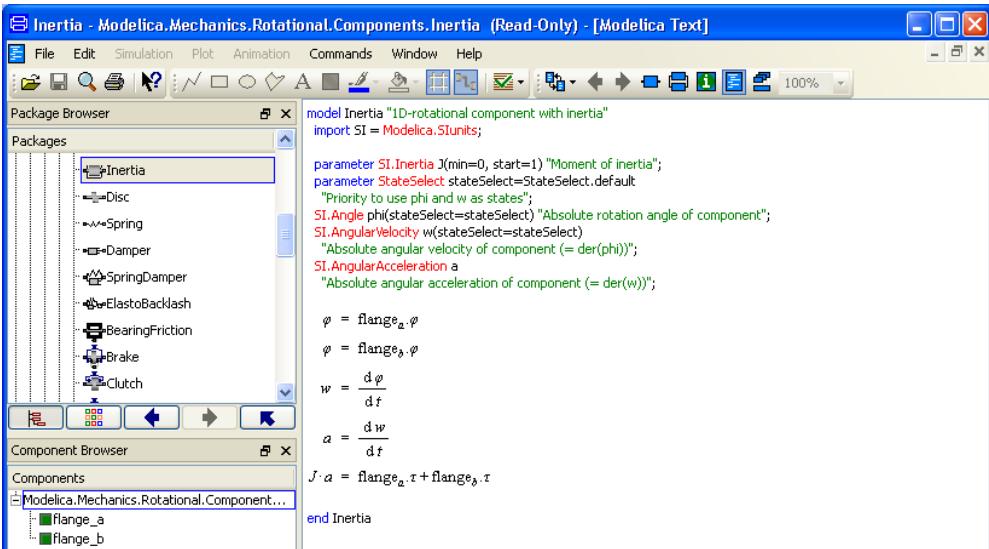
To get a model window for Inertia select **Open Class in New Window** in the same context menu. A window for the model Inertia is created. Switch to the **Modelica Text** representation, where you find Euler's equation as the last equation.

**Mathematical definition of a rotational inertia.**



If wanted, it is possible to look at the content with equations rendered with mathematical notation. Right-click to pop the context menu and select **Use mathematical notation**. The result will be:

## Displaying using mathematical notation.



The screenshot shows the Dymola software interface with the title bar "Inertia - Modelica.Mechanics.Rotational.Components.Inertia (Read-Only) - [Modelica Text]". The main window contains a "Package Browser" and a "Component Browser". The "Package Browser" shows a tree structure with "Inertia" selected. The "Component Browser" shows "flange\_a" and "flange\_b" under "Modelica.Mechanics.Rotational.Component...". The right pane displays the Modelica code for the Inertia component:

```

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(phi))";
SI.AngularAcceleration a
  "Absolute angular acceleration of component (= der(w))";

φ = flange_a.φ
φ̇ = flange_b.φ
w =  $\frac{d\varphi}{dt}$ 
a =  $\frac{dw}{dt}$ 
J·a = flange_a.τ + flange_b.τ
end Inertia

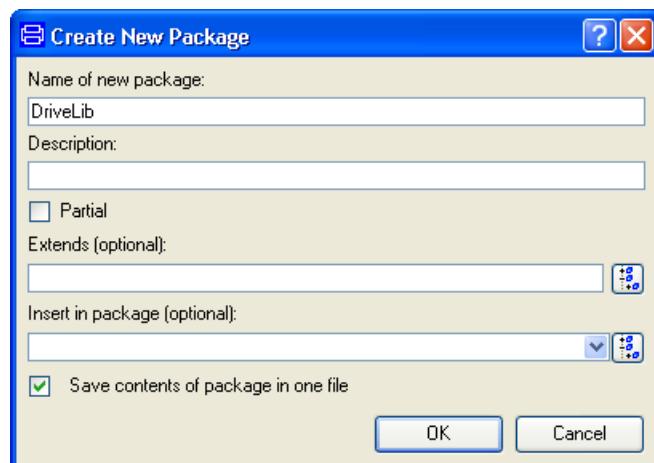
```

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. Go to the Dymola window, and select **File > New... > Package**. This will pop the dialog:

#### Creating a new Modelica package.

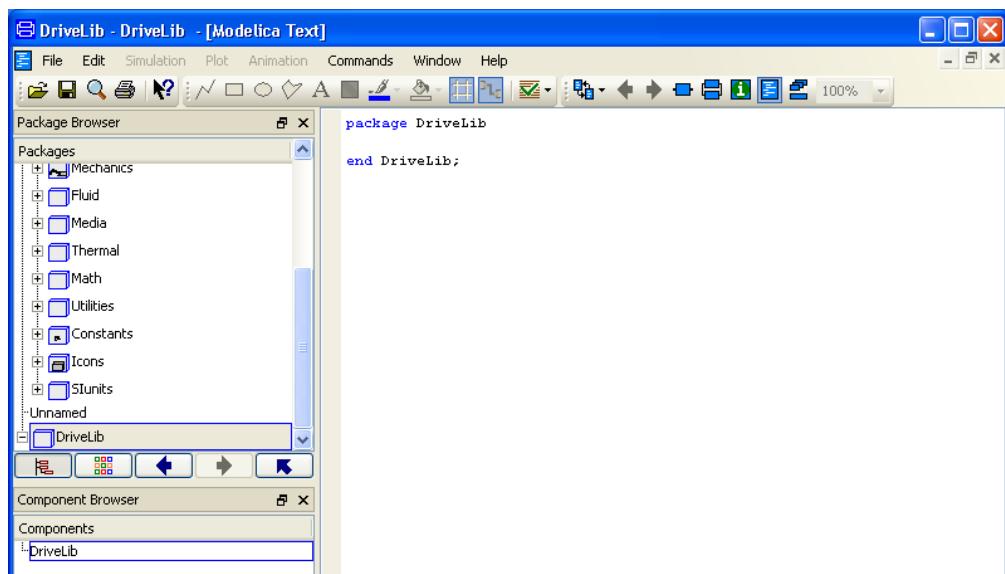


Enter **DriveLib** as the new name of the package and click **OK**, and **Accept** in the information window.

A package DriveLib is created and made visible in the package browser (if scrolled). Select Modelica text to get the Modelica representation, which at this stage just specifies a package with no contents.

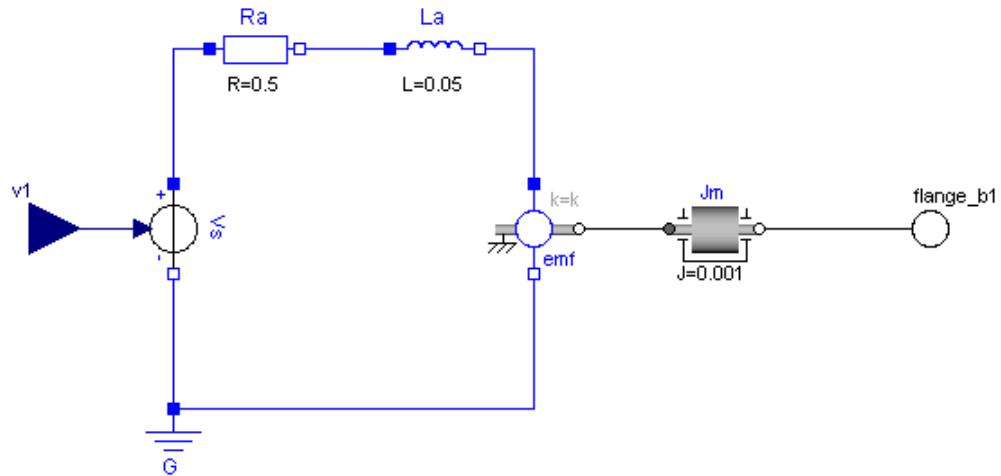
The layer shown in the package created (Modelica Text layer, Diagram layer etc) depends on what layer was shown when creating the package. From the result below it is obvious that the Modelica Text layer was shown when this package was created. The layer shown can easily be changed by buttons in the toolbar in the upper right of the window.

### DriveLib created.



## 2.4.3 Creating a model for an electric DC motor

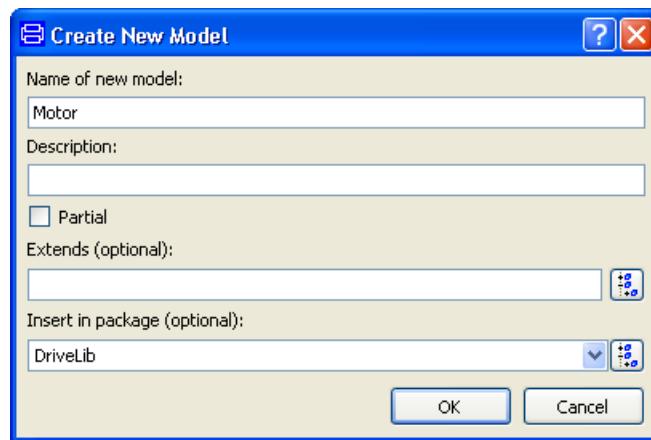
An electrical DC motor.



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.

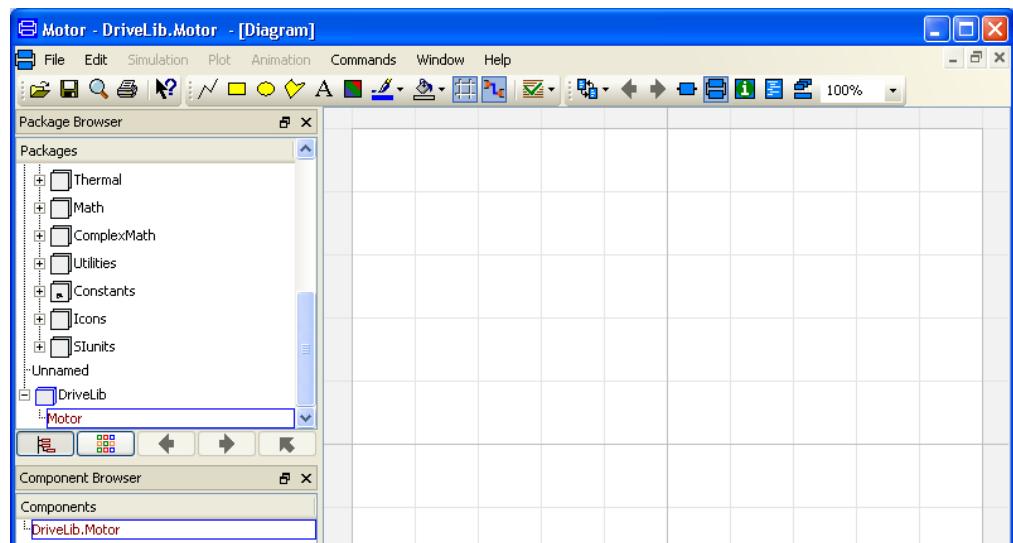
**Inserting Motor in DriveLib.**



Click **OK**.

The package browser shows that DriveLib has a component Motor as desired. The picture below shows the model with the diagram layer displayed (compare with the package created above).

**An empty Motor model.**



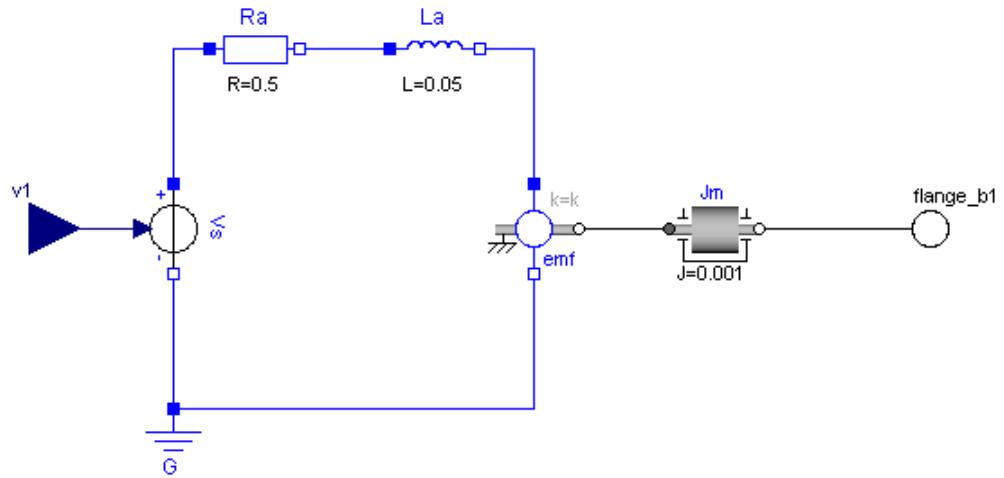
The model window now contains an empty Motor model. The edit window has a gray frame and grid to indicate that the component is not write-protected. It is possible to toggle the grid using the toolbar button.

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 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.

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

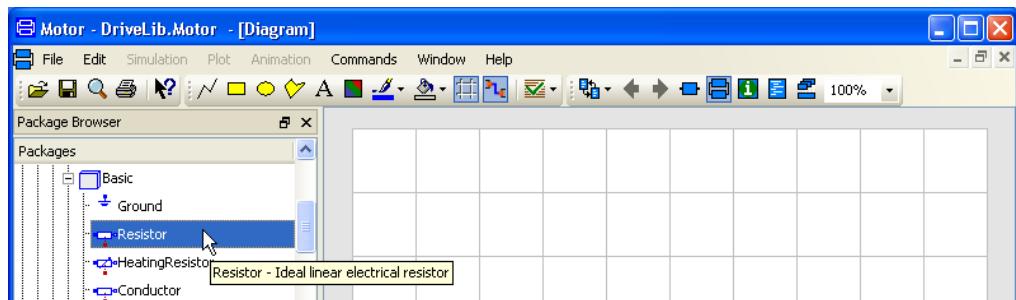
**The finished motor model with all components.**



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, open 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.

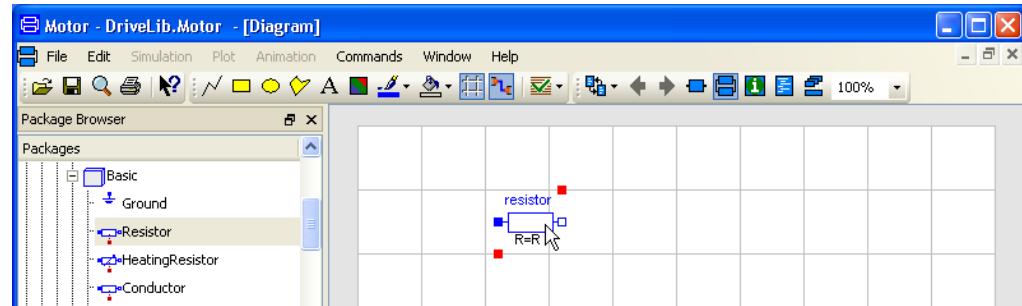
**About to drag a resistor from the package browser.**



(You can also drag from a library window.)

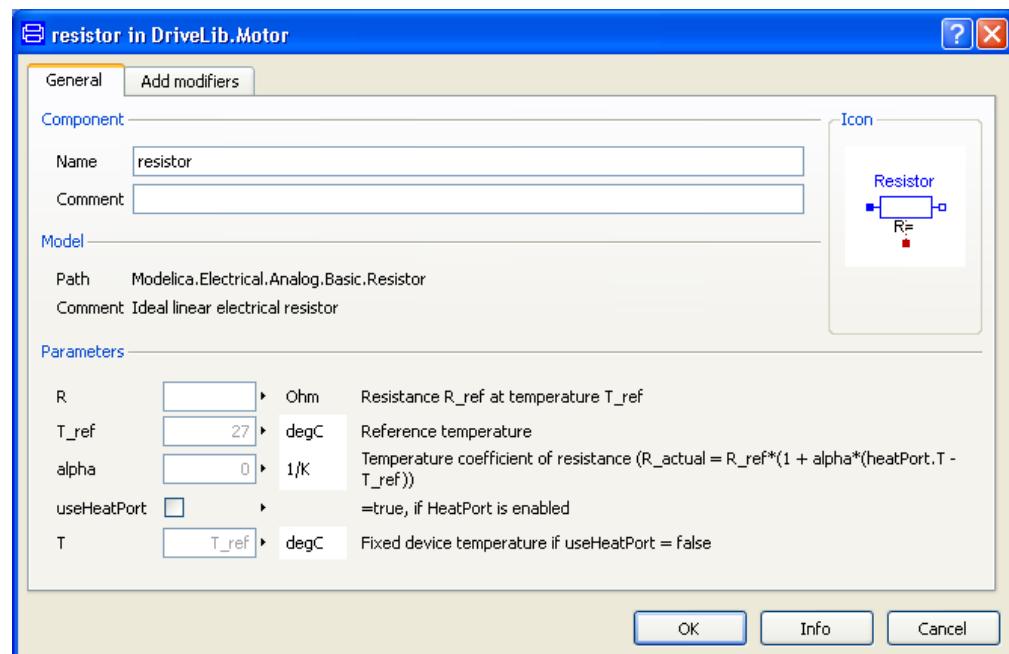
Drag a resistor from Basic to the Motor window and place it as shown above. The component browser displays that Motor has a component Resistor1.

**Inserting a resistor component.**



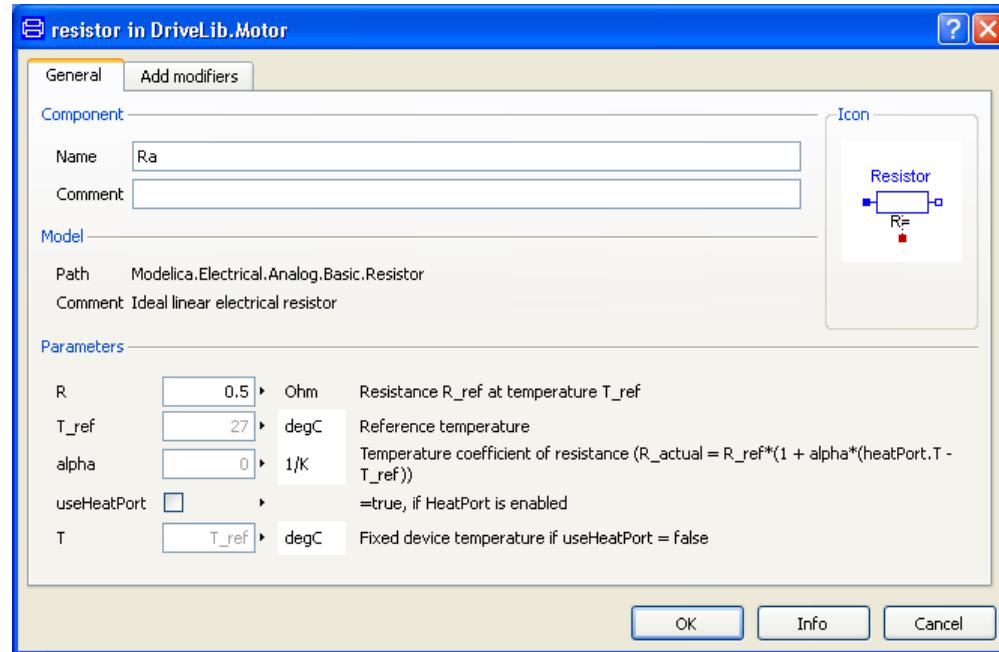
When inserting a component it is given an automatically generated name. The name may be changed in the parameter dialog. (The class name begins with an upper-case character, component instances begins with a lower-case character.) Double-click on the component, to get its parameter dialog. The parameter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

**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, EMF, 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 a library window or package browser to locate it. Select SignalVoltage and drag it to the model window of Motor. Name it Vs. Let Vs be selected and use **Edit > 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 **Edit > 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). Select the connection, right-click and select **Edit > 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.

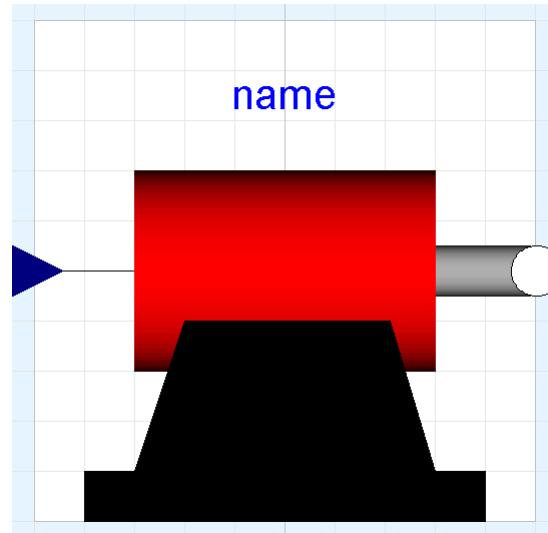
Finally, we have to introduce a signal connector for the voltage control and a flange connector corresponding to the shaft of the motor so the motor can be connected to an environment. We would like to place the icon of the connectors at the border of the grid of the drawing pane, because the icon of 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 toolbar button **Toggle Connect Mode** (when the background is white it is in connect mode).



Click on **Icon** toolbar button to look at the icon layer. You will see the icons for the connectors. 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 the toolbar for editing graphics.

**Toolbar for editing.**

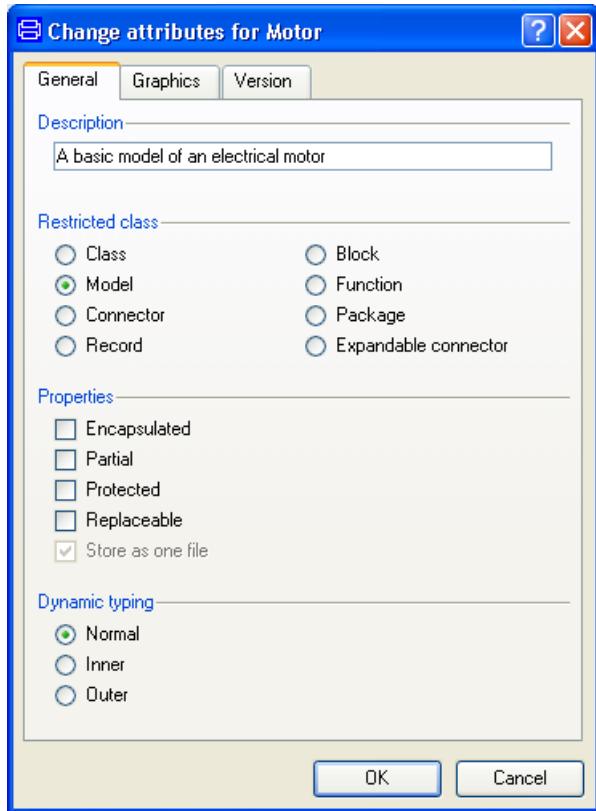


Start by drawing the big red cylinder (shaded rectangle); Click the **Rectangle** button (yellow rectangle) and draw a rectangle. Let it be selected. Click on the arrow to the right of the **Fill style** button. Select **Colors...** and then select a red color. Click **OK**. To select the gradient, click once again on the arrow to the right of the **Fill style** button. 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 A) and lay out 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. Select **Edit > Attributes...** to open the dialog.

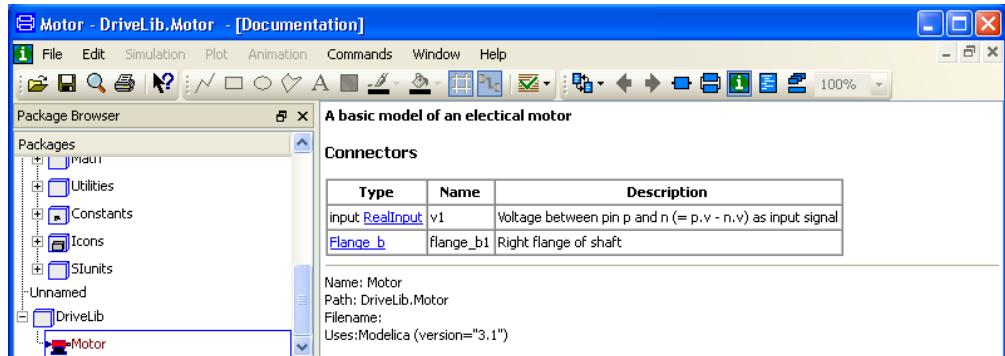
## Model attributes.



Enter a description and click **OK**.



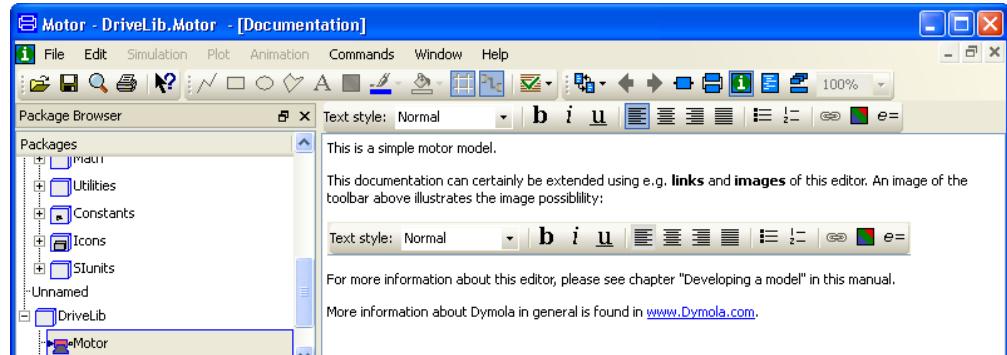
## Documentation View.



To enter a description, put the cursor in the window. Right-click and select **Info editor**. You now have access to a documentation editor for the model/class with text-editing possibilities

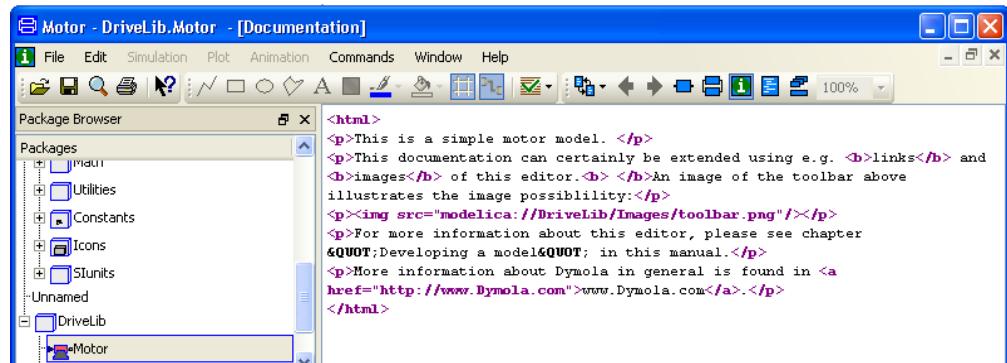
as well as link creation and image insertion. After insertion of some text, an image and a link the result can look like:

#### Documentation editor.



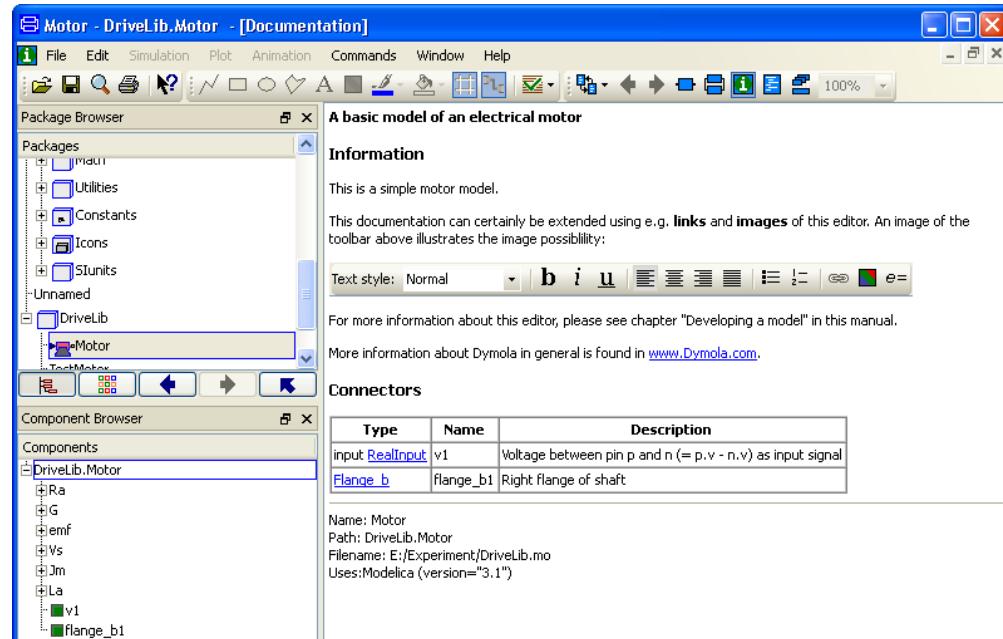
The created source code in html formatting can be viewed selecting **Info source**:

#### The corresponding html source code.



By right-clicking and selecting **Formatted documentation** the result will be shown:

### The final result.



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).

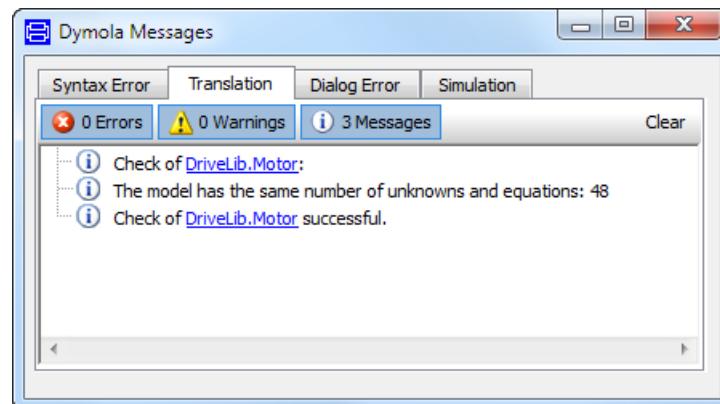
The revision information can be edited in a similar way using **Revisions editor**.

We have now created the model. Save it.

## 2.4.5 Testing the model

It is possible to check the model for syntactic semantic and structural errors. Select **Edit > Check**. Hopefully your model will pass the check and you will get the following message:

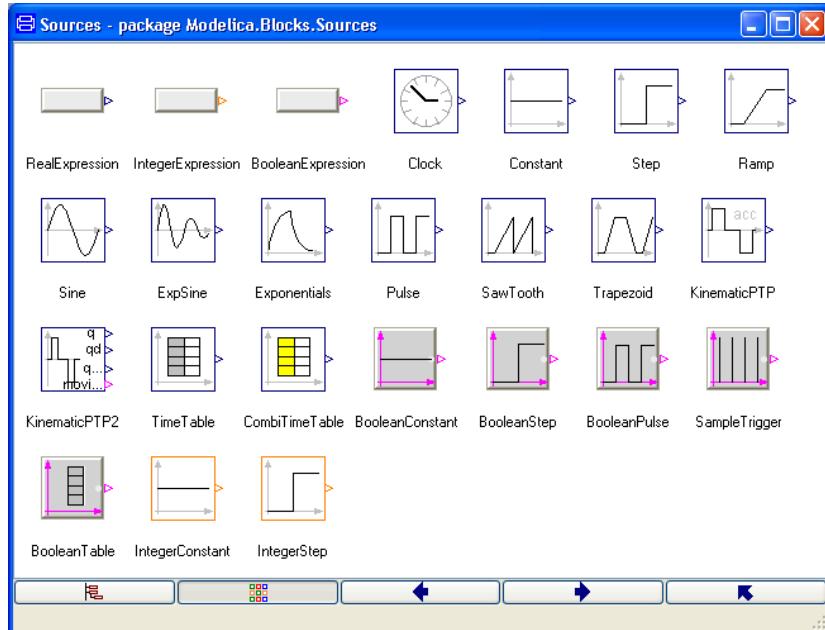
### Checking the model.



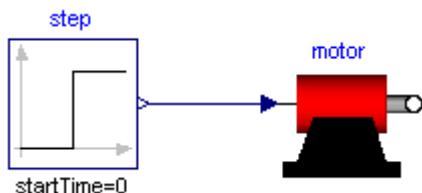
The connector inPort defines the voltage reference, and should be defined for the complete model, but is viewed as a known input to the model.

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.

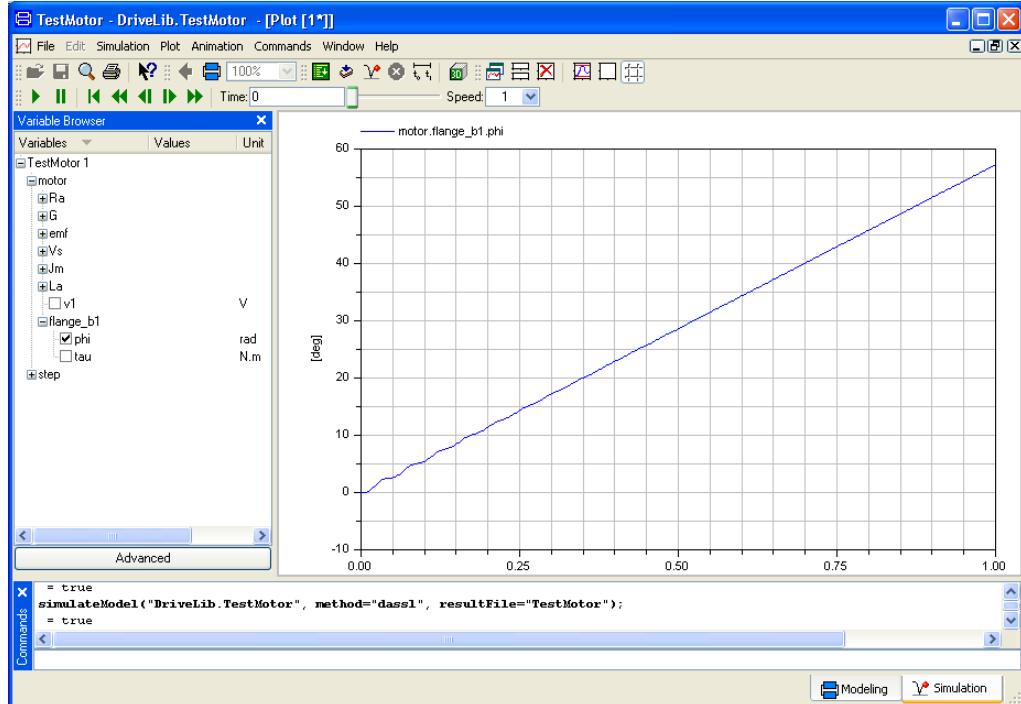


Now it is time to simulate. Click on the tab for **Simulation**. This will change the *mode* from Modeling mode to Simulation mode. To simulate *the model* click on the toolbar button **Simulate**. (Please note the difference.)

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.

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 + sign. Open the flange\_b and tick phi.

### Angular position.



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. The toolbar button **Erase Curves** (white rectangle) erases all curves in the active diagram. It is also possible to have several diagrams. To get a new diagram, select **Plot > New Diagram** or click on the toolbar button. The new diagram becomes active. Selecting a diagram makes it active. (Selecting **Plot > Delete Diagram** removes the diagram.)

(In addition to the above, there are numerous options in the plot window, e.g. you can

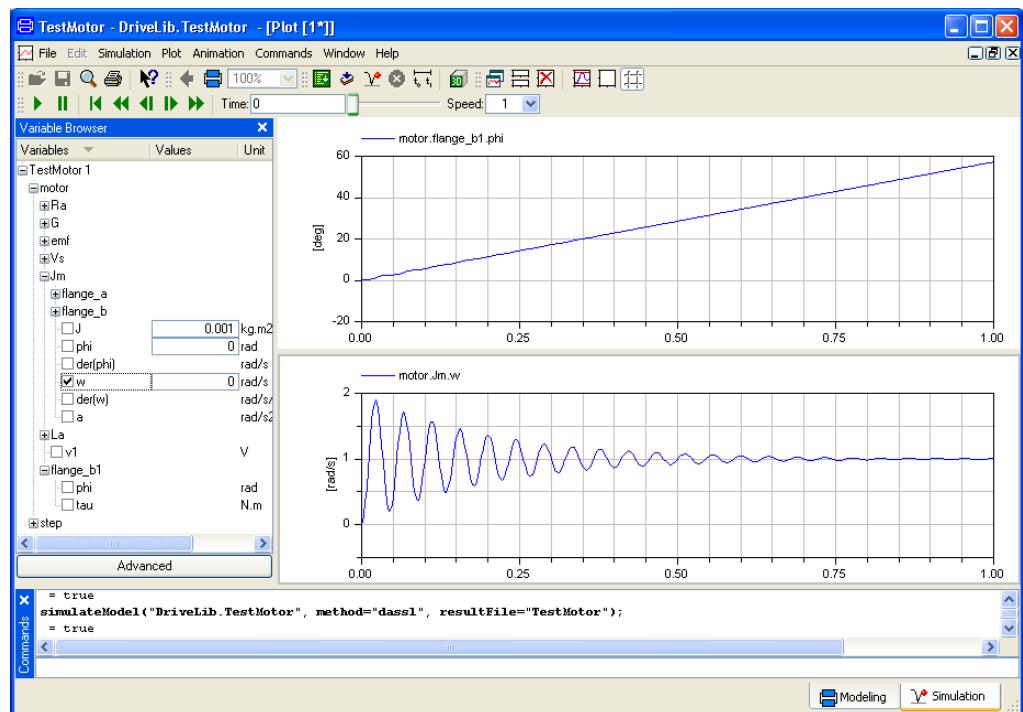
- Display dynamic tooltips (and copy the values to clipboard if needed).
- Display tooltip for the legend to see more information about the corresponding signal.
- Display tooltip for the x-axis and the y-axis.

- Moving and zooming (one way of zooming is to just drag a rectangle over the area that is of interest).
- Change time unit on x-axis.
- Change the display unit of a signal.
- Display signal operators.
- Plot general expressions.
- 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.
- Plot parametric curves.
- Display Boolean and enumeration signals.
- Change color, line/marker style and thickness of a signal.
- 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.)

For more information about these options, please see chapter “Simulating a model”.)

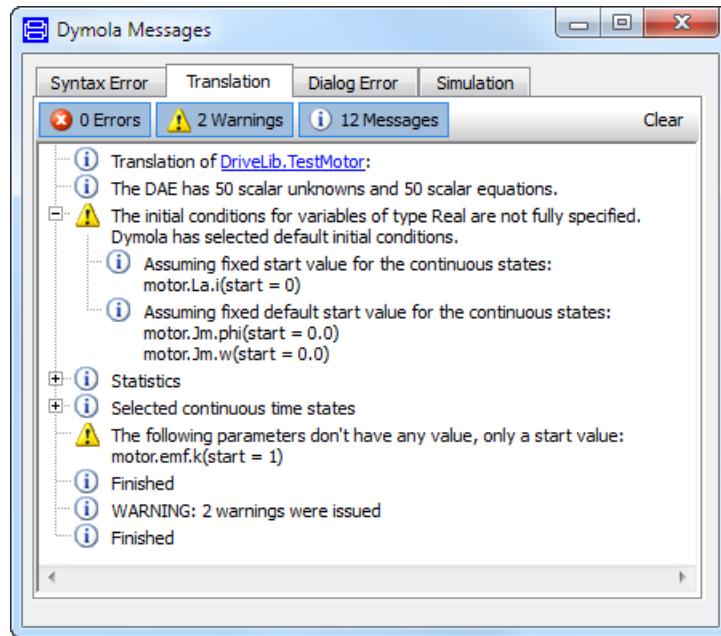
Using the option of adding a new diagram and ticking `motor.Jm.w` the result shown below is obtained.

## Angular velocity.



## 2.4.6 Handling the warnings

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

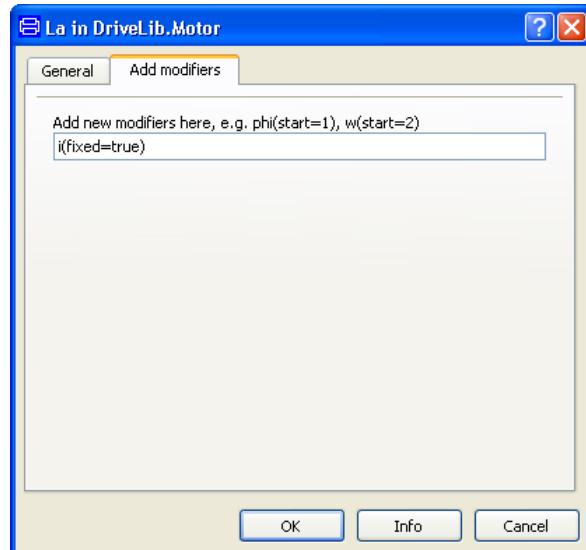


Two types of warnings are present for this example; a warning that initial conditions are not fully specified (at the top of the tab) and a warning that a parameter does not have any value, only a start value (at the bottom of the tab).

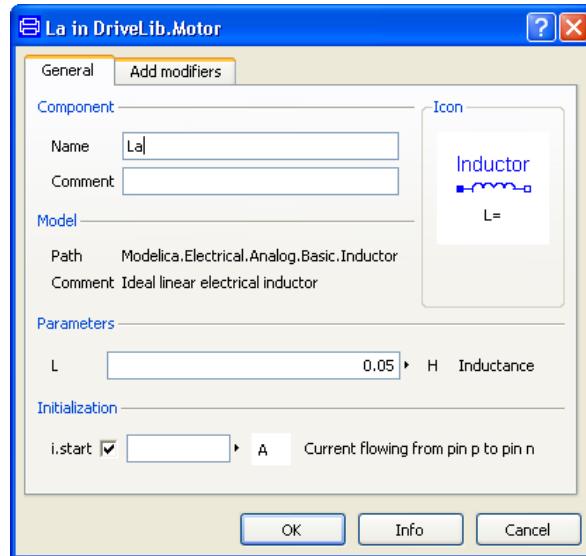
The first type of warning has been described previously, see section “Handling of warnings” on page 52. The difference here is that we can use the graphical user interface to set the variables to fixed; we do not have to enter code.

Looking at the warnings, they all have to do with components inside the Motor model. By double-clicking on the Motor in the package browser we return to this model.

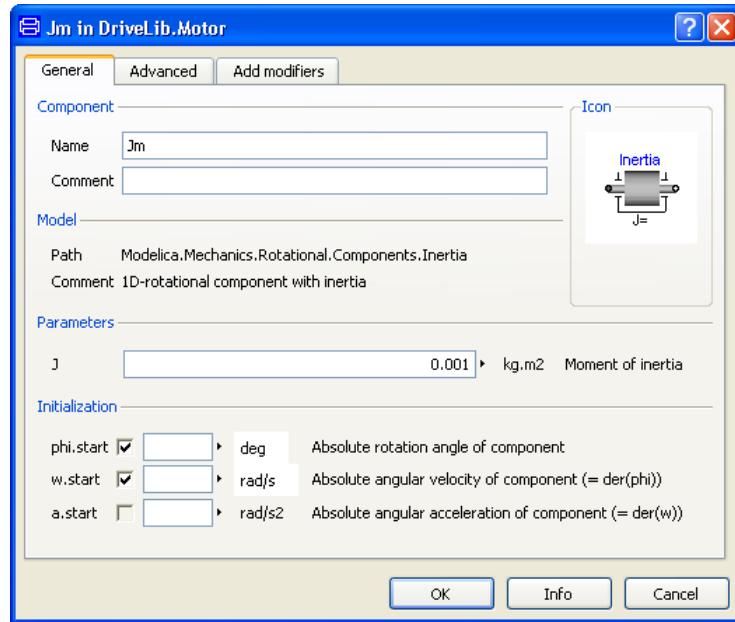
The first variable of the upper warning section is located in the inductance component La. Double-clicking on this component will however not display any Initialization section. But since the variable is known, we can click on the tab **Add modifiers** and enter the text `i(fixed=true)` and press **OK**.



When having clicked **OK**, the General tab will change to present the new initialization information. Double-clicking again on the inductance component will result in:



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



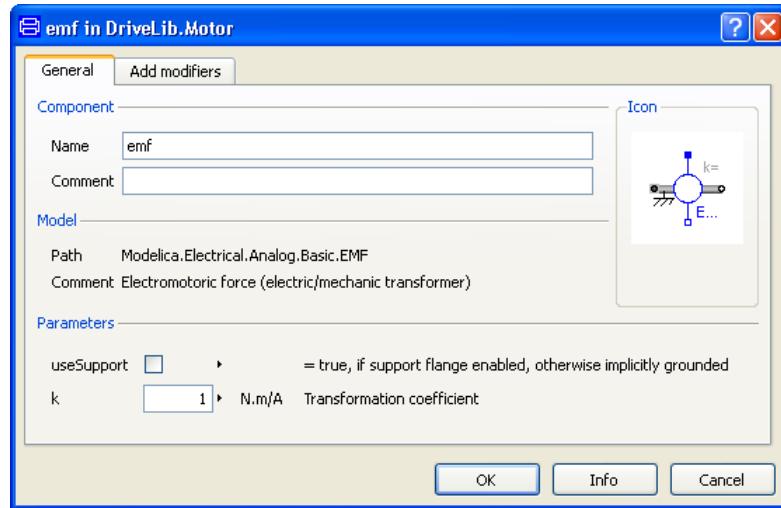
Click **OK** to confirm the changes.

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

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 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

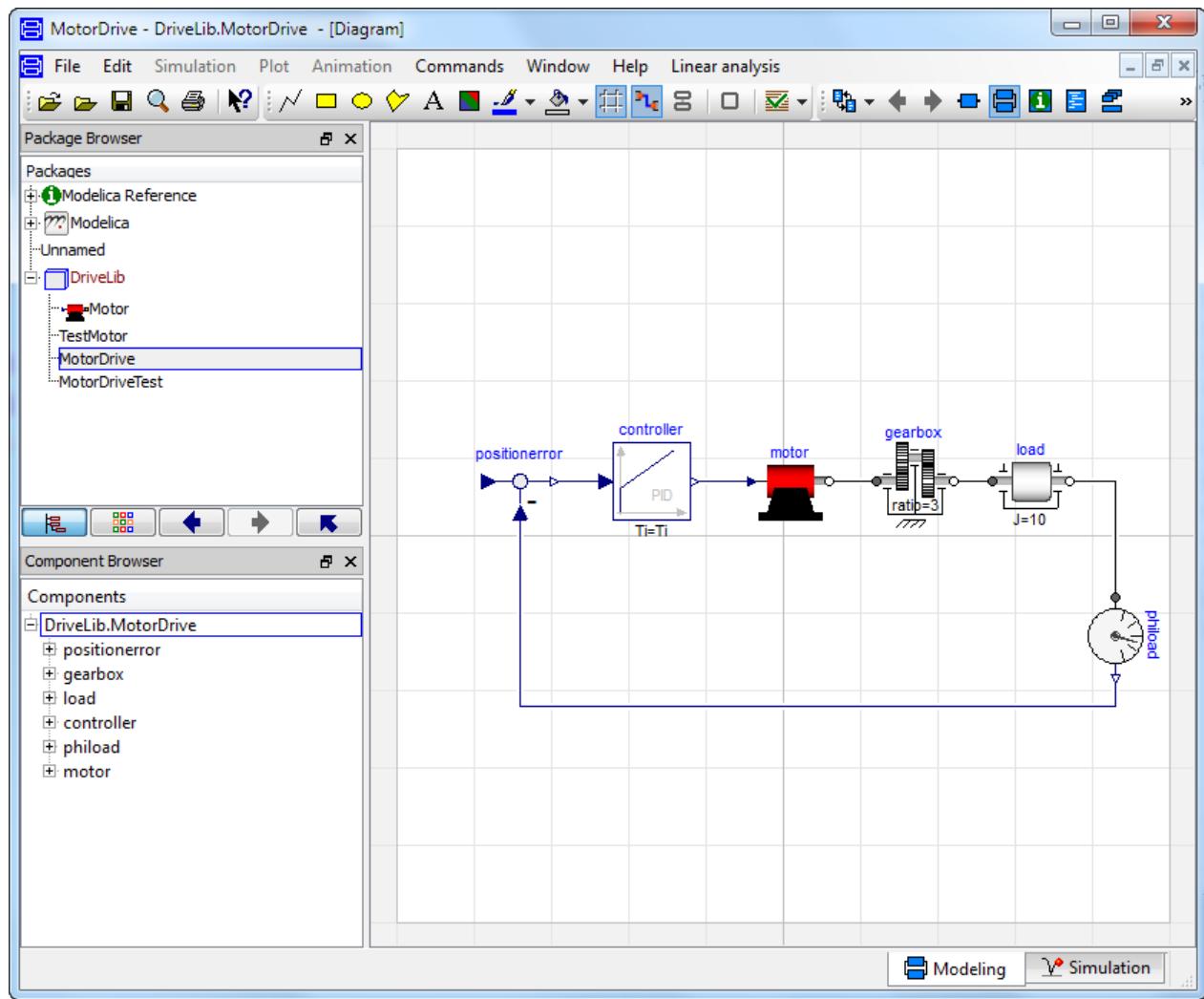
The task was to build a model for a motor drive and it ought now to be a simple task to complete the model. We will just give a few hints. Note that the full names of the components are given in the component browser at the lower left.

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. 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.



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 and select **Edit > 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. Drag a component Step from Modelica.Blocks.Sources to the edit window and connect it. Save the model.

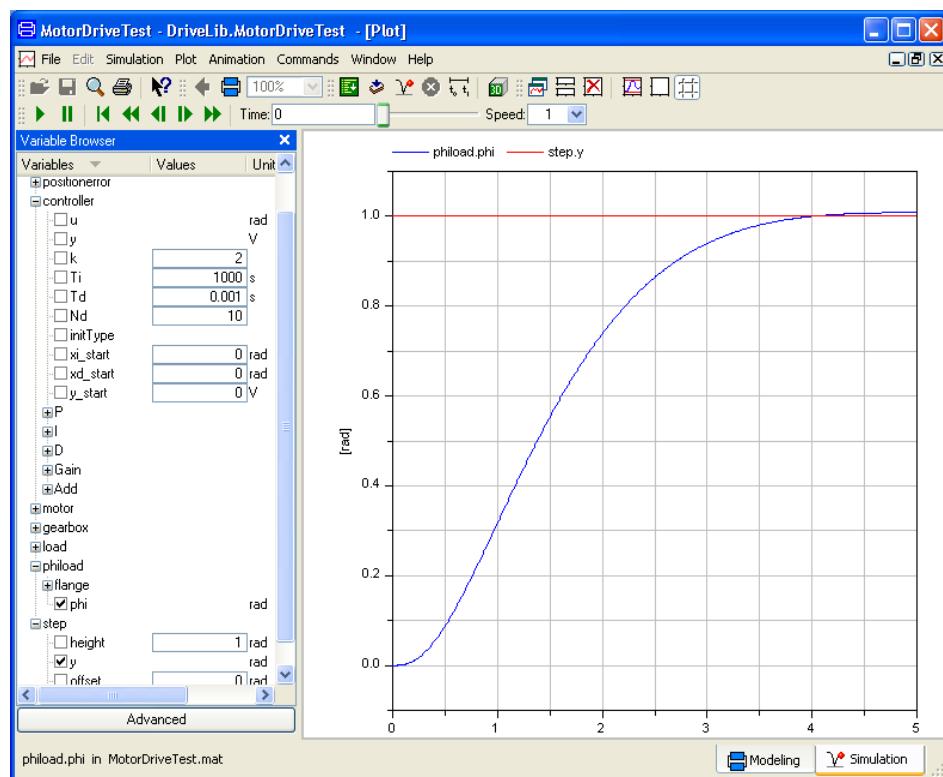
(It is valuable to give some thoughts comparing “extend” to “copy/paste+add”. The former strategy allows a far more clean structure, where an object can be reused in a number of locations. If that object is further refined, all objects extended from that one will gain from that refinement.)

A model can be used for different tasks. One is tuning the controller manually. Click on the tab for **Simulation**. Translate the model MotorDriveTest by clicking on the **Translate** button. (Also delete one of the plot windows and delete the previous curve.) The PID controller has four parameters: k, Ti, Td and Nd.

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

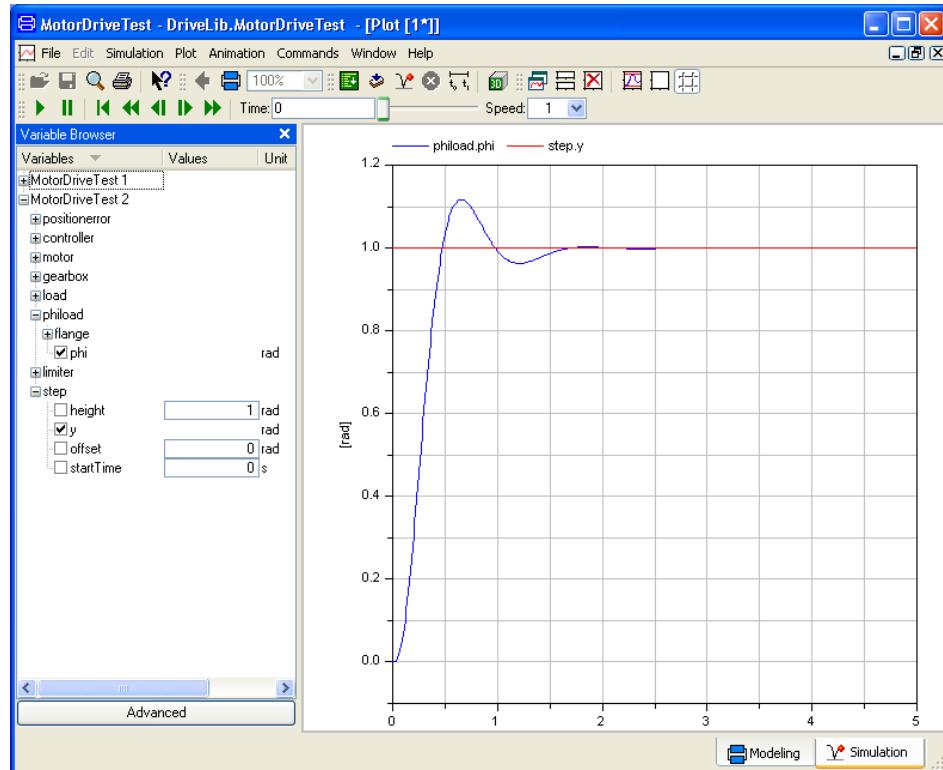
One way to tune a PID-controller is by first disabling the integrator and derivative parts. This is achieved by setting a large value for Ti and a small value for Td. Set k=2, Ti=1000, and Td=0.001 and simulate for 5 seconds. (Use **Simulation > Setup...** to change the stop time.) As can be observed, 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. The result will be:

#### Tuning the controller.



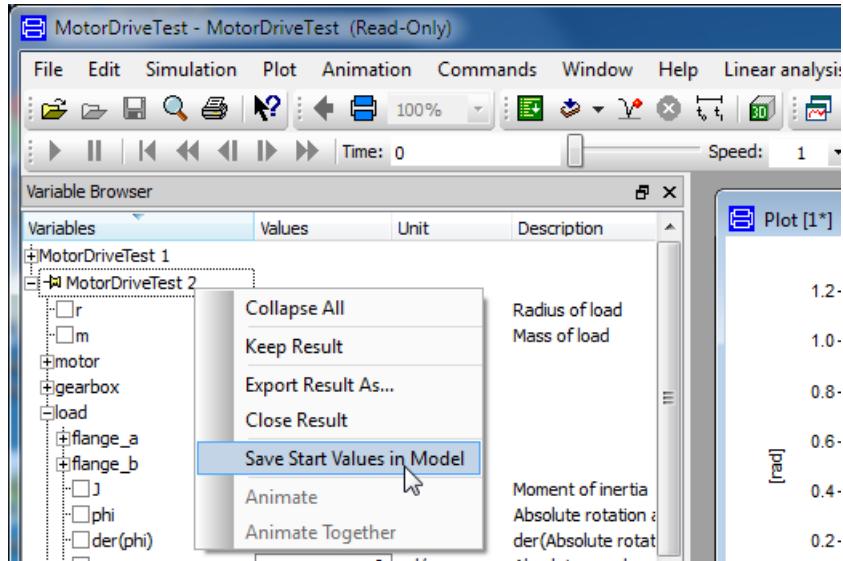
The interested reader may next proceed to tune the controller parameters to obtain a nice step response. For example, aim at a rise time around 0.4 seconds, a settling time around 1 second, maximum overshoot of 10 percent, and a maximum control signal magnitude of 100. Enforce the last constraint by 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 might be something like:

### Tuning result.



### Saving changed start values to the model

Having tuned values in simulation mode 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 (on purpose the wanted values are hidden under the menu, not to destroy the challenge of finding better values in previous section):



### Handling the warnings

There will be some warnings when translating. The principal handling of these warning has been dealt with previously. The only thing that ought to be mentioned here is that in order to make the variable `controller.D.x(start=controller.D.x_start)` fixed, you have to double click on the MotorDrive, 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 x is available you have to select the **Add modifier** tab, enter `x(fixed=true)` and press **OK**.

The rest of the warnings can be handled like described in a previous section.

### 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

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

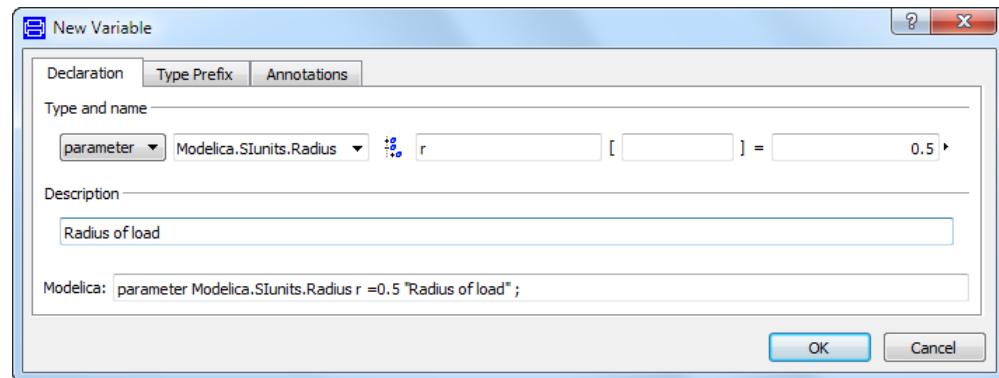


We need to declare m and r in MotorDrive. Open MotorDrive. Activate the Modelica Text representation; press the **Modelica Text** toolbar button (the second rightmost tool button).

The parameters and variables are more than real numbers. They are physical quantities. The Modelica standard library provides type declarations for many physical quantities. Select Open Modelica.SIunits 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 enter r and 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.

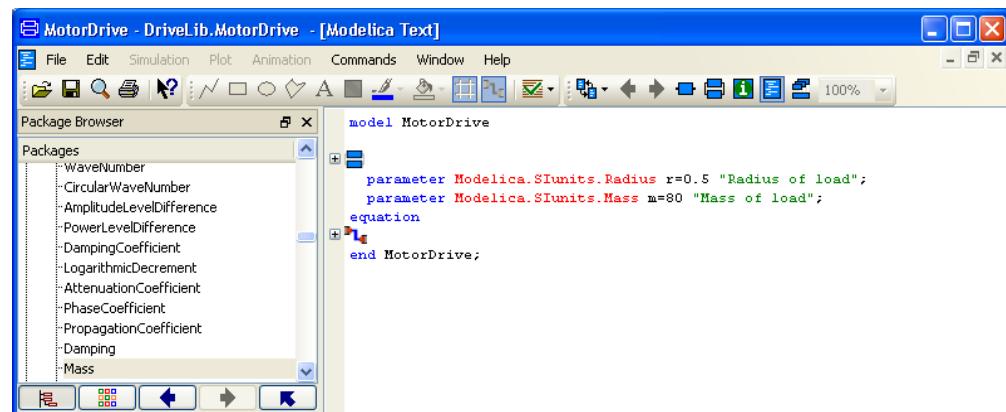
When you have found Radius, drag it to the component browser below. The choice to add a component is pre-selected. Click **OK**. A menu to declare a variable pops up. Complete the declaration (do not forget to change to a parameter):

### 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.

### Parameter declarations added to motor drive.



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

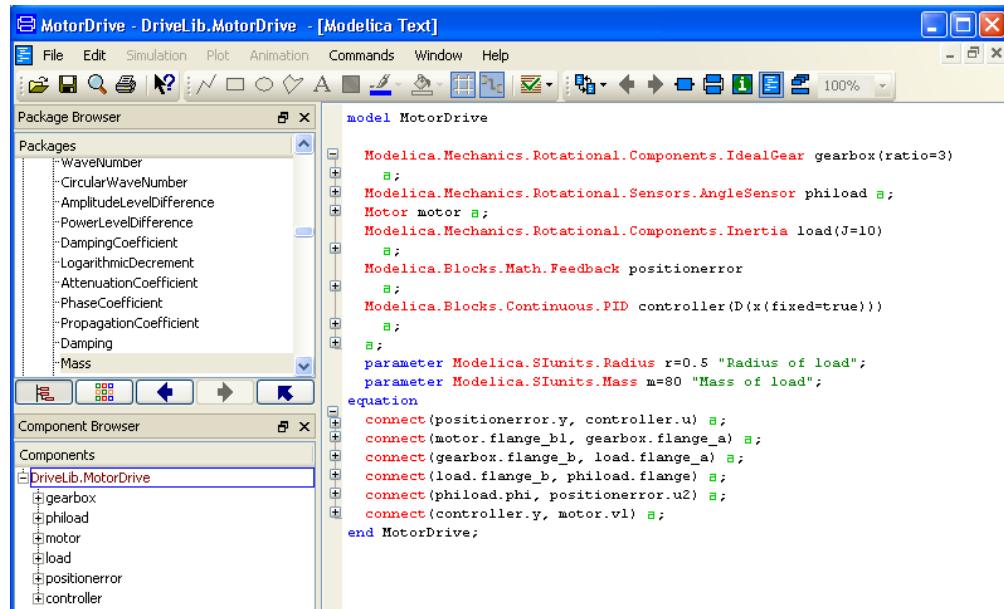
One way is to work with individual annotation. It is possible to click on the “+” or the corresponding icon to expand that annotation but no other. If “sub-annotations” are present, new “+” and icons will be visible, which in turn can be expanded. (Of course they can also be compressed using “-“.)

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 connect**. 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 connect** 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 compress that annotation individually).

### Expanded text representation.

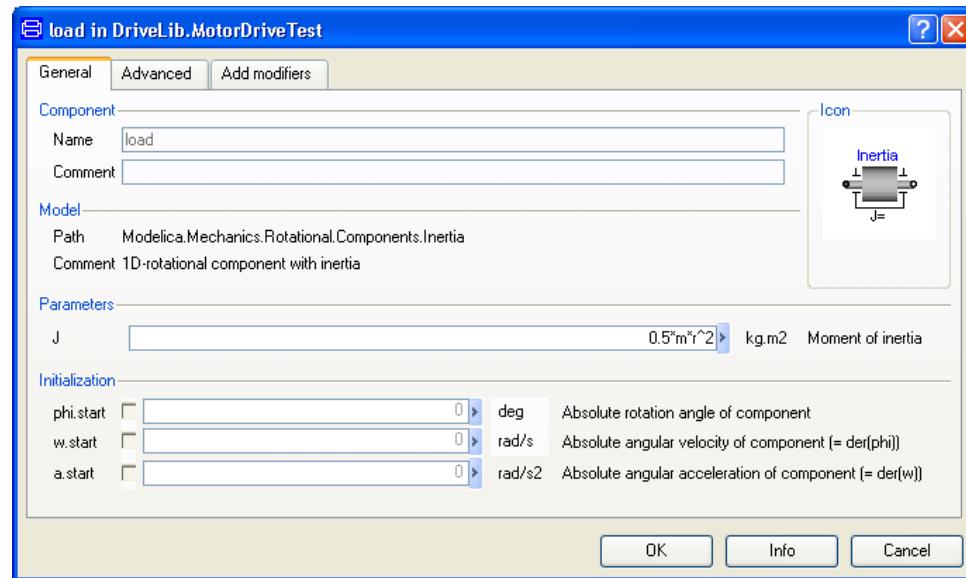


The screenshot shows the Dymola interface with the title bar "MotorDrive - DriveLib.MotorDrive - [Modelica Text]". The menu bar includes File, Edit, Simulation, Plot, Animation, Commands, Window, and Help. The toolbar contains various icons for file operations and simulation. The left side features two browser panes: "Package Browser" and "Component Browser". The "Component Browser" pane has "Components" expanded, showing "DriveLib.MotorDrive" selected, along with its sub-components: gearbox, phiload, motor, load, positionerror, and controller. The main central area displays the Modelica code for the "MotorDrive" model:

```
model MotorDrive
  Modelica.Mechanics.Rotational.Components.IdealGear gearbox(ratio=3);
  Modelica.Mechanics.Rotational.Sensors.AngleSensor phiload;
  Motor motor;
  Modelica.Mechanics.Rotational.Components.Inertia load(J=10);
  Modelica.Blocks.Math.Feedback positionerror;
  Modelica.Blocks.Continuous.PID controller(D(x(fixed=true)));
  parameter Modelica.SIunits.Radius r=0.5 "Radius of load";
  parameter Modelica.SIunits.Mass m=80 "Mass of load";
equation
  connect(positionerror.y, controller.u);
  connect(motor.flange_b1, gearbox.flange_a);
  connect(gearbox.flange_b, load.flange_a);
  connect(load.flange_b, phiload.flange);
  connect(phiload.phi, positionerror.u2);
  connect(controller.y, motor.v1);
end MotorDrive;
```

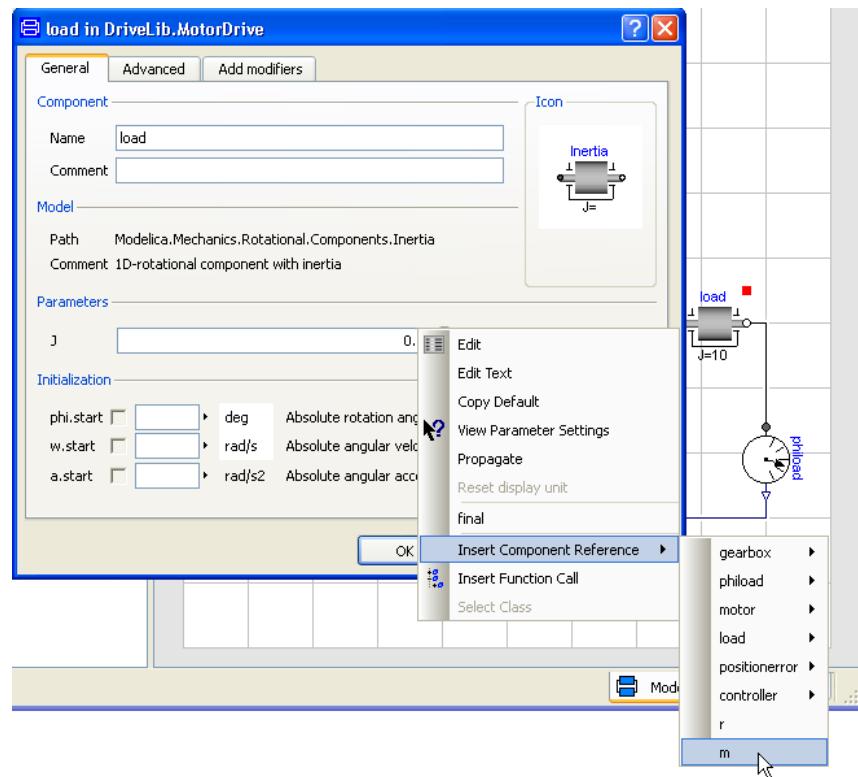
Ok, now activate the diagram representation. Double-click on the load icon to open the parameter dialog.

**Binding a parameter to an expression.**



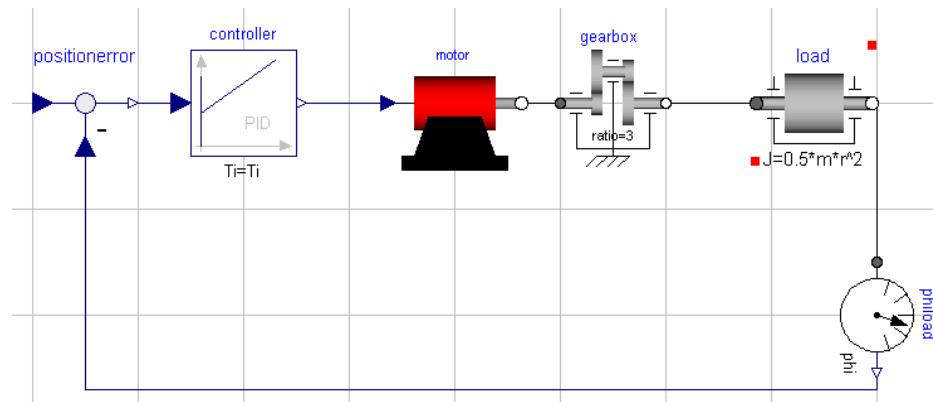
Click in the value field of J and enter the expression for J. Click **OK**. The model window now displays the expression for the load inertia. When entering the expression, you are sometimes not sure about the exact name of the variables names, for example is the radius called r, r0 or r1? The problem 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.**

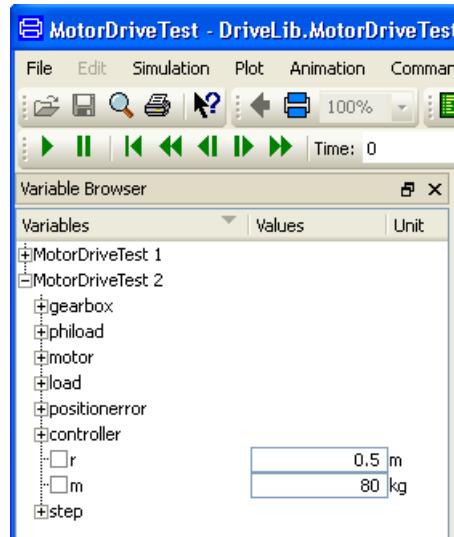


Open MotorDriveTest and switch to Simulation mode.

Translate.



**A bound parameter  
cannot be changed in-  
teractively.**



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  $m$ .

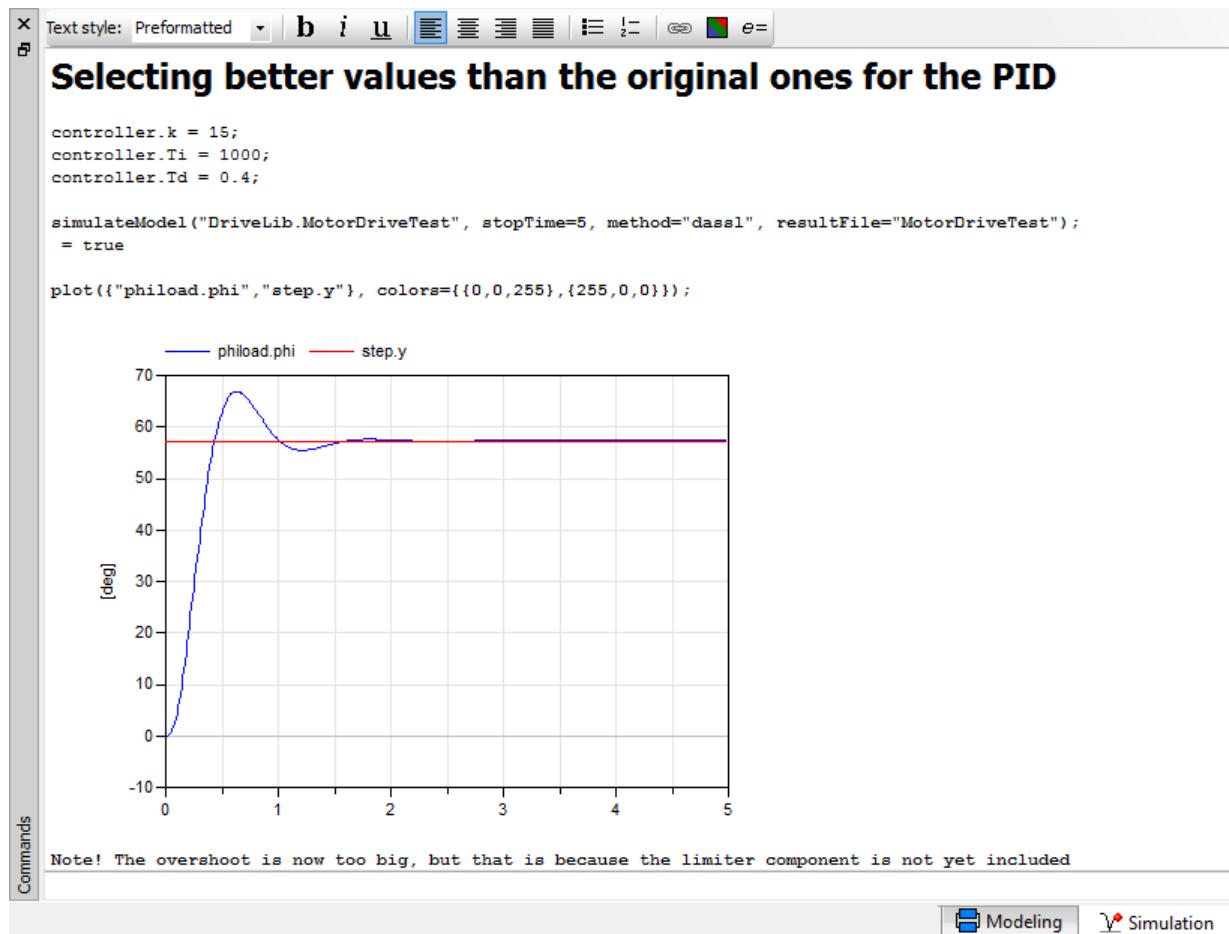
#### 2.4.9 Documenting the simulation

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.

Plot results can also be inserted in the command log (together with the corresponding plot command) by using the context command **Show In > Command Window**. (Note that you must not right-click on a curve etc., but in an empty area of the window, to get the general context menu.)

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 be automatically be included.

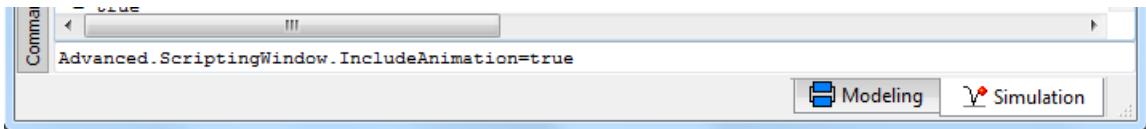
An example of documenting better values of the PID controller could be:



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 chapter “Simulating a model” in the manual. 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, the user right-clicked an empty space in the plot window, and selected **Show In > Command Window**.
- 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 a typical example 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.

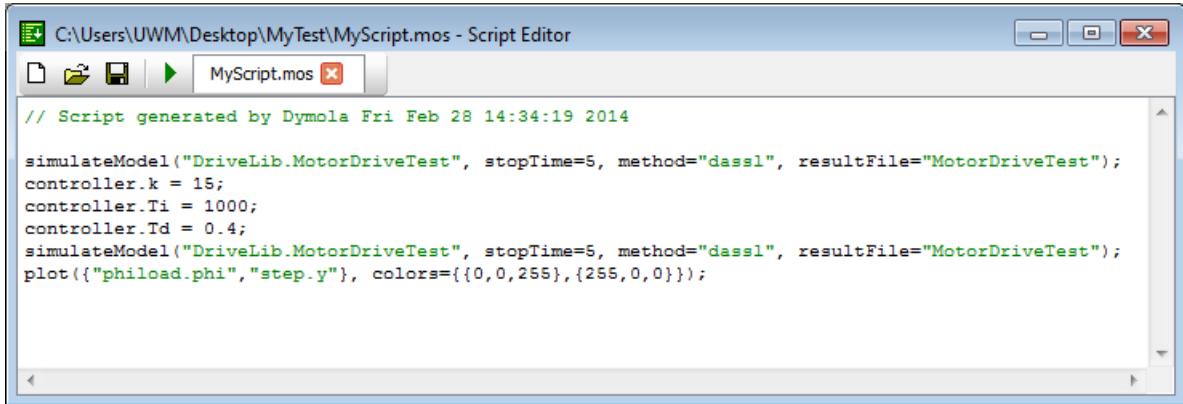
Creating functions is not treated here, please see chapter “Simulating a model” in the manual for more about scripting, in particular using functions.

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. Go to Modeling mode, 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.)



The screenshot shows the Dymola Script Editor window with the title bar "C:\Users\UWM\Desktop\MyTest\MyScript.mos - Script Editor". The editor window displays the following content:

```
// Script generated by Dymola Fri Feb 28 14:34:19 2014

simulateModel("DriveLib.MotorDriveTest", stopTime=5, method="dassl", resultFile="MotorDriveTest");
controller.k = 15;
controller.Ti = 1000;
controller.Td = 0.4;
simulateModel("DriveLib.MotorDriveTest", stopTime=5, method="dassl", resultFile="MotorDriveTest");
plot({"phiLoad.phi", "step.y"}, colors={{0,0,255}, {255,0,0}});
```

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 Simulation mode. The command **Simulation > Script > Run Script...** (or corresponding command button) can be used to open (execute) the script.

Some comments:

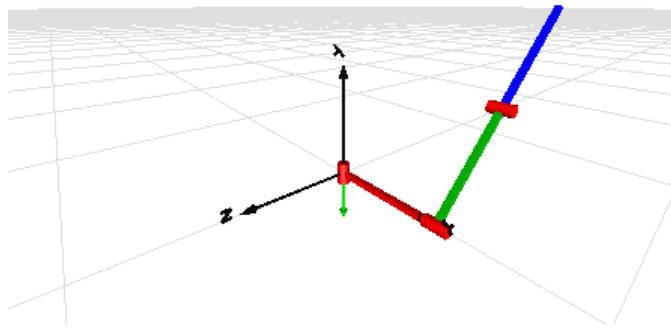
- This script file is very simple, just showing the idea of a script file rather than being a good example.
- Realizing how saving a script file works, it is not necessary to start all over to create the file, the total simulation can be saved, and afterwards the script file can be edited to keep only 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” in the manual for more information.

- Working with scripting using functions is even smarter, for more information please see the manual.

## 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.

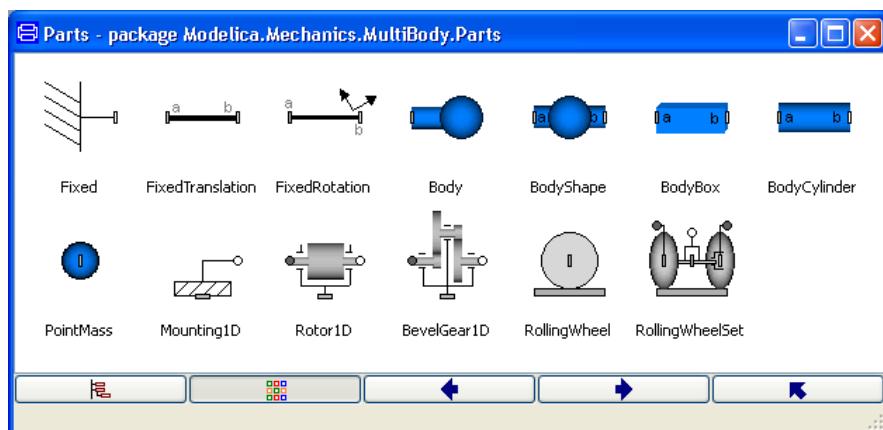
**The Furuta pendulum.**



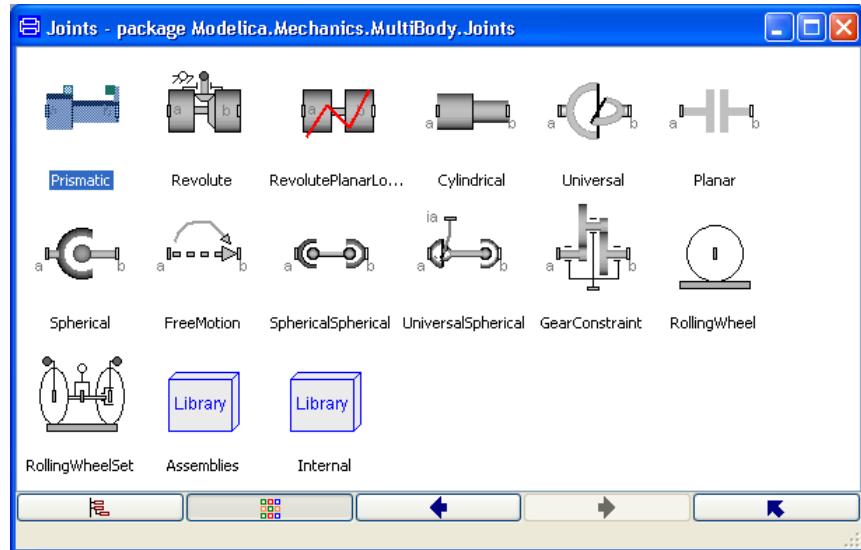
Start Dymola. Open Modelica Standard Library. In the package Mechanics open the sub-package MultiBody. This package includes 3D mechanical components such as joints and bodies, which can be used to build a model of the Furuta pendulum.

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.**

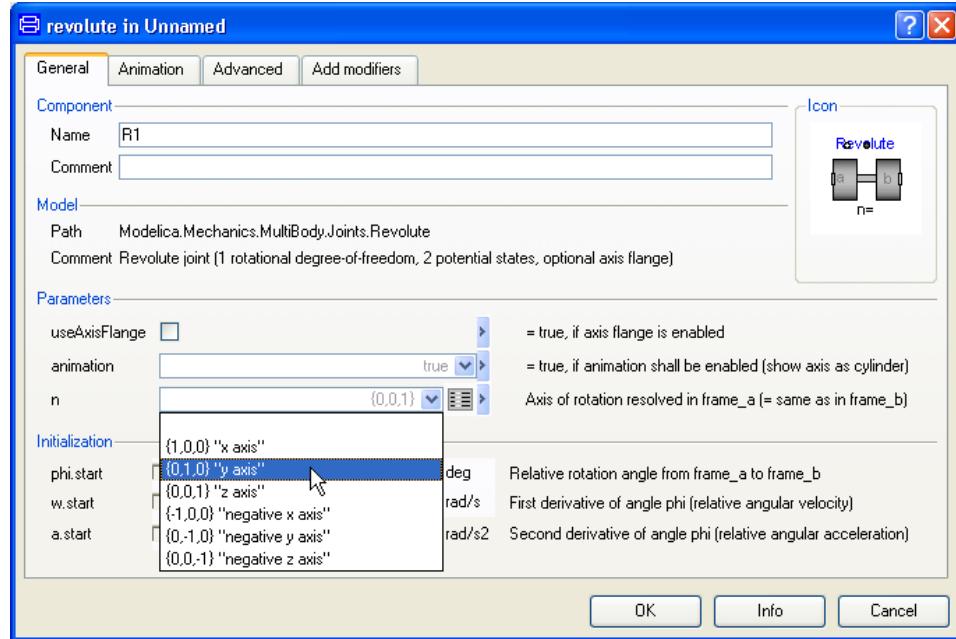


Select **File > New... > Model** and give the name **Furuta**.

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 edit window. 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 **Edit > Rotate 90**. This just changes the appearance of the icon. Double-click on the icon.

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.



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 icon. 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.

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.

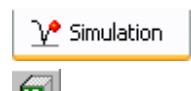
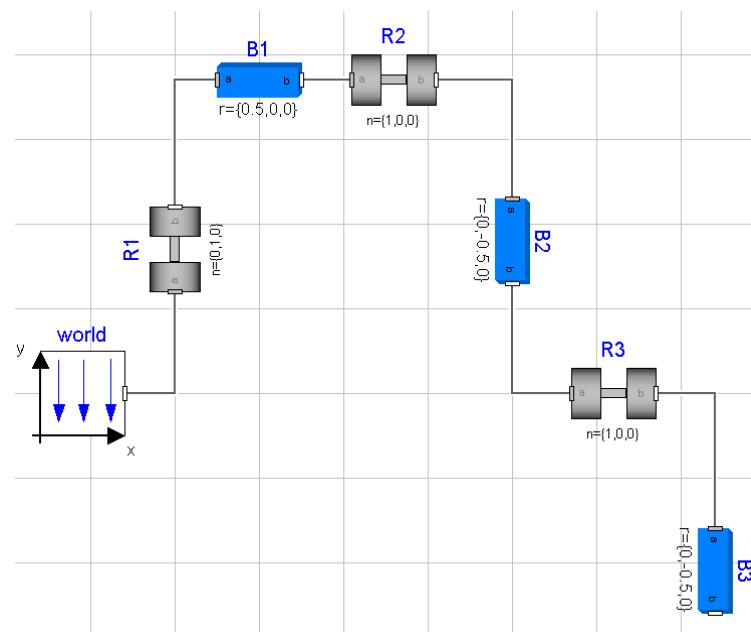
Fixed
True: start-value is used to initialize
False: start-value is only a guess-value
✓ Inherited: (False: start-value is only a guess-value)

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**. (The selection of the start value type for phi.start can be removed from R3 if wanted.)

You should now have arrived at a model that is similar to the following.

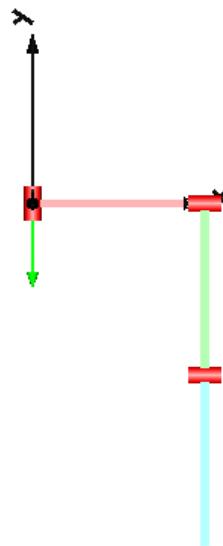
### The diagram of the Furuta pendulum.



Now it is time to simulate. To enter the Simulation mode, click on the tab at the bottom right. The simulation menu is now activated and new tool bar buttons appear.

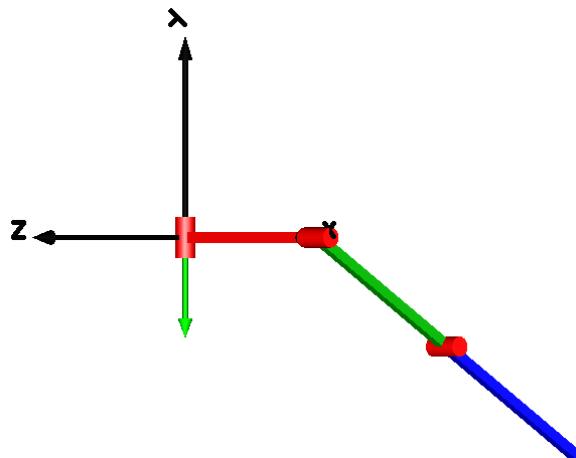
When building 3D mechanical models, it is possible to get visual feedback to check that the mechanical system is put together as intended. Click on the toolbar button **Visualize**. The animation window shows the initial configuration of the system.

**Initial configuration of system.**



Translate the model. In the variable browser, open R2 by clicking on the + in front of it and enter a value for phi\_start, say 1 rad, and simulate for 5 seconds (use the command **Simulation > Setup...** to change the stop time). 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 33 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.

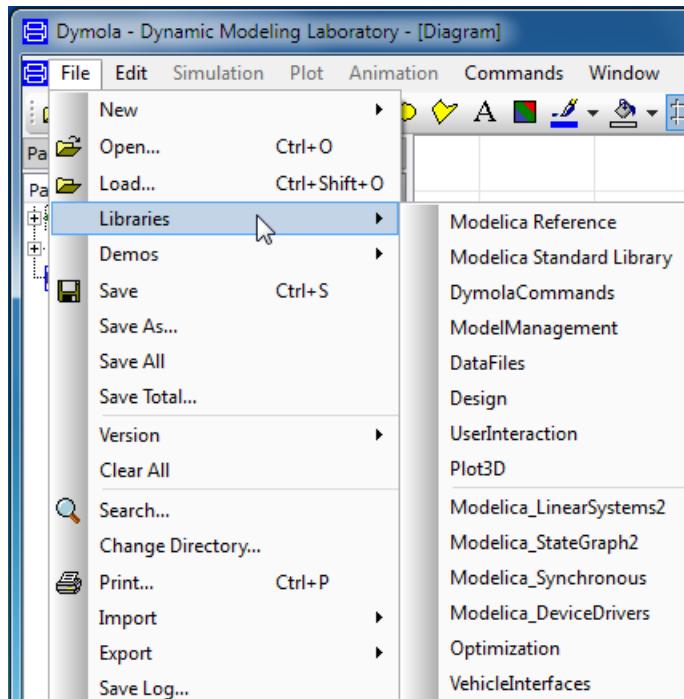
Try to control the pendulum in such a way as to get a stable upright position. (A trick is to use a “mechanical PD controller”, i.e. a spring and damper attached to the tip of the second bar and to a high position.)

## 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.**



**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.

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

**DymolaCommands** contains selected commands (built-in functions). They are grouped in the same way as they are documented in chapter 5 in this manual (section “Scripting”, subsection “Built-in functions in Dymola”). Note that this library is automatically opened when opening Dymola.

**Model Management** deals with version management, model dependencies, encryption, model and library checking (including regression testing of libraries), model comparison and model structure. It cannot be used free, it demands a license. For more information, please see the manual “Dymola User Manual Volume 2”, chapter “Model Management”.

**DataFiles** contains functions for reading and writing data. For more information, please see 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. The Model Calibration option is required for problems with more than one tuner. For more information, please see the manual “Dymola User Manual Volume 2”, 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 manual “Dymola User Manual Volume 2”, 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 more complex models. For more information, please see the manual “Dymola User Manual Volume 2”, chapter “Design optimization”. Note that a newer library is available also; 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.

**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\_Synchronous** library is a free library from Modelica Association 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. Optionally, quantization effects, computational delay or noise can be simulated. Continuous-time equations can be automatically discretized and utilized in a sampled data system. The sample rate of a partition needs to be defined at one location only. All Modelica libraries designed so far for sampled systems, such as Modelica.Blocks.Discrete or Modelica\_LinearSystems2.Controller are becoming obsolete and instead Modelica\_Synchronous should be utilized.

**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.

**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.

**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**.

## 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 **What's This?** command/button etc

The **What's This?** command is used to display information about many things. The command is activated in three different ways depending on where it should be used.

## In the Main Dymola window



If more information should be obtained for e.g. a button or a symbol in the Dymola main window, the easiest way is to click on the **What's This?** button and then click on the symbol that is of interest. (The **Help** menu can also be used to activate this function.)

Please note that this button does not work in the Edit window.

## In any dialog



In any dialog the **What's This?** 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, for components it will be the name of the component + the path to it.

## The Search command/button



The search functionality can be reached either using the command **File > Search** or clicking the **Search** button. See chapter “Developing a model”, section “Editor command reference - Modeling mode”, sub-section “Main window: File menu”, command “File > Search” for more information.

## **2.7.2 Reaching more extensive information**

### **The Info command/button**

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

### **The Help menu**

The **Help** menu can be used to display manuals and to go to the Dymola website. The help menu also contains the license handling. The help menu is described in chapter “Developing a model”, section “Editor command reference - Modeling mode”, sub-section “Main window: Help menu”.

### **The Search command/button**

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

### **The documentation layer of Edit window**

This layer can be used to display more extensive information about packages and components. Please see 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.

### **Books etc**

Manuals are available for Dymola and a number of libraries used, as well as for the Modelica language.



# **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.

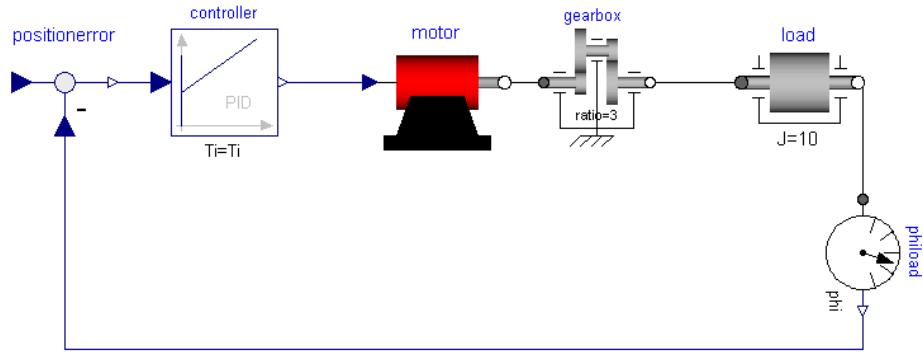
The benefits of acausal modeling with DAE's will be demonstrated in this chapter and compared to traditional block diagram modeling. It will also be shown that tools can incorporate computer algebra methods to translate the high-level Modelica descriptions to efficient simulation code.

---

## **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

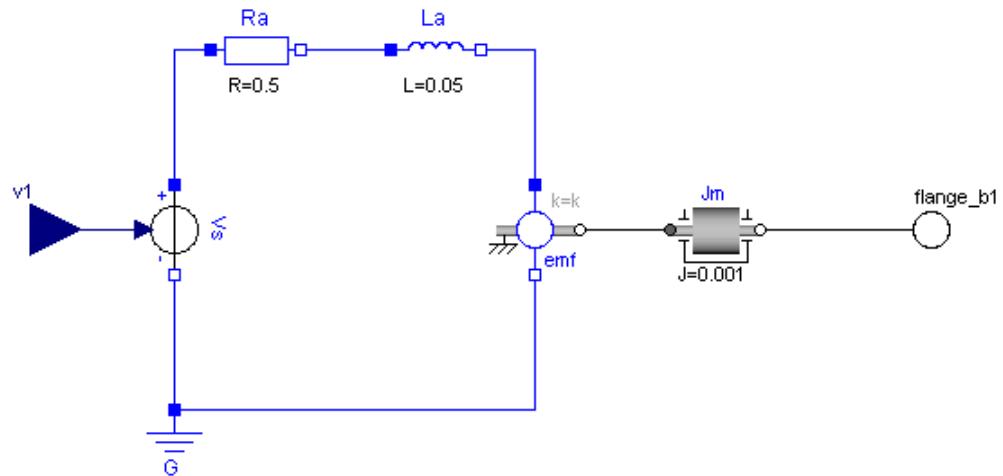
```

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 pinpoint what is wrong. It may be that a component is wrong of that you have connected them in a bad or unforeseen way.

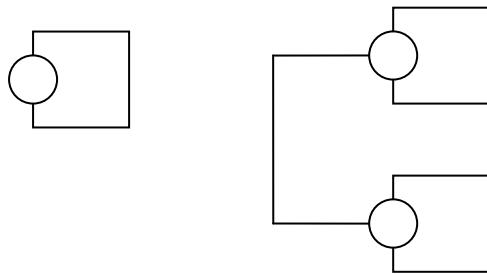
For a component having only connectors being defined as inputs and outputs it is possible to check that the component has the correct number of equations. We can here assume the inputs to be known and count that the component has the correct number of equations to calculate the output variables and the 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

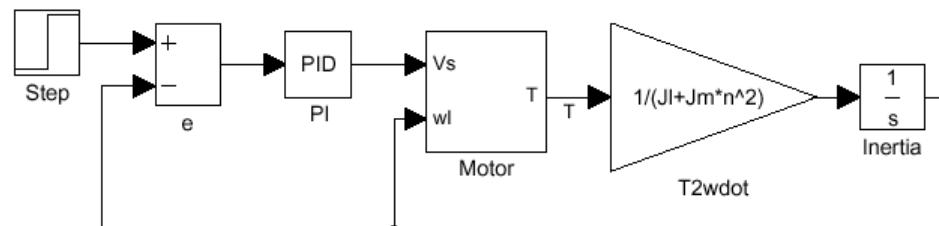
$$\frac{dx}{dt} = f(x, u)$$

$$y = g(x, u)$$

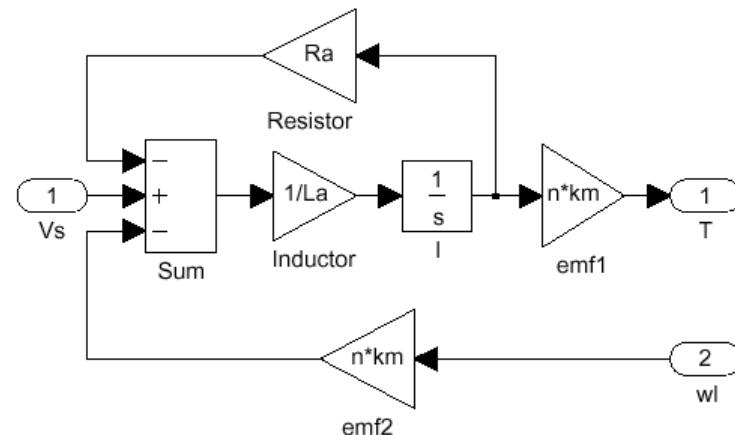
where **u** is input, **y** is output and **x** is the state. It is rare that a natural decomposition into subsystems leads to such a model. Often a significant effort in terms of analysis and analytical transformations is needed to obtain a problem in this form. It requires a lot of engineering skills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

To illustrate the difficulties, a Simulink model for the simple motor drive (see page 108) 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,

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

where  $\mathbf{x}$  is the vector of unknowns that appear differentiated in the equation and  $\mathbf{y}$  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 108.

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  $J_i + J_m n^2$ . 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  $a_0 + a_1s + \dots + a_ns^n$  are given as a vector  $\{a_0, a_1, \dots, a_n\}$ .

```
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 108 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 108 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,

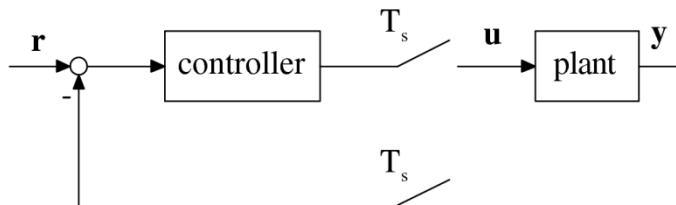
state charts, sequential function charts, DEVS, logical circuits, difference equations, CSP, process-oriented languages that are all suited for particular application areas.

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

$$\begin{aligned}\frac{dx_p}{dt} &= \mathbf{f}(x_p, u) \\ y &= \mathbf{g}(x_p)\end{aligned}$$

is controlled by a digital linear controller

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

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  $t_i \leq t \leq t_i + T_s$ ), 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;

```

```



```

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 `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 `pre(xc)` 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.

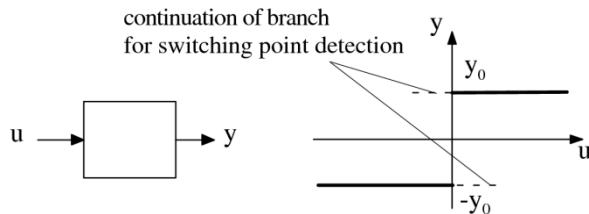
Handling hybrid systems in this way has the advantage that the synchronization between the continuous time and discrete event parts is automatic and leads to a deterministic behavior without conflicts. Furthermore, some difficulties to detect errors of other approaches, such as deadlock, can often be 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 Elmquist *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 `sqr` 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;
```

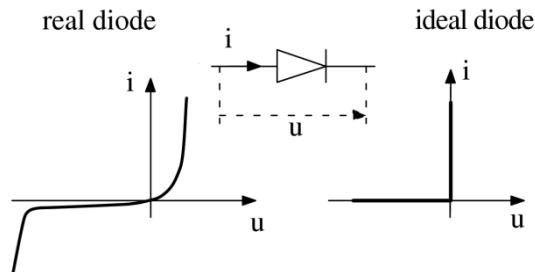
Modelica has a set of additional operators, such as **initial()** and **terminal()** to detect the initial and final call of the model equations, and **reinit(...)** to reinitialize a continuous state with a new value at an event instant. For space reasons, these language elements are not discussed. 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.

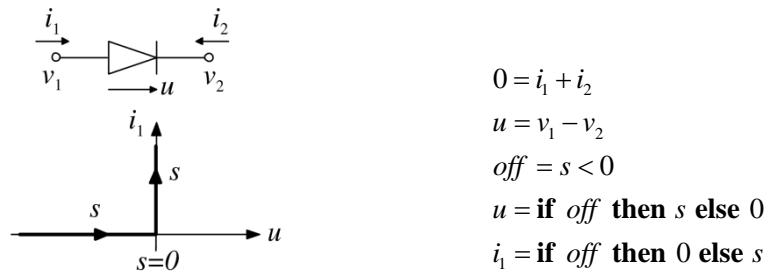
**Real and ideal diode characteristic.**



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.**



$$0 = i_1 + i_2$$

$$u = v_1 - v_2$$

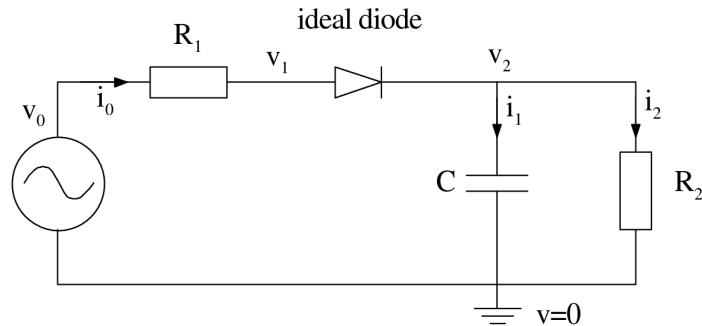
$$off = s < 0$$

$$u = \text{if } off \text{ then } s \text{ else } 0$$

$$i_1 = \text{if } off \text{ then } 0 \text{ else } s$$

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

$$off = s < 0$$

$$u = v_1 - v_2$$

$$u = \text{if } off \text{ then } s \text{ else } 0$$

$$i_0 = \text{if } off \text{ then } 0 \text{ else } s$$

$$R_1 \cdot i_0 = v_0(t) - v_1$$

$$i_2 = \frac{v_2}{R_2}$$

$$i_1 = i_0 - i_2$$

$$\frac{dv_2}{dt} = \frac{i_1}{C}$$

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.

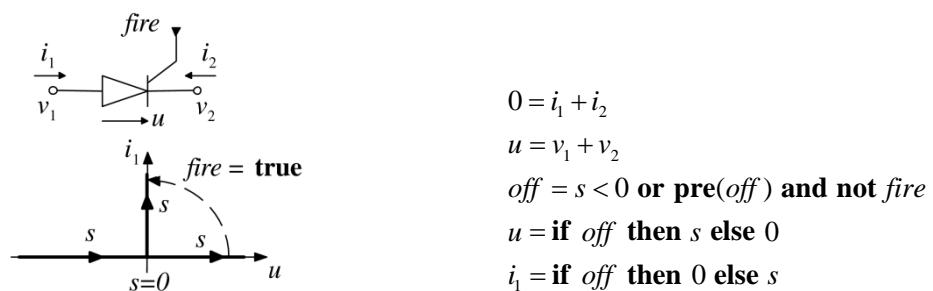
At an event instant, the first 5 equations are a mixed system of discrete and continuous equations which cannot be solved by, say, Gaussian elimination, since there are Real and Boolean unknowns. However, appropriate algorithms can be constructed: (1) Make an assumption about the values of the relations 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:

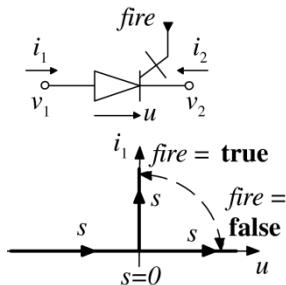
- Use the relation values computed in the last iteration.
- Try all possible combinations of the values of the relations systematically (exhaustive search).

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.



**Ideal GTO thyristor**

$$0 = i_1 + i_2$$

$$u = v_1 + v_2$$

*off* =  $s < 0$  or **not** *fire*

*u* = **if** *off* **then** *s* **else** 0

*i*<sub>1</sub> = **if** *off* **then** 0 **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), Elmquist *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,  $\frac{dx}{dt} = f(x, t)$ , 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  $\frac{dx}{dt} = 0$  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;  
else if 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,  $dx/dt = \mathbf{f}(\mathbf{x}, t)$ , exactly  $\dim(\mathbf{x})$  additional conditions are needed, in order to arrive at  $2*\dim(\mathbf{x})$  equations for the  $2*\dim(\mathbf{x})$  unknowns  $\mathbf{x}(t_0)$  and  $dx/dt(t_0)$ .

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

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

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

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

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  $\dim(\mathbf{x})$ .

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.

Please also see chapter “Simulating a model”, section “Handling initialization and start values”, sub-section “Over-specified initialization problems” for additional information.

### 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.

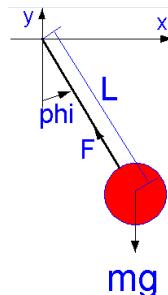
### Non-linear algebraic loops

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  $2\pi$  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.

Please also see chapter “Simulating a model”, section “Handling initialization and start values”, sub-section “Discriminating start values” for additional information.

### 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 through 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  $v = \text{pre}(v)$  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.
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 <b>File &gt; Libraries &gt; State Graph</b> .
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.
SIunits	Type and unit definitions based on SI units according to ISO 31-1992.

### **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. Jurth, 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. Fritzson (1999): "Design, simulation and visualization environment for object-oriented mechanical and multi-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 Systèmes 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. Elmquist (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. Elmquist, 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. Elmquist, 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 in Modeling mode”** starting at page 144 the windows available in Dymola in Modeling mode are described as an overview. More information about the use of some windows is given in the following sections.

Section 4.2 **”Basic model editing”** starting on page 161 describes the basics of how to edit a model (creation/editing of components, connectors, parameters/variables, graphical objects, equations and documentation).

Section 4.3 **”Advanced model editing”** starting on page 235 presents powerful tools for a more advanced model editing.

Section 4.4 **”Checking the model”** starting on page 283 describes the testing of the model. Unit checking and unit deduction is dealt with here.

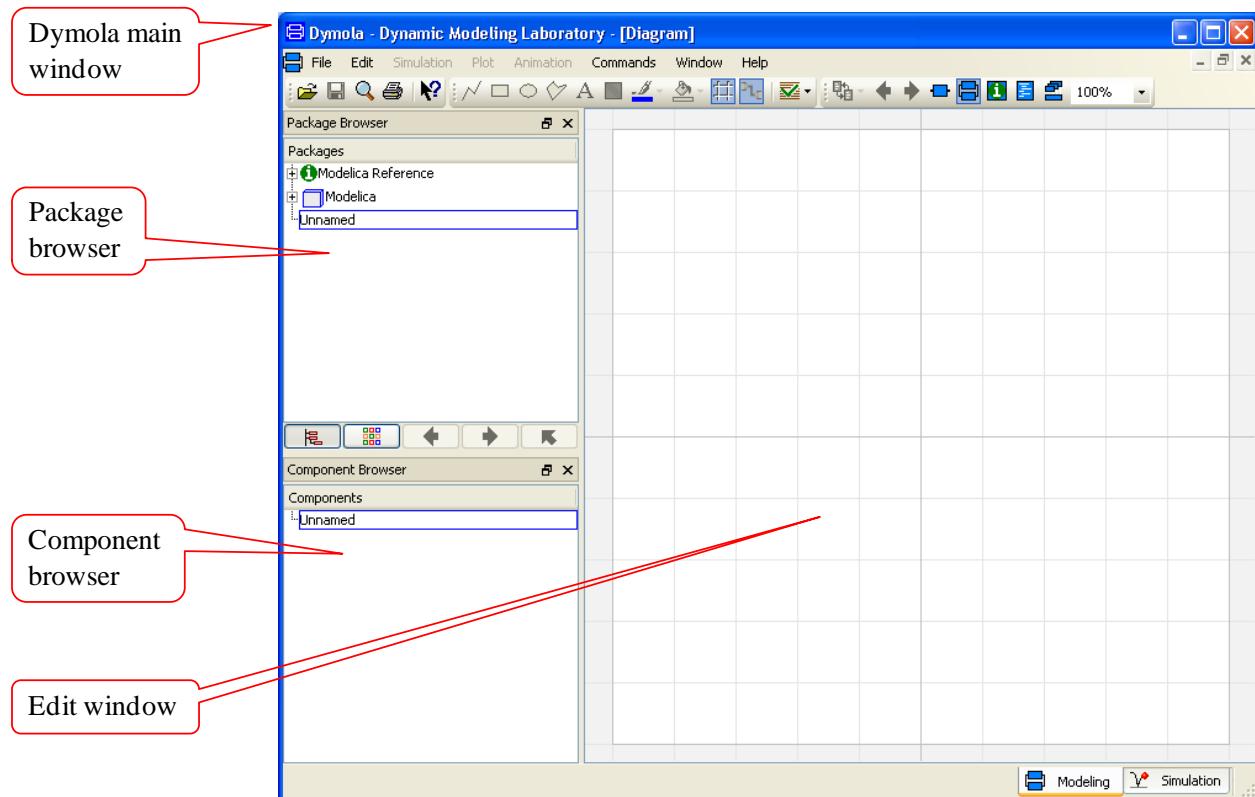
Section 4.5 **”Editing model reference”** starting on page 289 presents some items of interest for engineers building their own libraries.

Section 4.6 **”Editor command reference – Modeling mode”** starting on page 291 is a reference section, describing the menus available in Modeling mode. 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 in Modeling mode.

## 4.1 Windows in Modeling mode

A number of window types are available in Dymola in Modeling mode. The first time Dymola is started it looks the following:



What is seen is the Dymola main window which contains a number of sub-windows.

The user can decide what windows should be seen and some general settings concerning windows. Please see section “Window settings” on page 289 for more information.

### 4.1.1 Dymola Main window

What is available in the Dymola Main window by default depends partly on which mode Dymola is in - **Modeling** mode (which enables building/editing of models), or **Simulation**

mode (where the models can be simulated). This chapter deals with Dymola in Modeling mode. Dymola in Simulation mode is dealt with in the next chapter.

### Default content in Dymola Main window in Modeling mode

Dymola Main window contains by default the following in Modeling mode (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 – Modeling mode”, starting on page 291.
- Three sub-windows – package browser, component browser and edit window.
- Two window mode tabs (to the lower right) that is used to decide whether Dymola should be in **Modeling** mode or **Simulation** mode. These tabs are available in both modes.



(Alternatively the keyboard shortcuts **Ctrl+F1** and **Ctrl+F2** or the **Window > Mode** menu can be used. Please see “Main window: Window menu” on page 325.)

### Additional content available

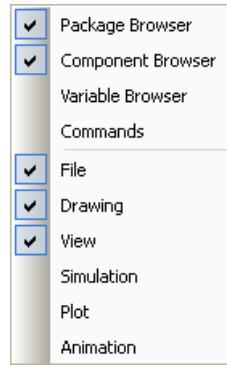
Additional content available depends on:

- User selection of content; by using the command **Window > Tools** the content of Dymola Main window can be changed. Features by default only available in Simulation mode can be added, and what toolbars are shown can be selected. (This menu is also available using the context menu of the main window, see next section for additional information.)
- Errors detected (e.g. syntax errors when programming). If errors are detected a message window will pop.
- When using any **Info** button or **Info** command an information browser pops.

### Context menu

Right-clicking in an empty space in the commands/tool bar section of the main window will display a menu where the content of the main window (available sub-windows and toolbars) can be changed.

**Context menu of main window in Modeling mode.**



This works also by right-clicking in an empty space in the gray header of any sub-window. For more information about this command, please see section “Window > Tools” on page 327.

### 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 in the command button line - the order of the buttons corresponds to the list above. (The layer can also be selected using commands.)

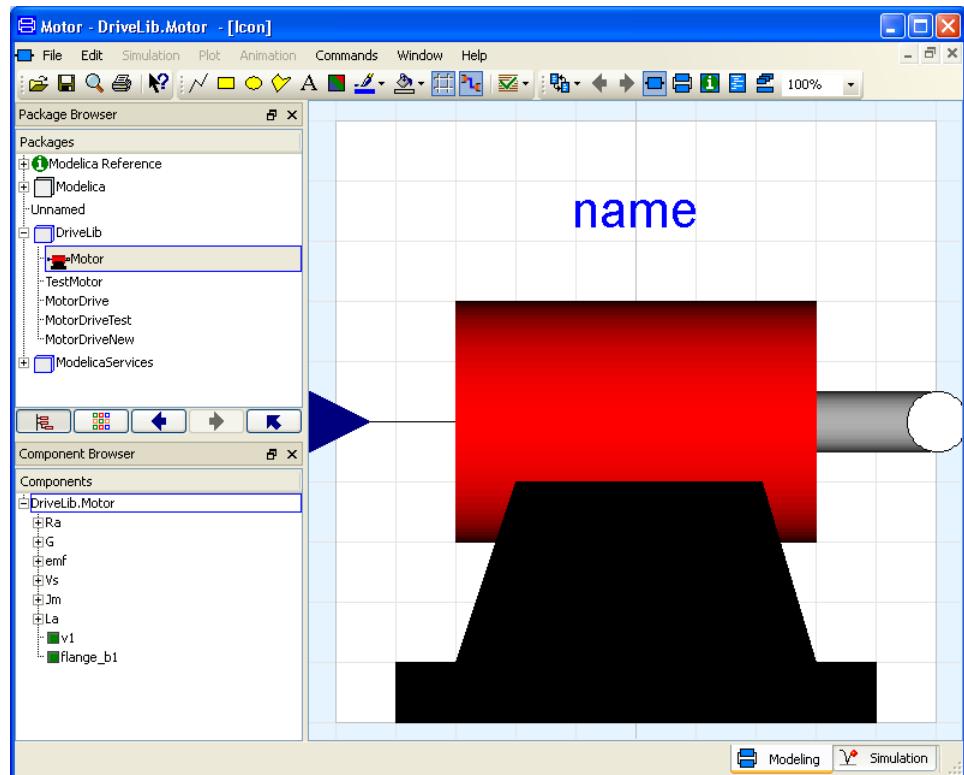
**Selection of class layers by buttons.**



What layer *is* displayed as top layer is indicated in three ways: by the button line (in this case the diagram layer is on top), by an icon to the left of the command bar (to the left of **File** menu.) and finally by a text in brackets in the end of the Dymola main window header.

### Icon layer

**The icon layer of the edit window.**

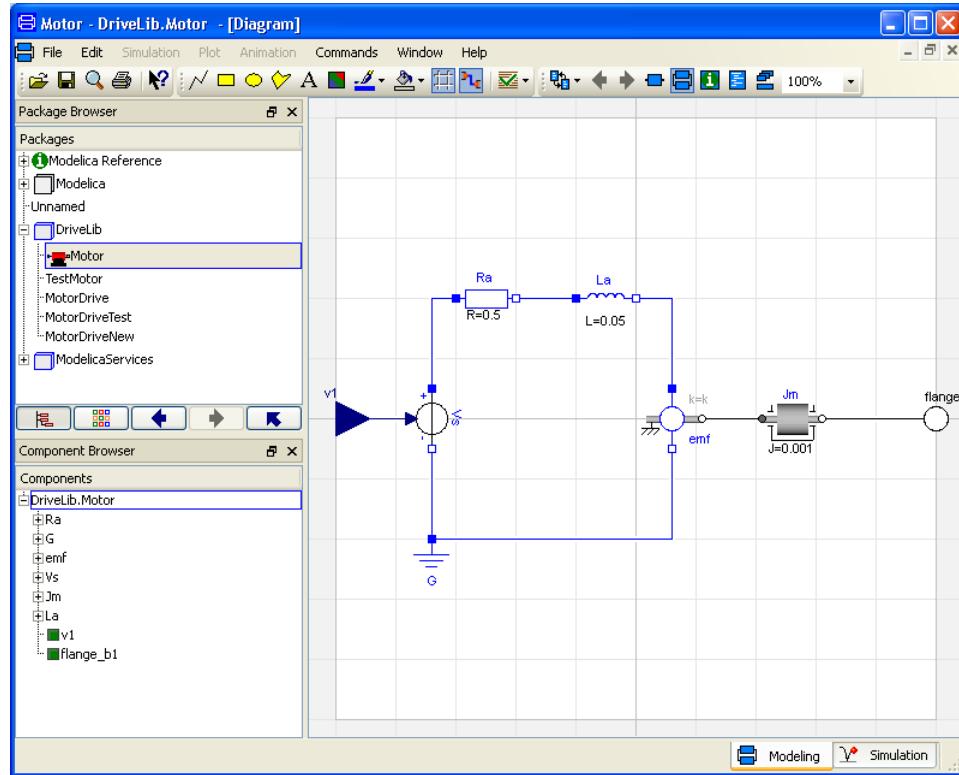


The icon layer represents the class when it is used as a component or connector in another model, that is, the diagram layer displays the icon layer of its components. The surrounding background of the icon layer is light blue when editing is possible.

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.

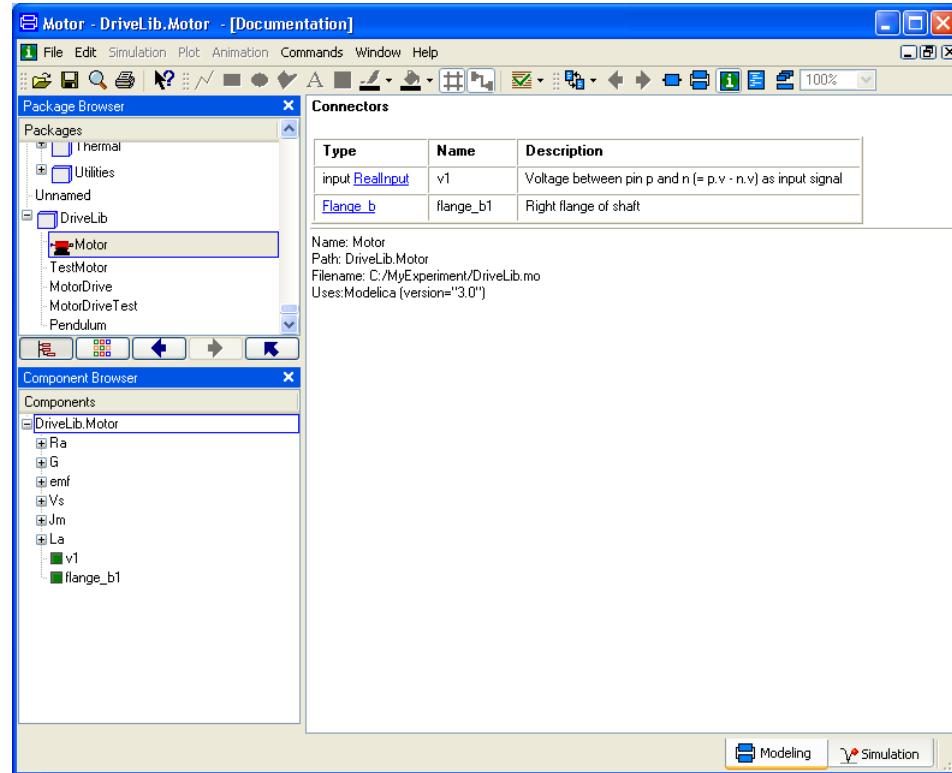
The name of a component can be shown as a tooltip by resting the cursor on it.

Moving the view and zooming is possible. Please see section “Moving and zooming in the diagram layer” on page 401.

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.**



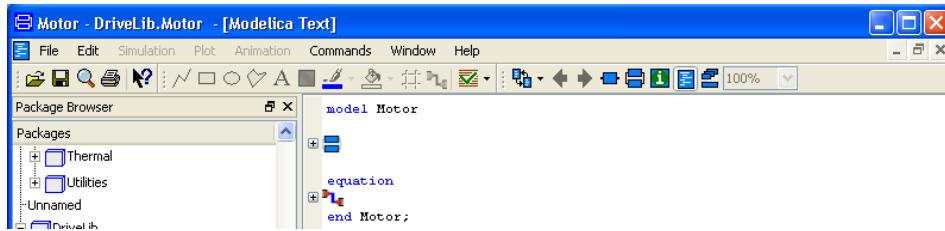
The documentation layer shows the one-line description of the class, plus the longer info text that gives a complete explanation of its behavior. Information about parameters, connectors, inputs/outputs, package content and revision is displayed as well. These texts are available when browsing classes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

More about documentation is presented in the section “Documentation” on page 215.

## Modelica Text layer

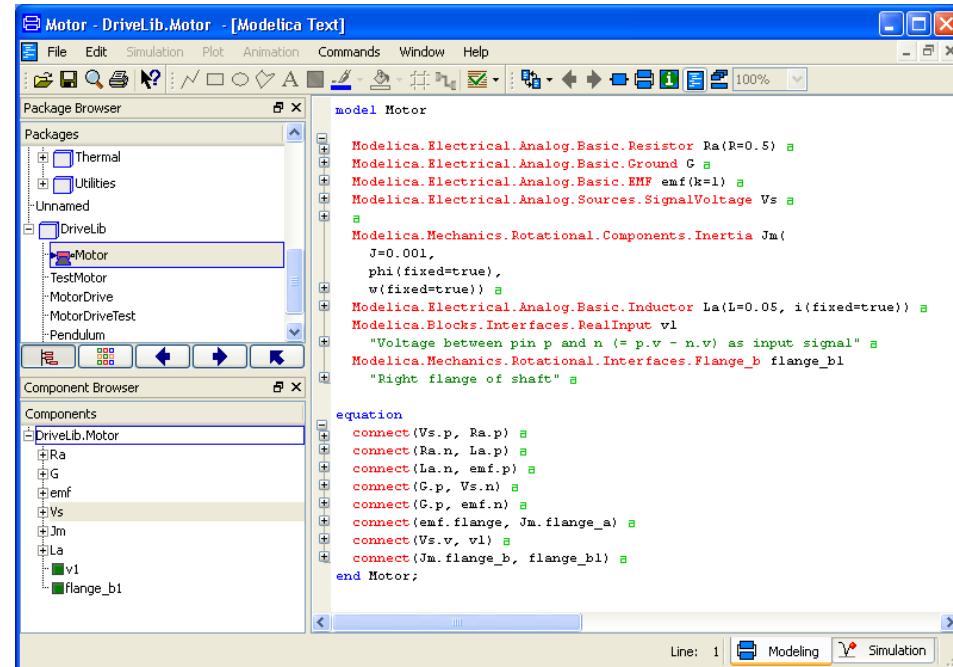
**Modelica Text layer of the edit window.**



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 “+” symbols or by right-clicking and selecting **Expand > Show components and connect**. 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 193.

## Used Classes layer

Used Classes layer of edit window.



```
model Motor
  Modelica.Electrical.Analog.Basic.Resistor Ra(R=0.5);
  Modelica.Electrical.Analog.Basic.Ground G;
  Modelica.Electrical.Analog.EMF emf;
  Modelica.Electrical.Analog.Sources.SignalVoltage Vs;
  Modelica.Mechanics.Rotational.Components.Inertia Jm(J=0.001);
  Modelica.Electrical.Analog.Basic.Inductor La(L=0.05);
  Modelica.Blocks.Interfaces.RealInput v1
    "Voltage between pin p and n (= p.v - n.v) as input signal";
  Modelica.Mechanics.Rotational.Interfaces.Flange_b flange_b1
    "Right flange of shaft";
equation
  connect(Vs.v, v1);
  connect(Jm.flange_b, flange_b1);
  connect(Vs.p, Ra.p);
  connect(Ra.n, La.p);
  connect(La.n, emf.p);
  connect(emf.flange, Jm.flange_a);
  connect(emf.n, G.p);
  connect(Vs.n, G.p);
end Motor;

model Modelica.Electrical.Analog.Basic.Resistor
  "Ideal linear electrical resistor"
  extends Interfaces.OnePort;
  parameter SI.Resistance R(start=1) "Resistance";
equation
  R*i = v;
end Resistor;

partial model Modelica.Electrical.Analog.Interfaces.OnePort
  "Component with two electrical pins p and n and current i from p to n"

  SI.Voltage v "Voltage drop between the two pins (= p.v - n.v)";

  
```

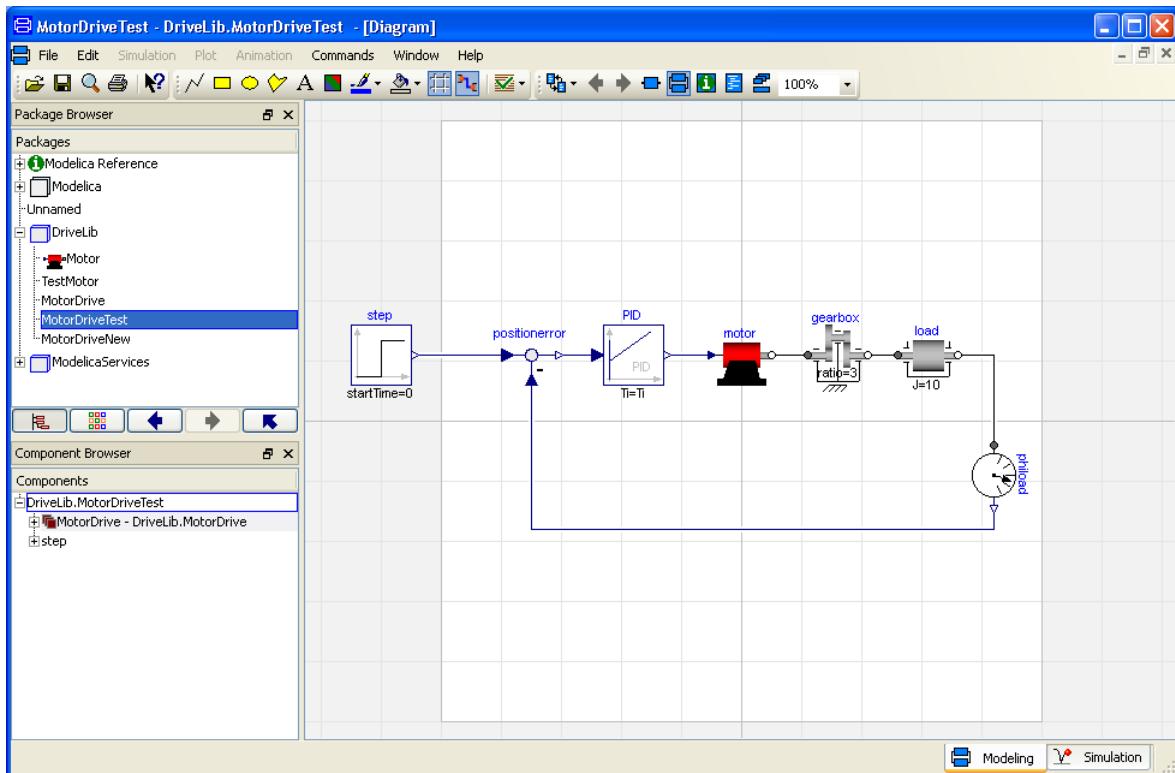
The used classes' layer shows the base classes, classes used for components and connectors, and the Modelica text of the class. This facilitates searching for the location of e.g. a specific variable.

By selecting the context command **Flat Modelica** the flat Modelica text is showed instead, if a block or a model is selected for display. The flat Modelica text is the result when component structure and inheritance in the model/block has been evaluated; it corresponds to the result when translating the model/block. 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 213.

## Coordinate system



Every class has a master 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 master 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 “Zooming” on page 326). The master 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.

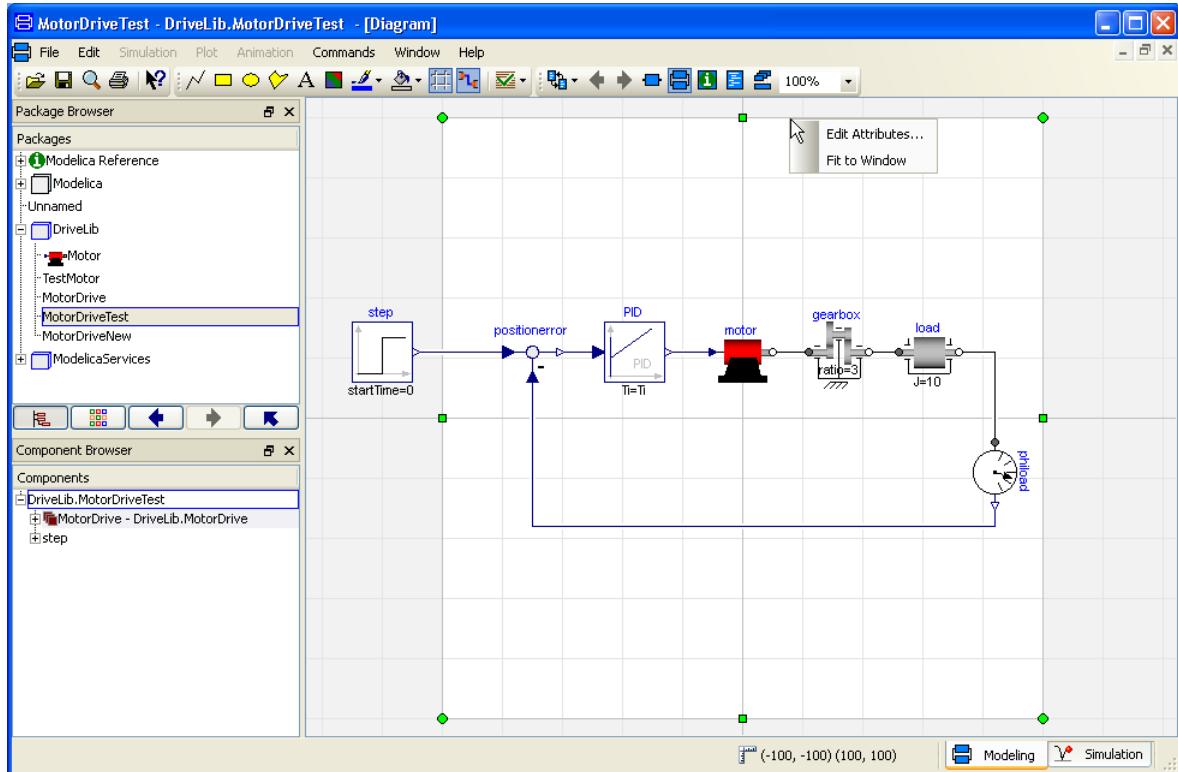
Associated with every model class is also a default component size. Components and connectors dragged into the model class will initially have the default size, but can be reshaped after insertion.

### 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 **Edit > Attributes** menu (see “Graphics tab” on page 317). 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 high-lighted 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 right-click and select **Fit to Window** to zoom to 100 %.

The default coordinate system, grid and component size are either

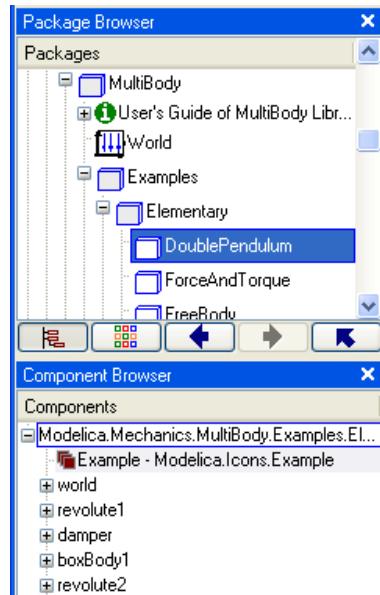
- Inherited from a base class, or
- Copied from the program defaults  $(-100, -100)$  to  $(100, 100)$  with grid  $(2, 2)$ . The default component size is  $(20, 20)$ .

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

## 4.1.3 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.

### Window handling



The package browser and component browser can be undocked using the **Undock** button or by double-clicking on the window header or dragging it.

To dock the window again, the best way is to double-click on the window header. It can also be dragged to its original place (you will then see how space for it is created automatically before releasing it).

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.

### 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

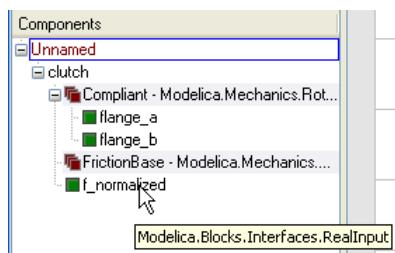
window). By *double-clicking* on the package in the package browser the displayed package is changed, however. (An alternative is to right-click on the package in the package browser and select any **Open Class...** command.)

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. The content is sorted in alphabetical order by clicking on the header line. Clicking a second time restores declaration order. Typing e.g. "m" will display the first components starting with "m" etc.

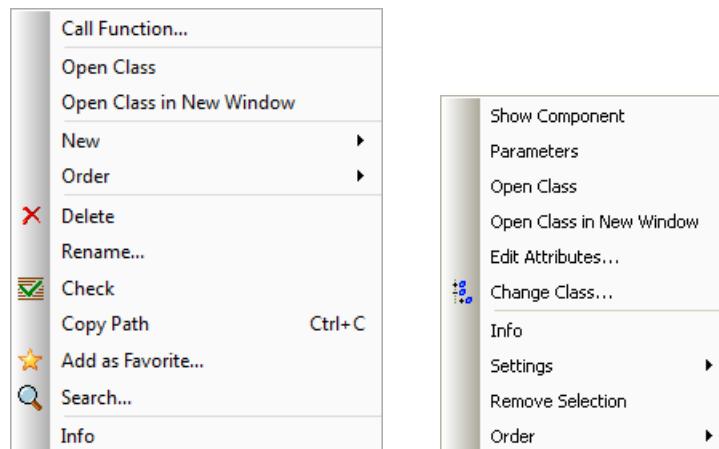
By dragging components from the package browser to the component browser certain actions can be obtained (changing of type, variable declaration etc.). Please see section "Inserting a component or a connector" on page 168.

#### 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 343.

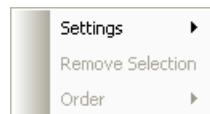
Right-clicking **on a component** in the **component browser** presents a context menu with the choices in the figure to the right in the figure below. For more information on the alternatives, please see section "Context menu for components in the diagram layer and the component browser" on page 346.

**Examples of context menu from a component in package and component browser.**

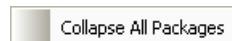


**Examples of context menu from an empty area the component browser.**

Right-clicking **in an empty area below the components** in the **component browser** presents a menu. Please see section “Context menu: Component browser” on page 343 for more information.



Right-clicking **the header “Packages”** in the package browser displays a context menu that will collapse all packages in the package browser if selected.



## Presentation in package browser

### What is not shown

Protected classes are by default not shown in package browser.

### 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.

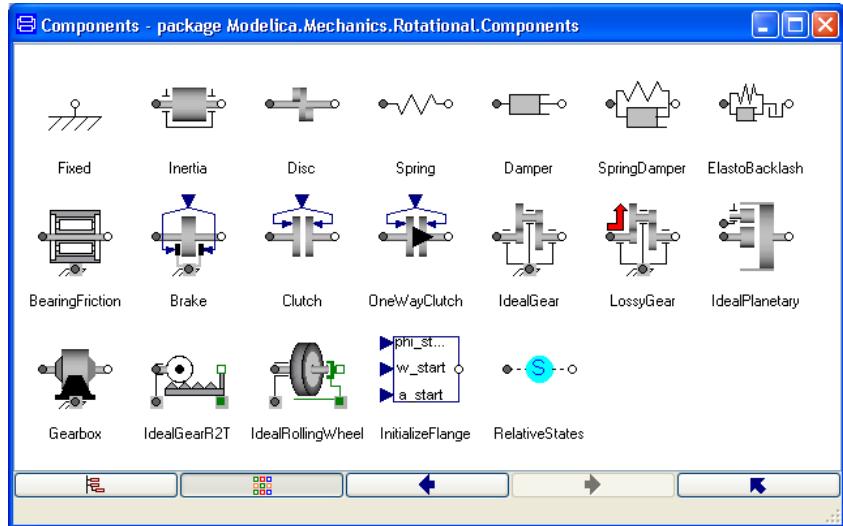
#### 4.1.4 Library window

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 **Window > 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.**



#### 4.1.5 Command window

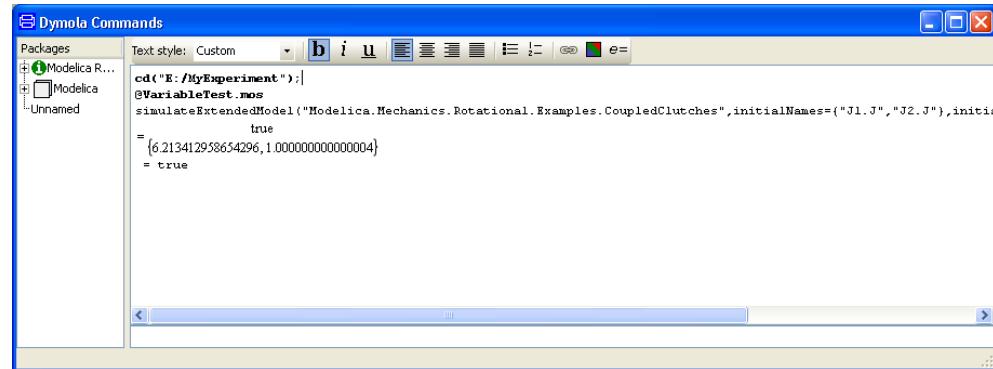
The Command window is used for entering commands to Dymola.

It is not shown by default in Modeling mode but can be made visible in two ways:

- As a sub-window of Dymola Main window (this is default in Simulation mode) by using the command **Window > Tools > Commands**.
- As a free-standing window by using the command **Window > 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.

## **4.1.6 Message window**

The message window will not be displayed by default in **Modeling** mode, but will pop up when an error (e.g. a syntax error) is detected. Since the window is much more used in **Simulation** mode, please see this chapter for more information.

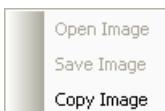
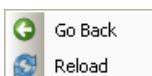
## 4.1.7 Information browser

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 screenshot shows a Windows-style window titled "Modelica.Mechanics.Rotational.Components". The main content area displays the "Ideal planetary gear box" component. It includes a green-bordered icon of a planetary gear assembly. Below the icon is a detailed description of the component's function and behavior. A mathematical formula  $zp := (zr - zs) / 2$  is shown to calculate the number of planet teeth. The "Parameters" section contains a table with one row: "ratio" (number of ring\_teeth/sun\_teeth (e.g. ratio=100/50)). The "Connectors" section contains a table with four rows: "sun" (Flange of sun shaft), "carrier" (Flange of carrier shaft), "ring" (Flange of ring shaft), and "ring" (Flange of ring shaft). The toolbar at the top of the window includes standard icons for back, forward, stop, reload, copy, and print.

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.

## 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 235.

### 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 or 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.

From Dymola 2015 FD01 infinite diagram/icon layer is supported; moving works independently of zoom factor, and scroll bars are not shown.

Previously the zoom factor had to be larger than 100 % for moving to work (otherwise the whole diagram or icon layer was shown anyway). When the zoom factor was larger than 100% also scroll bars appeared that could be used to move the view. This old behavior can be retained by setting the flag

```
Advanced.InfiniteDiagramLayer = false
```

The flag is **true** by default.

##### Zooming

To zoom, there are four different options:

- Press and hold the **Alt** key and span a rectangle<sup>1</sup>. 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.

---

<sup>1</sup> On Linux, **Ctrl+Shift** have to be used instead of **Alt**.

A convenient way of returning to 100% of zooming is to right-click an empty area, having nothing selected, and select **Fit to Window** from the context menu.

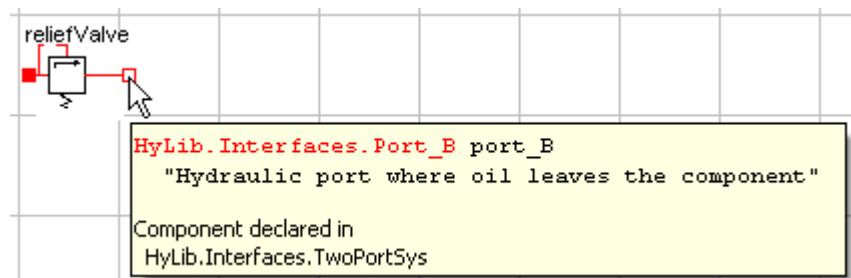
## Dynamic tooltips

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 an object opens a dialog with additional information. For components and connectors several attributes may be changed, see “Parameter dialog – advanced” on page 268.

See also “Edit > Select All” on page 309.

### 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 in Modeling mode are presented as sub-sections in the section “Editor command reference – Modeling mode” starting on page 291.

### 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 **Edit > Attributes** – see “Graphics tab” on page 317). If the **Ctrl** key is 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).

Connections are automatically adjusted after a move operation to make sure that the end points still reach the corresponding connectors. “Edit > Manhattanize” can be used to clean up skewed connections (see page 312).

### 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.

Common reshaping operations are also available in the context menu of the object or in the **Edit** menu, for example, rotation and flipping horizontally and vertically (see page 312 and 343).

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 351.

## **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.

## **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).

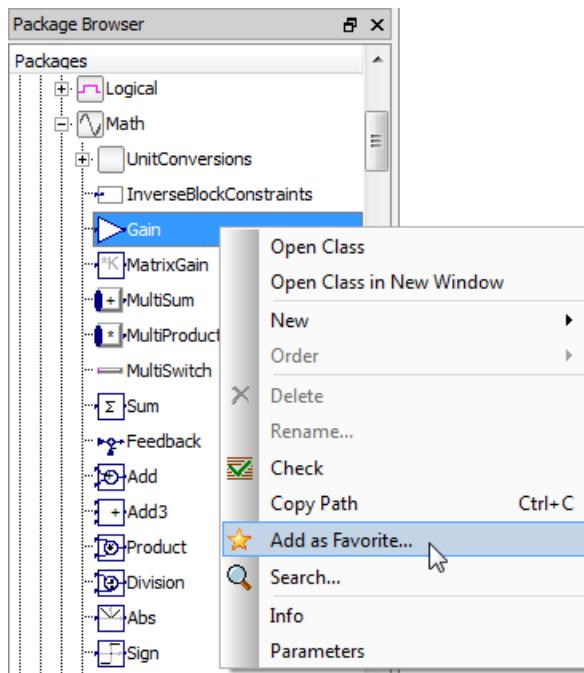
For a simple example of creating a library, see the chapter “Getting started with Dymola”, section “Using the Modelica Standard Library”, subsection “Creating a library for components”.

The command has options to create partial packages (“template packages”) and to extend (“reuse”) packages. See “Advanced model editing” starting on page 235 for information on such more advanced features.

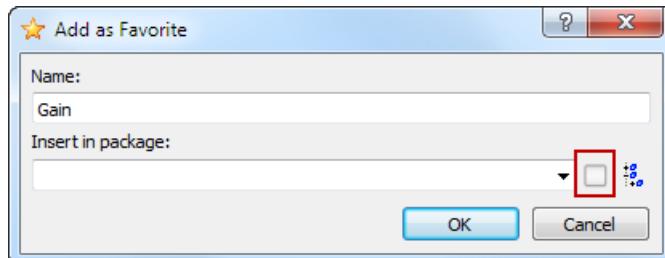
#### **Creating a favorite package**

When working with inserting a number of components from different libraries, a useful feature is to create a “favorites” package, where you can organize commonly used classes to get easy access to them.

It is possible to create a package containing “favorite” classes, by right-clicking on a class in the package browser, and selecting **Add as Favorite...**,



This opens a simplified class creation dialog:



Here you can select name and package. To create a new package, click the marked icon. This opens the normal Create New Package dialog. The newly created package gets preselected after creation.

To add a currently open class as a favorite, right-click the background of the diagram layer and select **Add as Favorite....**

As an example, consider a package of favorites: **Favorites**.

The **Favorites** package looks like an ordinary package, but behaves differently:

- When you open the **Favorites** package, 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.

Note that there is a more advanced way of creating favorite packages, possibly with prefilled parameter values when creating components from the classes. See “Dymola User Manual Volume 2”, 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.

### **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 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 235 for information on such more advanced features.

### **Creating a model by extending an existing module**

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.

A simple example how to extend a model is given in 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.

For more advanced use, see “Advanced model editing” starting on page 235 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 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 235 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.

### **Duplicating a class**

You can duplicate a class 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.

Note. If you want to reuse a class, it is usually better to extend from it.

### **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.

## 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 152.

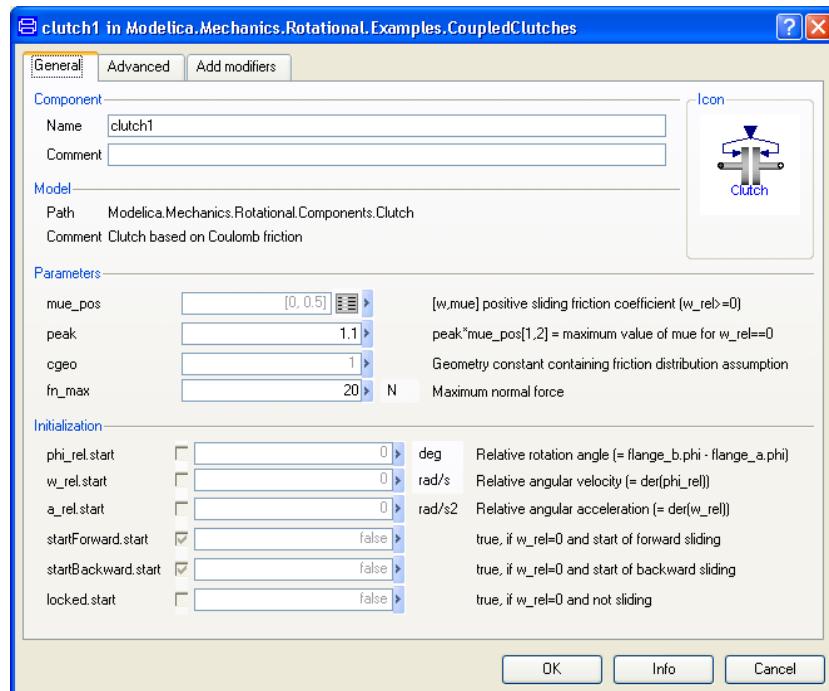
### Editing name and comment

Name and comment can be edited in the parameter dialog<sup>2</sup>. This dialog is reached either by double-clicking on the component or by selecting **Parameters...** in the context menu of the component. The parameter dialog can look like:

---

<sup>2</sup> Please note that changing the name of a component does not by default update references to it. (This may be enabled by setting Advanced.ActivateSmartRename=2.)

## Parameter dialog.

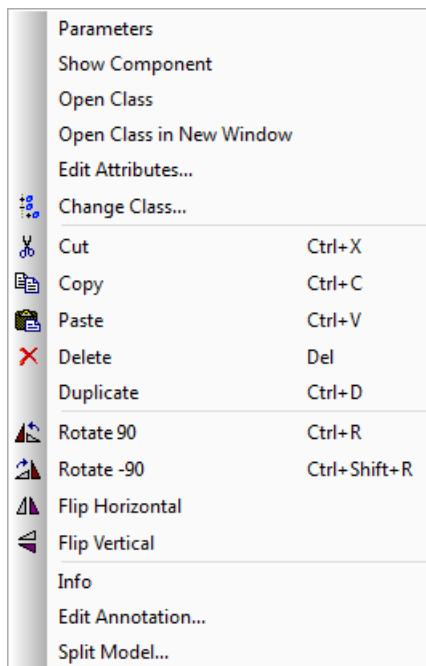


In the **General** tab the **Component** group gives the possibility to edit the name of the component and insert a comment (if the component or connector is not read-only and provided the component or connector is found at the top-level in the component browser). The **Icon** group shows the icon of the component, while the **Model** group shows the path to it and the comment to the base class.

## 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 343.

### Editing parameters and variables

Below the use of the parameter dialog is used. Note that initial conditions for certain parameters and variables also can be changed by saving values from the simulation directly into the model, see section “Programming in Simulate mode – saving start values in model” on page 213 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 261.)
- Replaceable classes in the model. Please see the section “Parameter dialog – advanced” on page 268 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 268 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.

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.**

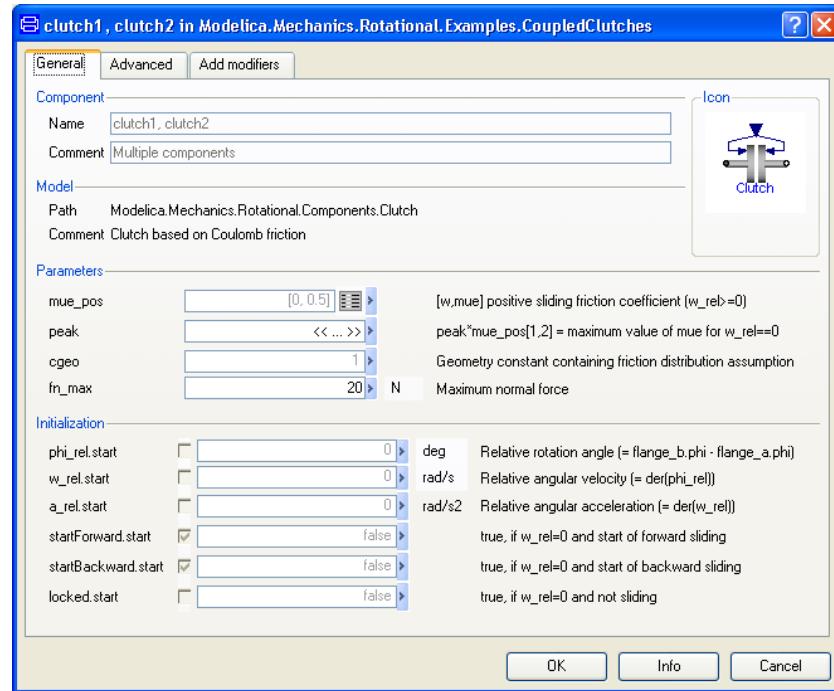
The parameters of the parameter dialog can be structured at two levels. The first level consists of tabs which are displayed on different pages of the window. The second level is a framed group on the page. Different variables with the same tab and group attributes are placed together.

The user can define tabs when declaring variables. See section “Annotations tab” on page 267.

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.

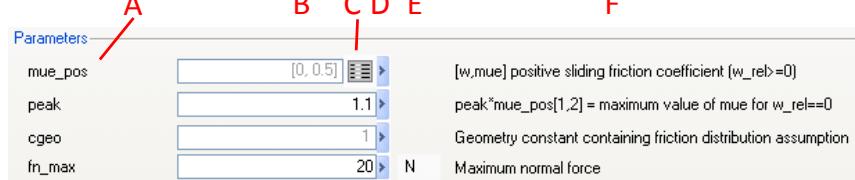
#### Model group

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. Also the class-specific comment is shown.

#### Parameters group

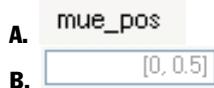
The parameters group lists the parameters in the component. From left to right, a number of “columns” can be seen

### 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:



The *name* of the parameter is the first column.

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 – compare the next parameter).

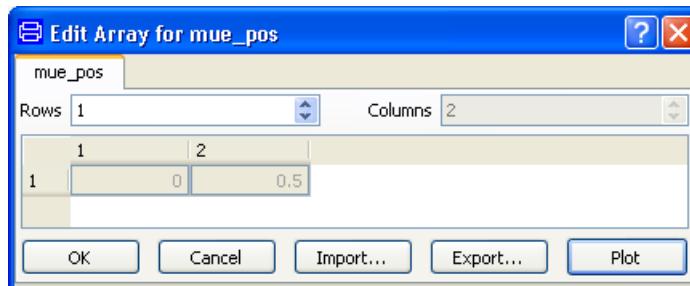
In some cases a drop-down menu is available to select between values. It should be noted that it is possible to enter values without using the drop-down menu. This enables the use of expressions, for example.

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.



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):

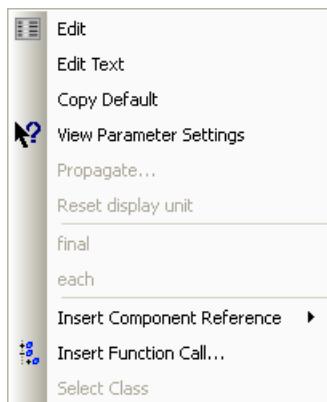
### Window from the edit button.



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 357.



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 362 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.

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 279.

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:



	A	B	C	D	E	F
Initialization						
phi_rel.start	<input type="checkbox"/>	<input type="text" value="0"/>	deg	Relative rotation angle (= flange_b.phi - flange_a.phi)		
w_rel.start	<input type="checkbox"/>	<input type="text" value="0"/>	rad/s	Relative angular velocity (= der(phi_rel))		
a_rel.start	<input type="checkbox"/>	<input type="text" value="0"/>	rad/s <sup>2</sup>	Relative angular acceleration (= der(w_rel))		
startForward.start	<input checked="" type="checkbox"/>	<input type="text" value="false"/>		true, if w_rel=0 and start of forward sliding		
startBackward.start	<input checked="" type="checkbox"/>	<input type="text" value="false"/>		true, if w_rel=0 and start of backward sliding		
locked.start	<input type="checkbox"/>	<input type="text" value="false"/>		true, if w_rel=0 and not sliding		

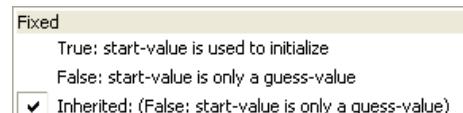
(indicated by the letters A-F also in the side head below). Let us use the first start value in the figure above (phi\_rel.start) as an example:

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.

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.

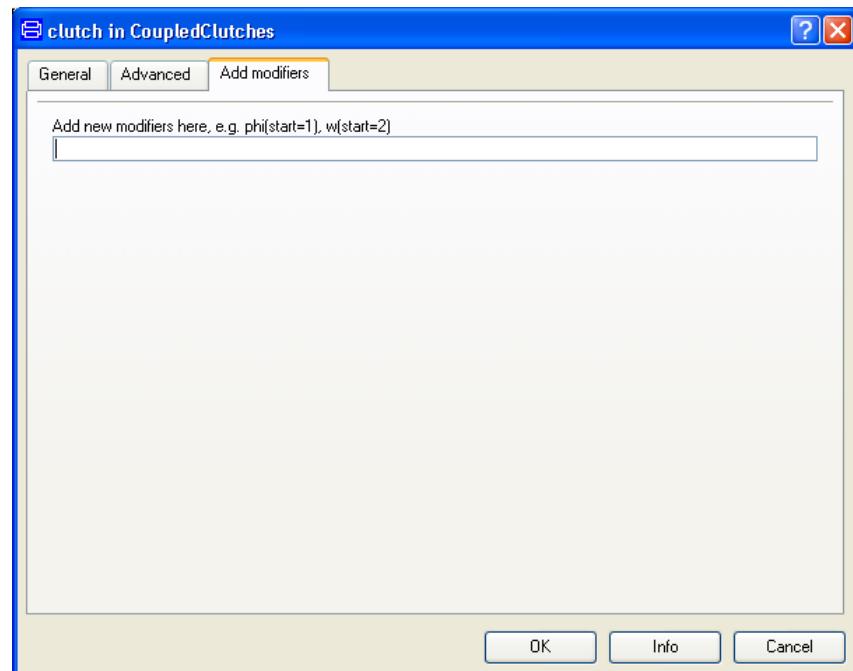
Finally a comment to the parameter will form the last column:

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 269.

## 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.

## 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.

**Info** displays the Modelica documentation for the class using an information browser:

**Modelica.Mechanics.Rotational.Components**

**Modelica.Mechanics.Rotational.Components.IdealPlanetary**

**Ideal planetary gear box**

**Information**

The IdealPlanetary gear box is an ideal gear without inertia, elasticity, damping or backlash consisting of an inner **sun** wheel, an outer **ring** wheel and a **planet** wheel located between sun and ring wheel. The bearing of the planet wheel shaft is fixed in the planet **carrier**. The component can be connected to other elements at the sun, ring and/or carrier flanges. It is not possible to connect to the planet wheel. If inertia shall not be neglected, the sun, ring and carrier inertias can be easily added by attaching inertias (= model Inertia) to the corresponding connectors. The inertias of the planet wheels are always neglected.

The icon of the planetary gear signals that the sun and carrier flanges are on the left side and the ring flange is on the right side of the gear box. However, this component is generic and is valid independently how the flanges are actually placed (e.g. sun wheel may be placed on the right side instead on the left side in reality).

The ideal planetary gearbox is uniquely defined by the ratio of the number of ring teeth  $z_r$  with respect to the number of sun teeth  $z_s$ . For example, if there are 100 ring teeth and 50 sun teeth then ratio =  $z_r/z_s = 2$ . The number of planet teeth  $z_p$  has to fulfill the following relationship:

$$z_p := (z_r - z_s) / 2$$

Therefore, in the above example  $z_p = 25$  is required.

According to the overall convention, the positive direction of all vectors, especially the absolute angular velocities and cut-torques in the flanges, are along the axis vector displayed in the icon.

**Parameters**

Name	Description
ratio	number of ring_teeth/sun_teeth (e.g. ratio=100/50)

**Connectors**

Name	Description
sun	Flange of sun shaft
carrier	Flange of carrier shaft
ring	Flange of ring shaft

Depending on the complexity of the class, the documentation can contain pictures, tables, references etc.

If no html documentation file is available for the class (e.g. if the class has been copied without generating any documentation), the text part of the original class documentation is shown in a separate window, without any pictures.

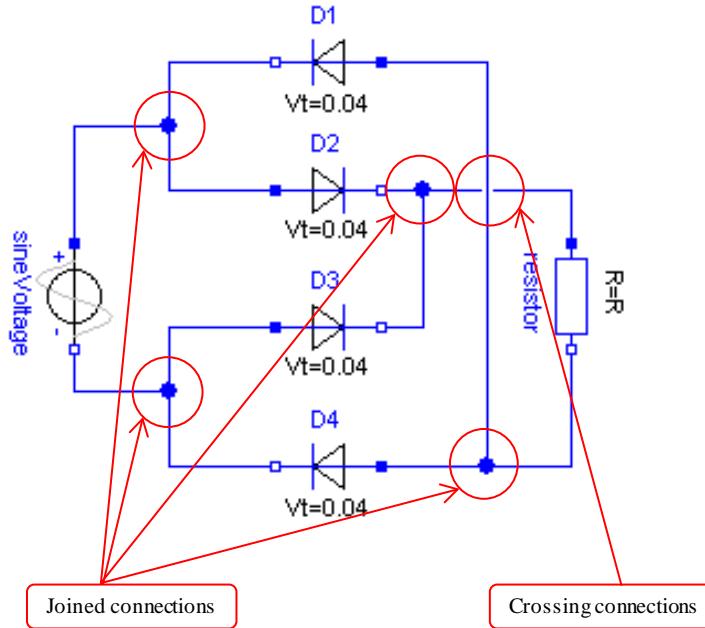
#### 4.2.4 Connections



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.

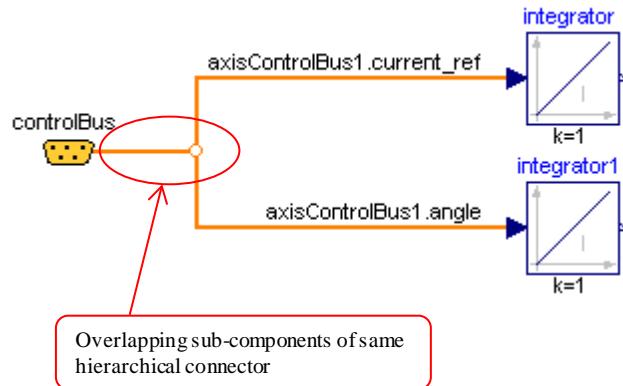
If connections are joined (more than one connection to the same connector) this will be graphically represented by a circular solid “blob”. Connections crossing each other will be indicated by a tiny space in the crossing for one connector.

**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 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.

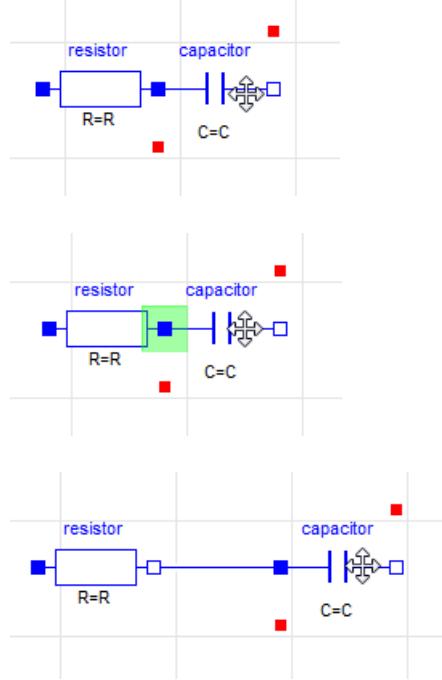


These features are also disabled when **Connect Mode** is not selected.

Working with the features, **Ctrl+Z**, or **Escape**, are the general commands used to undo an action.

### Smart connection

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:

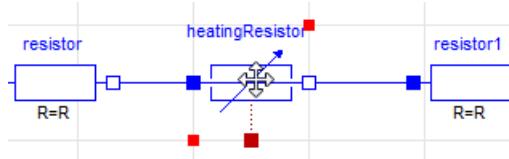
- Green: Connection succeeded.
- Yellow: It is possible to create a connection, but not with smart connect. This is the case with e.g. expandable connectors and array of connectors.
- Red: Connection not possible; e.g. connectors are not matching.

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.

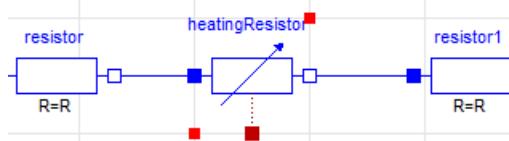
### Drop component on connection

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.

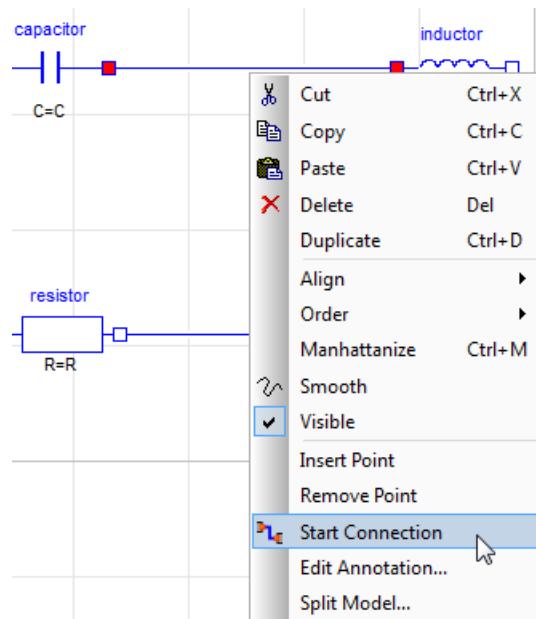
The connection is split only if the relevant sub connectors of the dropped component match the connection (in the above example the positive and negative pin).

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**

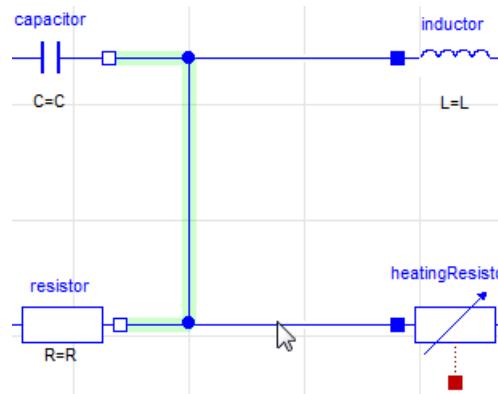
A connection can now be started from an existing connection, and a connection can be ended on an existing matching connection.

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 185). 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 with **Edit > 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.

If automatic manhattanize is enabled (using **Edit > Options...** and ticking, in the **Appearance** tab, **Automatic Manhattanize of connections**) connections are manhattanized immediately when created, moved or reshaped. If automatic manhattanize is on, 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.

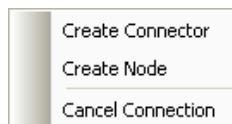
When drawing a connection, the default color is taken from the starting connector. The connection gets the color of the outline of the first rectangle, polygon or ellipse in the icon layer of the starting connector. This change facilitates color-coding of connections depending on the type of connector.

In the graphical editor, connection lines are drawn *over* components and other graphical objects. This means that connections are not hidden by mistake.

### Warning!

Note that drawing an ordinary line between two connectors (see line drawing below) does *not* create a connection!

## 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 350.

## 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

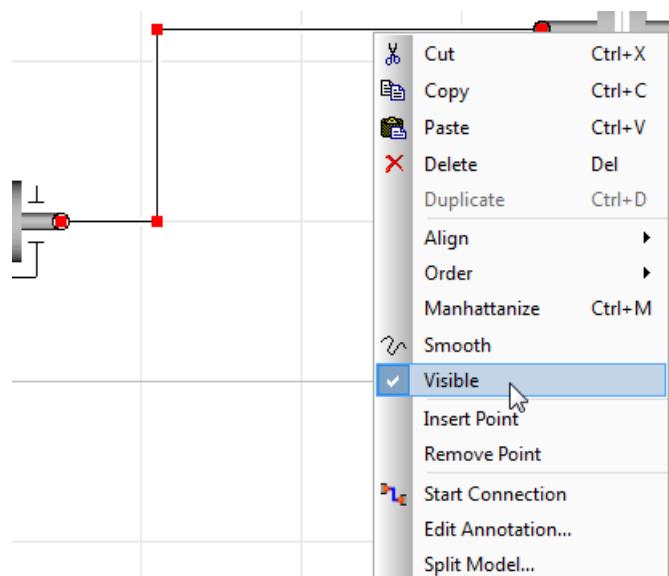
message displays the declarations of the two connectors. The connection can then be cancelled.

- 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.

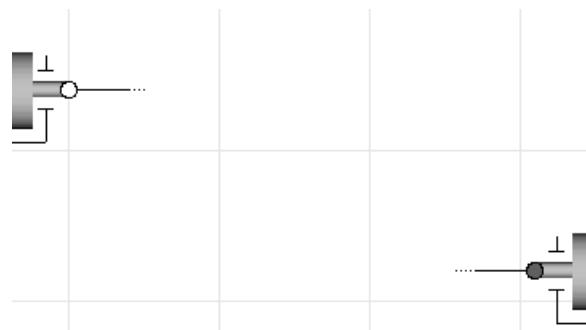
### Hiding graphical connections

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 **Edit > Options...**, in the **Appearance** tab. See section “Appearance tab” on page 319. By default such objects are hidden.

Accidental hiding is corrected using **Edit > 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 drawing toolbar. The corresponding toolbar button is shown in a depressed state while the drawing operation is in progress.

The first 6 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 is used for toggling of the grid. The two last buttons are used for drawing connections and perform check. They are not described here.

**The drawing toolbar.**



The graphical tools described in this section (corresponding to the six first buttons in the toolbar 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 very convenient, since it contain most options for editing the object.

### Lines and polygons



Click on the **Line** or **Polygon** button in the toolbar, 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.

**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 in the toolbar and draw the shape in an edit window. 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



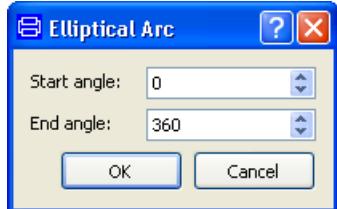
Click on the **Ellipse** button in the toolbar and draw the shape in an edit window. Ellipses are drawn to touch the bounding box at four points.

An ellipse could be made circular by keeping the **SHIFT** button pressed while drawing the object.

It is possible to create circle sectors. Do the following:

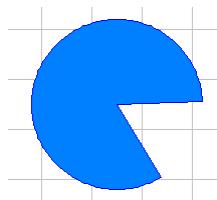
Draw a circle. Color it. Select **Elliptical Arc...** from the context menu. The following menu will appear:

#### **Edit angles menu.**



Changing the values to 0 – 300 will give:

#### **Examples of circle segments.**



Changing the values to 300 – 360 will give:



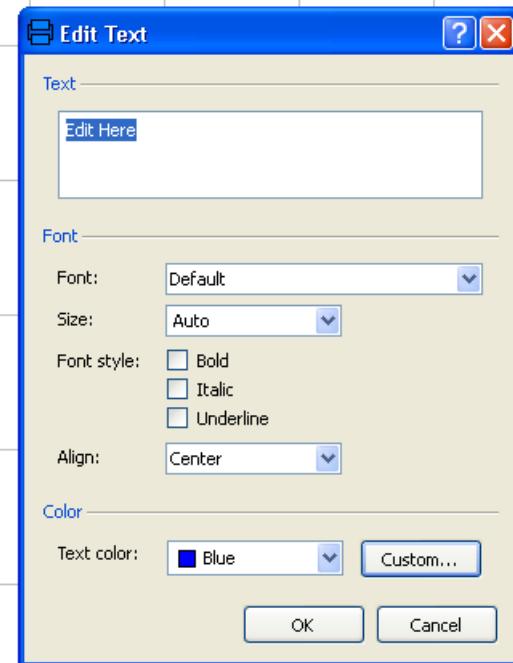
## **Text**

**A**

Click on the **Text** button in the toolbar, and draw the text object in an edit window, 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.

## Edit Here



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.

**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 section “Edit > Options...” on page 319, 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:

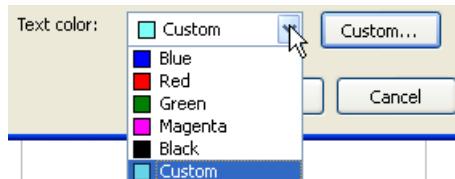
- %class Name of the enclosing class.
- %name Name of the component this text is enclosed in.
- %path The full path name of the component.
- %par The value of parameter named par in the model.
- %=par The value of parameter named par in the model, printed as par=value.

## 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 “Graphical object menus: Line and fill style” starting on page 353.

### “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).

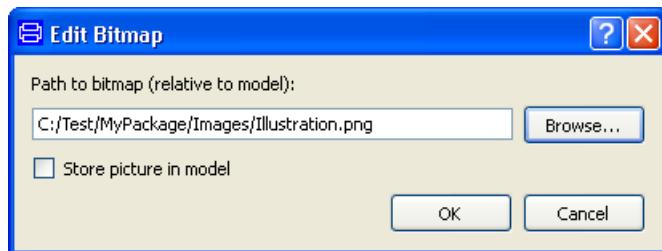
### Bitmap



Draws a bitmap which is read from an external file. Click on the **Bitmap** button in the toolbar, draw the outline in an edit window and then specify the bitmap filename. Supported file formats are BMP, GIF, JPEG and PNG. 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.

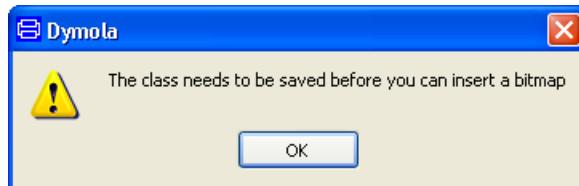
Using the recommended location for images above for an image “Illustration.png” for a top package “MyPackage”, the URI

```
modelica://MyPackage/Images/Illustration.png
```

will be the resulting URI in the annotation. This annotation can be seen by selecting the image, right-clicking and selecting **Edit 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** 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. However, options like color and filling must be changed using separate commands. A general command **Edit** is available, containing some more options – please note however that this command does *not* contain all possibilities covered by the context menu.

Please also note that editing using **Edit Annotation** works in a specific way. See below. This menu is by default displayed for most graphical object when double-clicking the border.

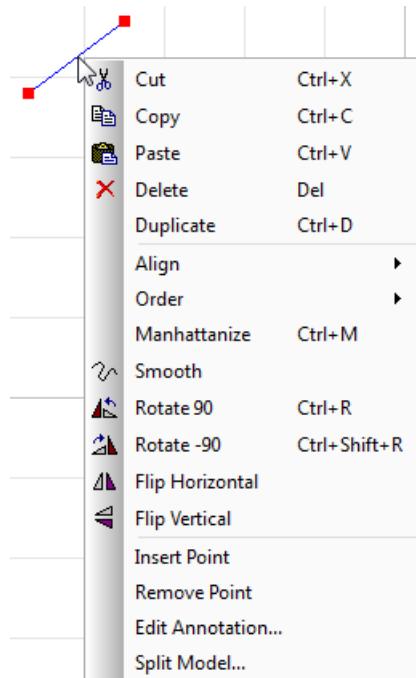
## Menus used for editing

### 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 351.

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 line.**

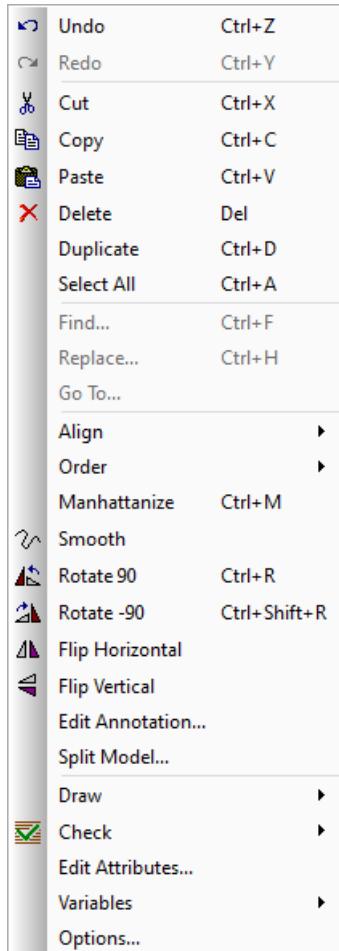


Please note that the context menu includes specific options (e.g. corner radius in the context menu for a rectangle and elliptical arc for the context menu of an ellipse).

**Edit Annotation...** makes it possible to edit graphical annotations by a menu, in a specific way. Please see section “Editing graphics using Edit Annotation” on page 281.

## Edit menu

The **Edit** menu.



The **Edit** menu contains a lot of entries that can be used to edit graphical objects. This menu is described in section “Main window: Edit menu” starting on page 308.

The **Edit** menu does *not* include all options available for a specific graphical component (e.g. corner radius in the context menu for a rectangle and elliptical arc for the context menu of an ellipse).

The last section of the **Edit** menu cannot be reached from the context menus.

### Line style and Fill style



The user must first select one or more primitive graphical objects, and then pick one of the style buttons on the toolbar 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 section “Graphical object menus: Line and fill style” starting on page 353.

## 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 introductory chapter “Getting started with Dymola” in this manual 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 using the command **Help > Documentation**. More information is also available at [www.modelica.org](http://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 click on the **Modelica Text** button in the toolbar of the upper part of the Dymola main window. That will open the Modelica Text layer for that class.

Doing this for Modelica.Mechanics.Rotational.Components.Gearbox reveals the following:

The screenshot shows the Dymola interface with the Modelica Text layer open. The title bar reads "Gearbox - Modelica.Mechanics.Rotational.Components.Gearbox (Read-Only) - [Modelica Text]". The menu bar includes File, Edit, Simulation, Plot, Animation, Commands, Window, Help. The toolbar has various icons for file operations. The left side features a Package Browser and a Component Browser. The Package Browser lists packages like BearingFriction, Brake, Clutch, OneWayClutch, IdealGear, LossyGear, IdealPlanetary, and Gearbox. The Component Browser lists components under Modelica.Mechanics.Rotational.Component... such as PartialTwoFlangesAndSupport, lossyGear, and elastoBacklash. The main area displays the Modelica text for the Gearbox class:

```
model Gearbox "Realistic model of a gearbox (based on LossyGear)"
  extends Modelica.Mechanics.Rotational.Interfaces.PartialTwoFlangesAndSupport;

  parameter Real ratio(start=1)
    "transmission ratio (flange_a.phi/flange_b.phi)";
  parameter Real lossTable[;, 5]=[0, 1, 1, 0, 0]
    "Array for mesh efficiencies and bearing friction depending on speed (see docu of LossyGear)";
  parameter SI.RotationalSpringConstant c(final min=Modelica.Constants.small, start=1.0e5)
    "Gear elasticity (spring constant)";
  parameter SI.RotationalDampingConstant d(final min=0, start=0)
    "(relative) gear damping";
  parameter SI.Angle b(final min=0) = 0 "Total backlash";
  parameter StateSelect stateSelect=StateSelect.prefer
    "Priority to use phi_rel and w_rel as states" a;

  Modelica.SIunits.Angle phi_rel(start=0, stateSelect=stateSelect, nominal=1e-4)
    "Relative rotation angle over gear elasticity (= flange_b.phi - lossyGear.flange_b.phi)";
  Modelica.SIunits.AngularVelocity w_rel(start=0, stateSelect=stateSelect)
    "Relative angular velocity over gear elasticity (= der(phi_rel))";
  Modelica.SIunits.AngularAcceleration a_rel(start=0)
    "Relative angular acceleration over gear elasticity (= der(w_rel))";

  equation
    phi_rel = flange_b.phi - lossyGear.flange_b.phi;
    w_rel = der(phi_rel);
    a_rel = der(w_rel);
  end Gearbox;
```

The content of the Modelica Text layer differ 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 symbols “+” in the sidebar to the left of the code. If expanded, the symbol will disappear and the “+” will turn into a “-“ and a corresponding line indicating the extension of the expansion.

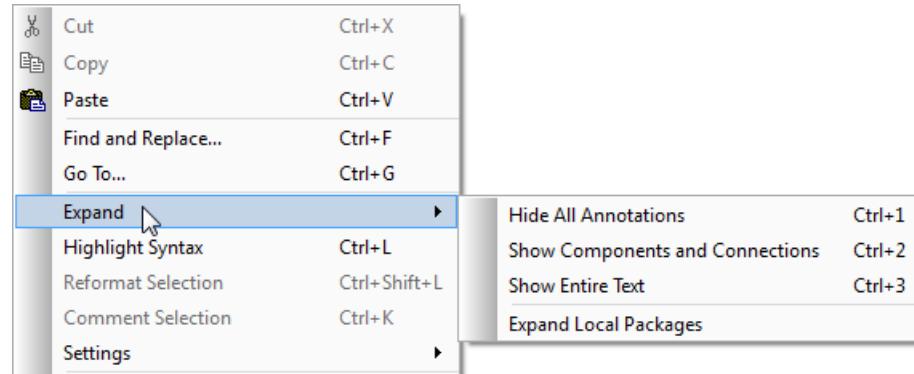
A screenshot of the Modelica IDE interface. The title bar says "Inertia - Modelica.Mechanics.Rotational.Components.Inertia (Read-Only) - [Modelica Text]". The main window shows the Modelica code for the Inertia component. The code includes imports, parameters, annotations, and equations. The left sidebar shows the Package Browser with "Inertia" selected, and the Component Browser with "flange\_a" and "flange\_b" listed under "Modelica.Mechanics.Rotational.Component...".

```
model Inertia "1D-rotational component with inertia"
import SI = Modelica.SIunits;
Rotational.Interfaces.Flange_a flange_a "Left flange of shaft" a;
Rotational.Interfaces.Flange_b flange_b "Right flange of shaft"
annotation (Placement(transformation(extent={{90,-10},{110,10}},
rotation=0)));
parameter SI.Inertia J(min=0, start=1) "Moment of inertia";
parameter StateSelect stateSelect=StateSelect.default
  "Priority to use phi and w as states" a;
SI.Angle phi(stateSelect=stateSelect) "Absolute rotation angle of component" a;
SI.AngularVelocity w(stateSelect=stateSelect)
  "Absolute angular velocity of component (= der(phi))" a;
SI.AngularAcceleration a
  "Absolute angular acceleration of component (= der(w))" a;
a;
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;
```

In the example above the Connection annotation is expanded, revealing the flanges. The annotation of Flange\_b is in turn expanded, revealing the placement annotation for that flange.

Expanding can either be done individually for each annotation, by clicking on the + sign (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).

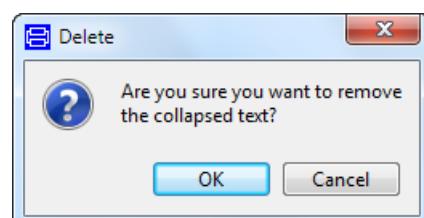
Collapsing can either be done individually for each expansion by clicking on the corresponding “–“ sign, or collectively using any of the context menu commands mentioned.

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.

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. Top-level annotations can be edited in the Modelica Text layer. Annotations of components (e.g. variables) can be edited the same way, but when it comes to annotations of variables, using the variable declaration dialog is more convenient (see above, about editing variables).

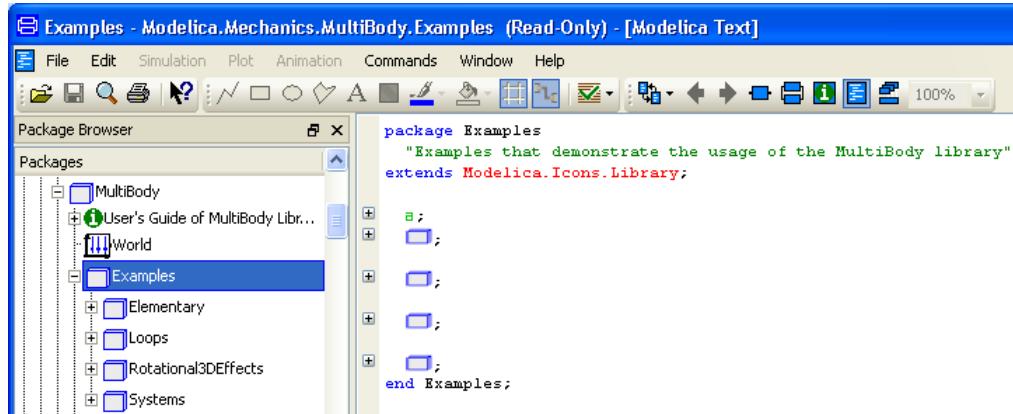
Code containing collapsed item (component, connection, annotation or local package) can be selected and deleted; a question will be displayed whether to do it:



## Local packages

Local packages are packages defined in another package. In the figure below, the four last symbols are local packages; they correspond to the four packages under Electrical in the package browser:

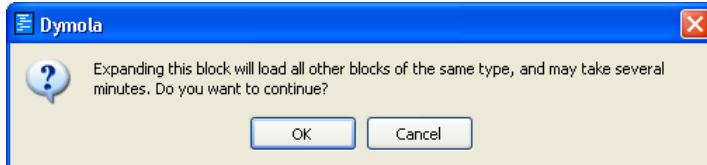
### Local packages.



Local packages are by default only shown as symbols in 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 symbol or “+”.
2. Click on a “+” 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.

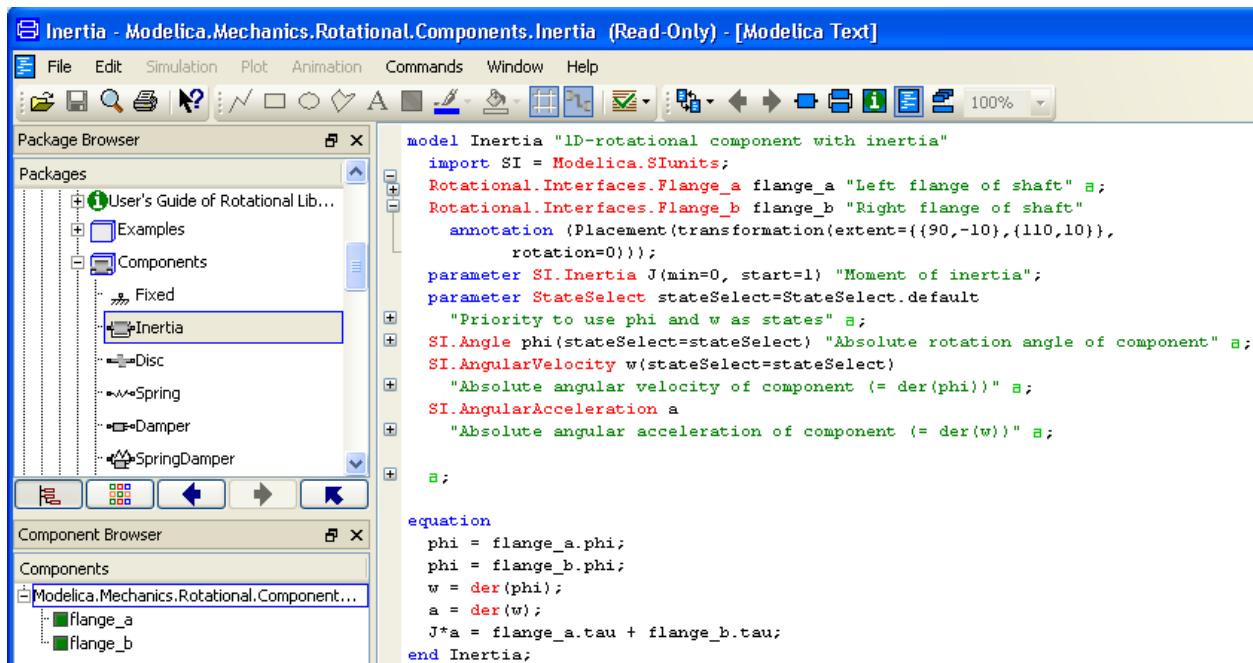
## Navigation in the Modelica Text layer

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.



```

model Inertia "1D-rotational component with inertia"
import SI = Modelica.SIunits;
Rotational.Interfaces.Flange_a flange_a "Left flange of shaft";
Rotational.Interfaces.Flange_b flange_b "Right flange of shaft";
annotation (Placement(transformation(extent={{90,-10},{110,10}},
rotation=0)));
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))";

φ = flange_a.φ
φ = flange_b.φ
w =  $\frac{d\varphi}{dt}$ 
a =  $\frac{dw}{dt}$ 
J·a = flange_a.τ + flange_b.τ

end Inertia

```

The mathematical notation mode is selected by right-clicking to pop the context menu and then clicking on **Use mathematical notation**. (The mode is left by unchecking this entry.)

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 in the equation part 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.

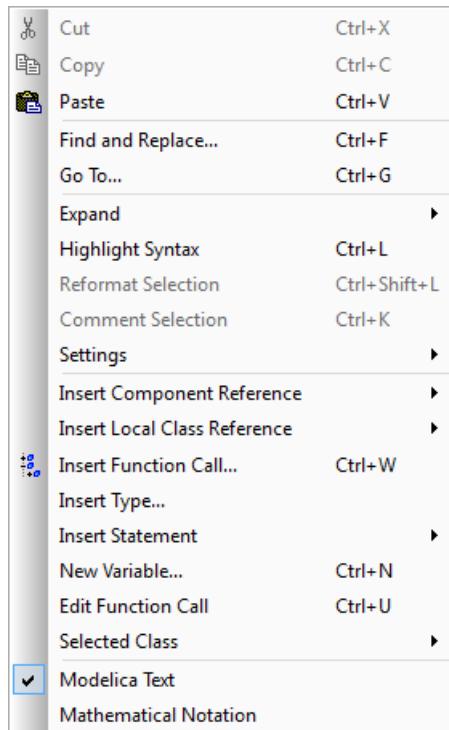
More about copying and other selections available in the context menu of the editor is described in the section “Context menu: Edit window (Modelica Text layer – mathematical notation)” starting on page 340.

## 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
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

## 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 335.

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).

- Click on the **Check** button (or select **Edit > Check** 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.

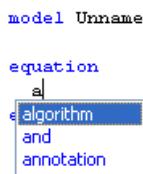
HTML text in annotation will have automatic color coding of tags as well. Please see section “Documentation” below.

### Code completion

Code completion can be activated by pressing **Ctrl+Space** in the Modelica Text Layer. This will bring up a context menu containing all the words beginning with the letters written so far or all 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.

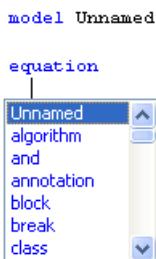
#### Code completion example.

```
model Unnamed
equation
  a|
  algorithm
  and
  annotation
```


 A screenshot of a code editor showing the start of a word 'a'. A dropdown menu is open, listing several words: 'algorithm', 'and', and 'annotation'. The word 'algorithm' is highlighted with a blue selection bar.

The code completion menu contains all the words from the current Modelica Text Layer, all Modelica keywords (highlighted in blue) as well as all the words from models previously shown in the Modelica Text Layer. 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.

```
model Unnamed
equation
  |
  Unnamed
  algorithm
  and
  annotation
  block
  break
  class
```


 A screenshot of a code editor showing the start of a word 'a'. A dropdown menu is open, listing several words: 'algorithm', 'and', 'annotation', 'block', 'break', and 'class'. The word 'algorithm' is highlighted with a blue selection bar. A vertical scroll bar is visible on the right side of the menu.

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, either click on the word in the menu or navigate to the word with the arrow keys and press **Enter** to input the currently selected word. Closing the menu without inputting any word can be done by pressing **Escape** or clicking outside of the menu.

## 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.

## Selection of user-defined or automatic text formatting

The user can select whether the text should be shown according to the formatting implemented by the author of the class, or whether automatic formatting should be used. This selection is done the following way: Right-click and put the cursor on **Settings**. When the setting **Manual formatting** is ticked (default), the formatting of the class author is used. Note that the default of this setting can be changed, and saved between sessions, using the command **Edit > Options...**, the **Appearance** tab. See section “Edit > Options...” on page 319.

## Adjustable maximum line length

A dotted vertical line is present in the editor to indicate the maximum line length. It can be dragged 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 default value of the max line length in the Modelica text editor can be changed by the **Edit > Options...** command, the **Appearance** tab. See the section “Appearance tab” starting on page 319.

## Other text formatting

Apart from the features mentioned above, the user can select any part of the text, right-click and select **Reformat Selection** 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 empty lines.
- 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 **Edit > Options...** command and temporary changing the **Base font size** in the **Appearance** 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 195 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 195 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

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. 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.

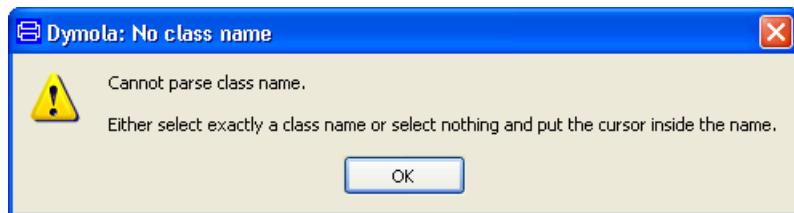
Right-click and select **Selected Class**. The following menu will be the result:

#### Navigation in Modelica Text editor.



It is now possible to open the selected class in the present window, 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:

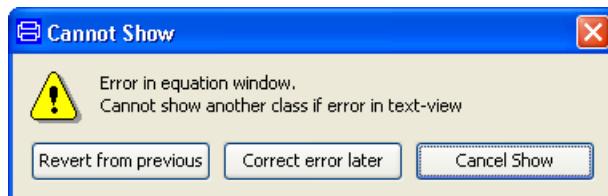
- Right-click and select **Highlight Syntax** from the context menu (or use the shortcut **Ctrl+L**). The command will also automatically color-code the existing code, please see above.
- Click on the **Check** button (or select **Edit > Check** or press **F8**). This will check more than the syntax, 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.
- 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, the Modelica Text icon in the upper *left* part of the main window will be red.

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 shown.

**Cancel Show** will cancel the action – the present class will still be displayed.

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 the command **Edit > Check**) 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, a setting **Pedantic mode for checking Modelica semantics** is available in Simulating mode, using the command **Simulation > Setup...**, the **Translation** tab. 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 **New Variable...** in the Modelica Text layer
- By using the command **Edit > Variables > New Variable....**
- 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....**

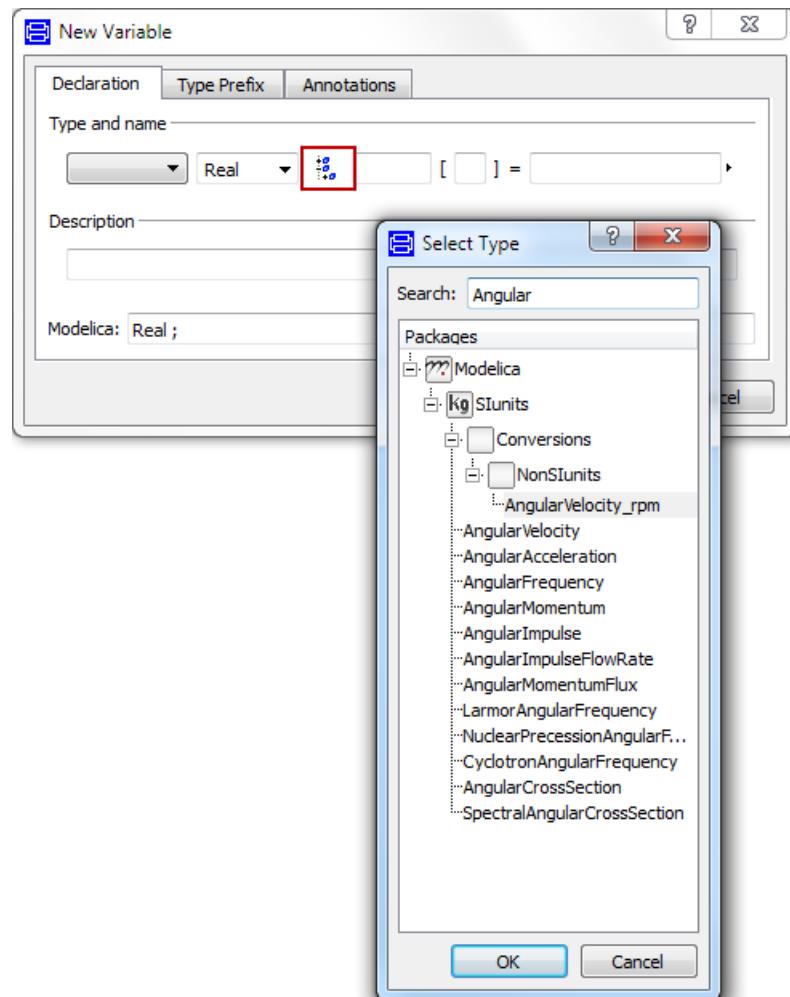
The first three alternatives use the variable declaration dialog, the last one text.

The second and third alternatives are described in detail in section “Declaration of parameters, variables and constants”, starting on page 262.

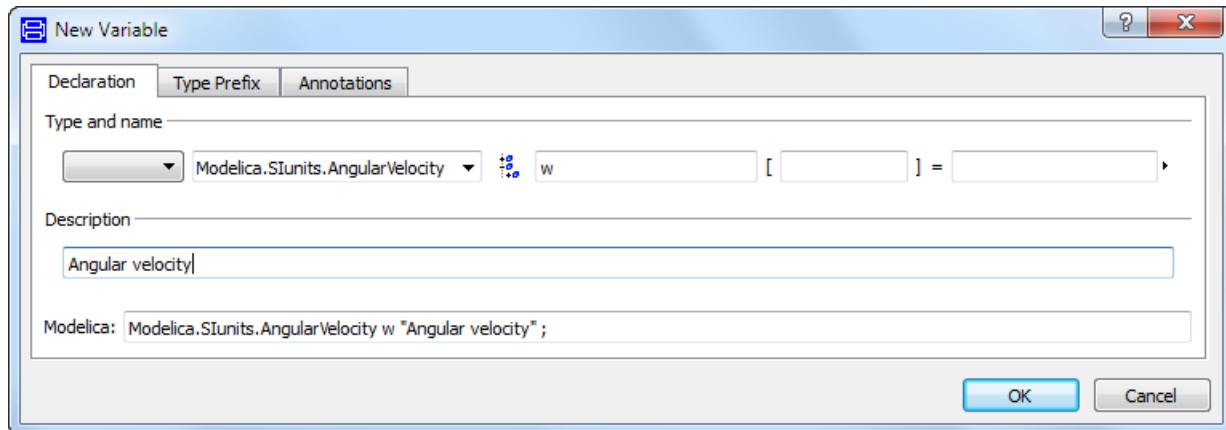
Using any of the three first alternative about for declaring a new variable, the result will be a variable declaration dialog (with some pre-defined data if dragging is used).

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 264“.

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 "Length 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);
  + b;
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 use the command **Edit > Variables** and then select the variable. This will pop the variable declaration dialog for that variable.

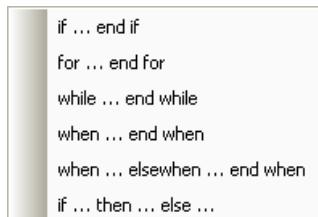
Variables can of course be edited directly by typing in the Modelica Text layer.

For more information about the variable declaration dialog, please see section “Variable declaration dialog” starting on page 264. For more information about variables in general, please see section “Parameters, variables and constants” starting on page 261.

## 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.

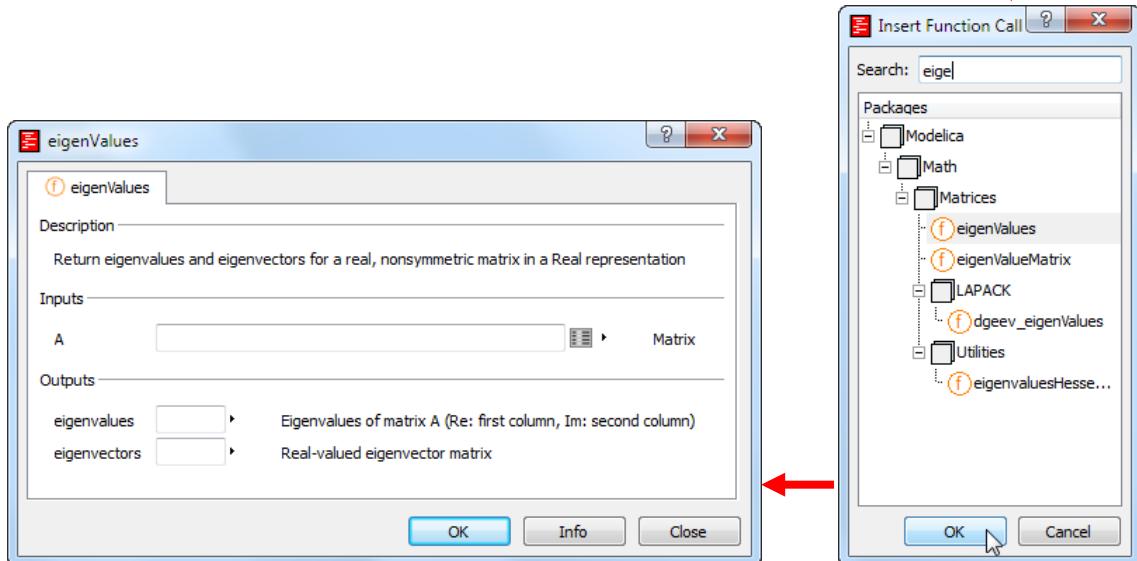
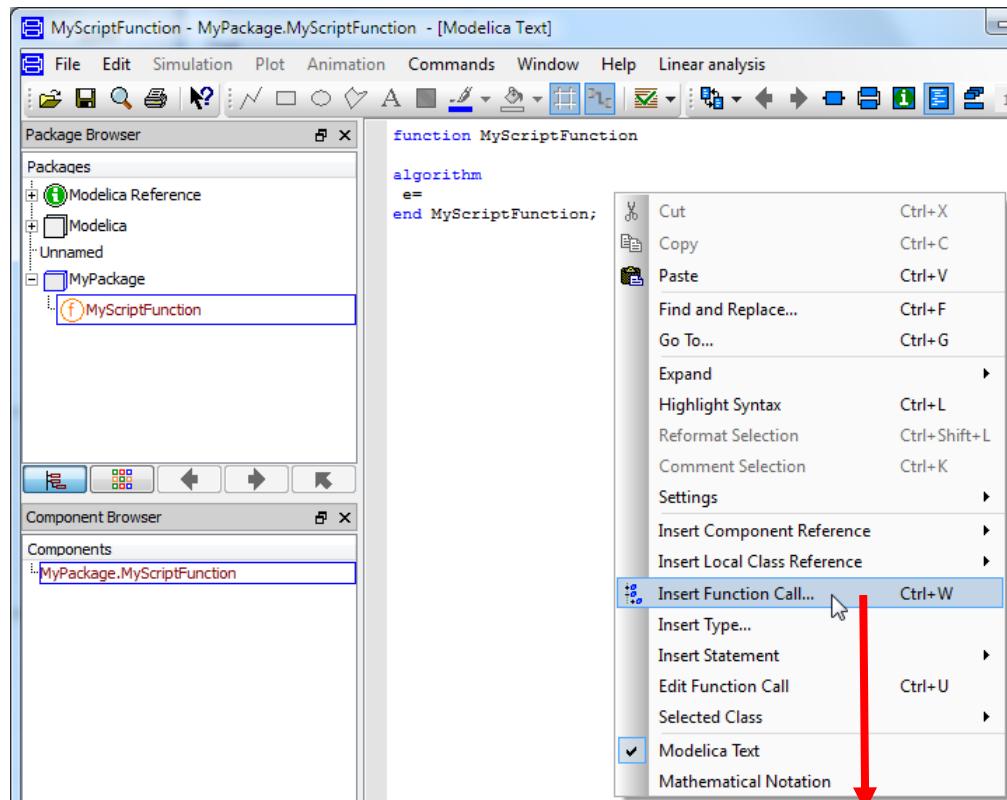


## Handling of function calls

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 in Simulate mode – saving start values in model**

Modified initial conditions and parameter values entered in the variable browser in Simulation mode 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. Refer to next chapter for more information.

There are some limitations; initial values of the following parameters cannot be saved this way:

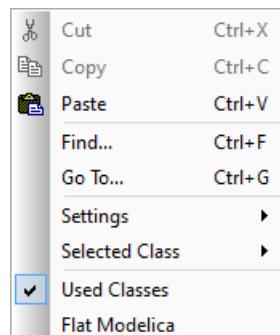
- Evaluated parameters.
- Final parameters.
- Protected parameters.
- Parameters in array of components.
- Parameters being part of bounded record components.

## **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 a model/block).

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 341.

### **Displaying Used Classes**

This is the default setting of the Used Classes layer (see the image of the context menu above).

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 a model or a block is selected in the package browser, it is possible to show the Flat Modelica text of that model/block. This is done by using the context menu and tick **Flat Modelica**.

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;
  + b;
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 re-declared classes.

Keywords are in blue, comments in green, other items are black. (Presently types and operators are not red but black.)

Displaying a model/block 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* a model or a block, Used Classes will be displayed, in spite of Flat Modelica being selected.

It is not possible to show a model/block that contains errors as Flat Modelica text. When trying to display e.g. such a model, the following text will be shown:

```
//Errors was detected when making a flat Modelica description.  
//Please, check the model and correct the errors.
```

## 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. 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.

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.

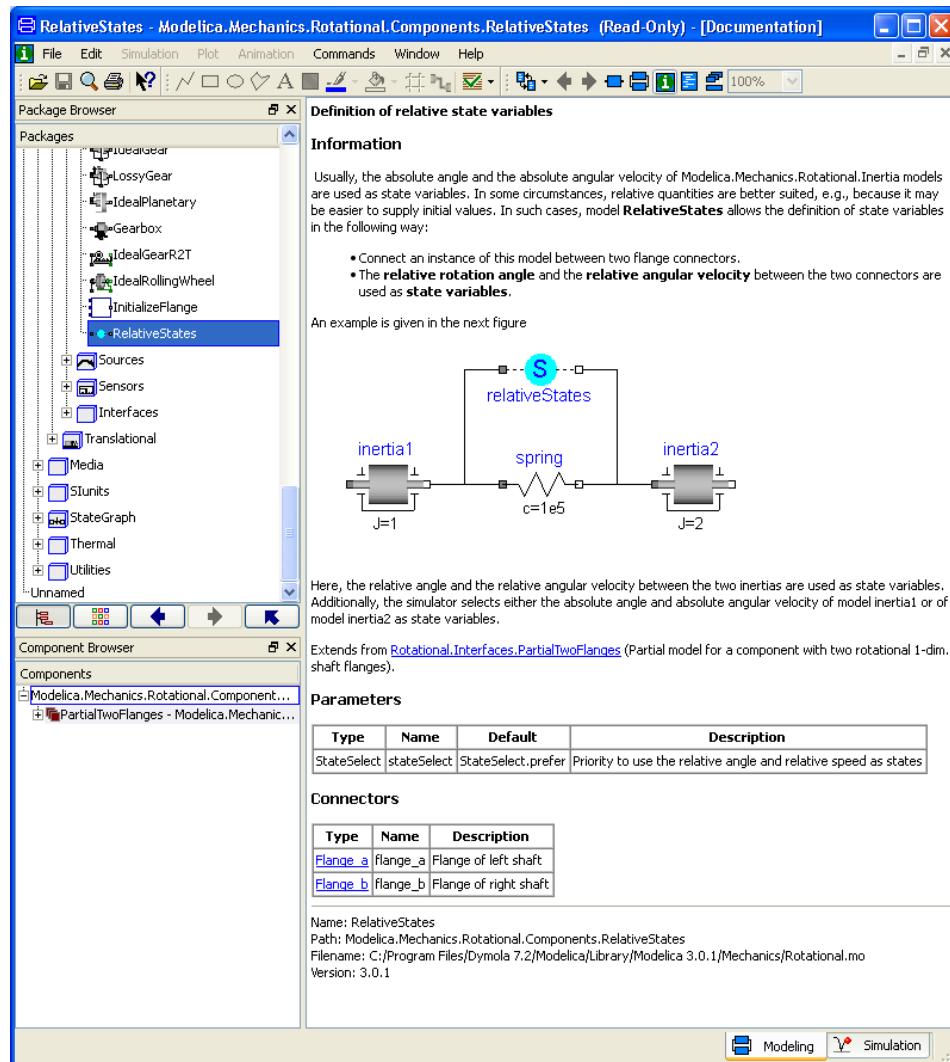
The three last items have been dealt with previously. For more information about this, please see the description of these features. The first two items will be treated here.

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. 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 of an edit window.**



The first part “**Linear 1D rotational spring**” (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.

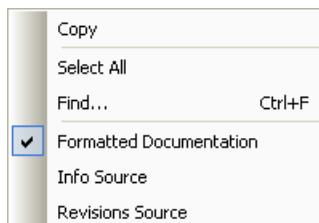
(Sometimes custom sections might be available, e.g. “**Release Notes**”.)

The last part of the documentation is the **Revision** documentation. 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.

By right-clicking in the formatted documentation layer, a context menu is available. The Show alternative looks different depending on if the documentation layer is read-only or not.

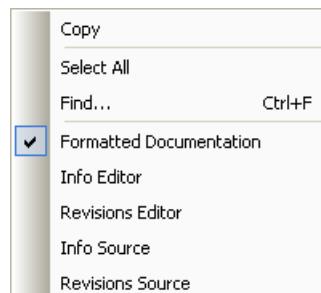
If read-only, the HTML code of the information part and revision part can be looked at separately.

**Context menu for read-only documentation layer.**



If not read-only, the information part and the revision part of the documentation layer can also be edited using a documentation editor.

**Context menu for  
editable  
documentation layer.**

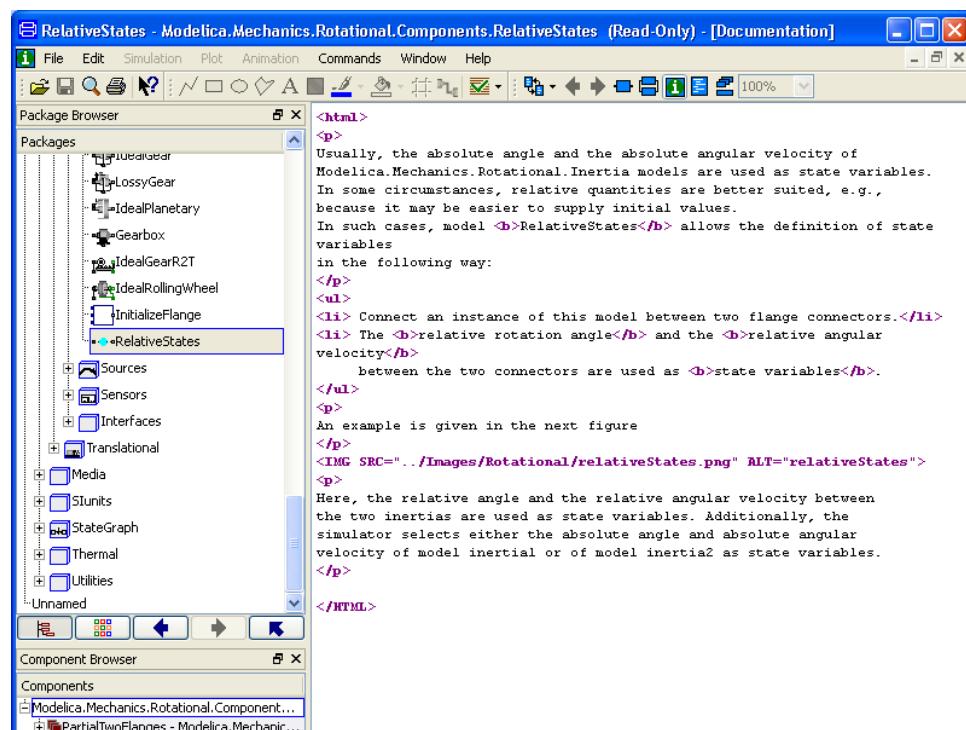


More information about the context menu can be found in the section “Context menu when viewing formatted documentation” on page 333.

### 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 **Info source** and **Revisions source**, respectively, the HTML source code can be seen.

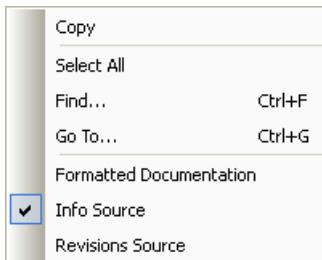
**The HTML source code  
of the information part  
of the documentation  
layer.**



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 the following if the text is read-only:

**Context menu for read-only HTML info source code.**



For more information, please see section “Context menu for read-only HTML source code for info/revisions part” on page 335.

To go back to the formatted documentation, right-click and select **Formatted documentation**.

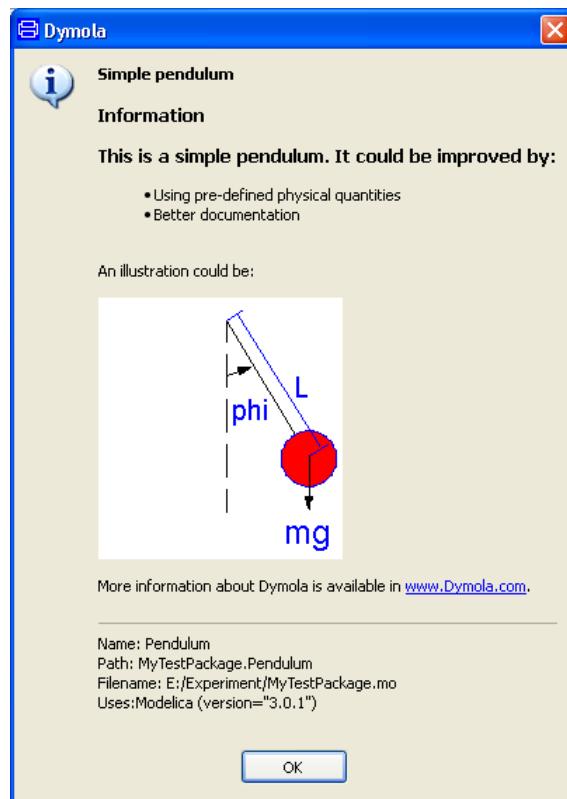
**Looking at the documentation layer (and more) using the Info button/menu entry**

**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 information part of the documentation layer is displayed, plus some info about filename, path etc.

**Info on class that has no generated HTML file.**



This also means that if a model has been extended, and no additional documentation has been added, only the last part of the information in the figure above is shown (Name, Path, Filename and Uses). It is worth noting that not even the information that the model is extended from another model is visible if the information part of the documentation level does not contain anything.

If HTML files have been generated, the result might look like (the size of the window has been adapted to show all info):

**Info on class with generated HTML file.**

**MyTestPackage**

**MyTestPackage.Pendulum**

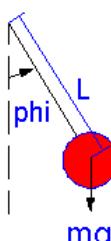
**Simple pendulum**

**Information**

This is a simple pendulum. It could be improved by:

- Using pre-defined physical quantities
- Better documentation

An illustration could be:



More information about Dymola is available in [www.Dymola.com](http://www.Dymola.com).

**Parameters**

Type	Name	Default	Description
Real	m	1	Mass of the pendulum
Real	L	1	Length of the pendulum
Real	g	9.81	Gravity of acceleration
Real	J	$m \cdot L^2$	Moment of inertia

**Modelica definition**

```
model Pendulum "Simple pendulum"
  parameter Real m=1 "Mass of the pendulum";
  parameter Real L=1 "Length of the pendulum";
  parameter Real g=9.81 "Gravity of acceleration";
  parameter Real J=m*L^2 "Moment of inertia";
  Real phi(start=0.1) "Pendulum angle";
  Real w "Angular velocity";

equation
  der(phi) = w;
  J*der(w) = -m*g*L*sin(phi);
end Pendulum;
```

By comparison it can be seen that much more information is generated if HTML documentation is 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 “File > Export... > HTML...” starting on page 300 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 selecting **Edit > Attributes...** from the menu bar. This command can also be used to change a number of other class attributes, see “Edit > Edit Attributes...” on page 316 for more information.

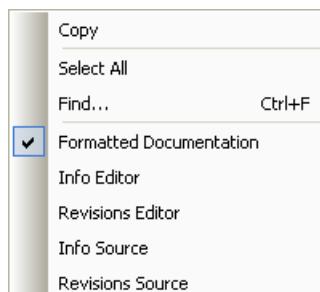
The second part of documentation, the longer information text (with the exception the auto-generated information about extending, if any), can be edited either using a documentation editor or directly in the HTML source code if not read-only. Please see separate sections about the documentation editor 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

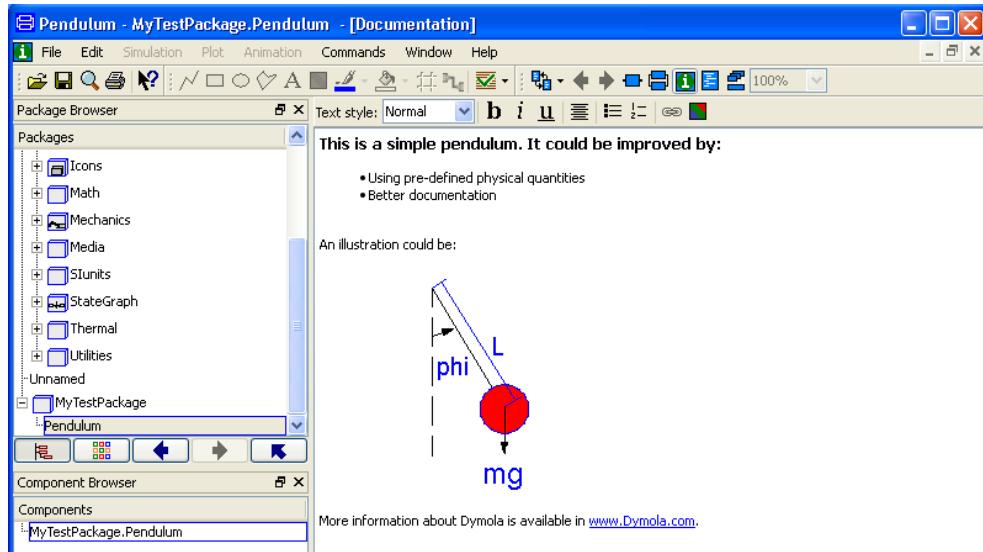
The documentation editor is reached in the documentation layer by right-clicking and selecting **Info editor** or **Revisions editor**, respectively (depending on what should be edited).



For more information about the entries in this menu please see section “Context menu: Edit window (Documentation layer)” on page 333.

Using the former might yield the following result:

**Example of documentation editor for info part.**



**Details of the documentation editor:**

The toolbar available:

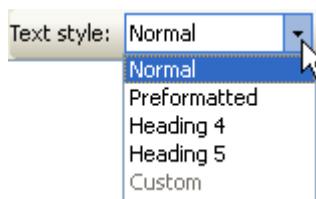
**Toolbar.**



Text style: Normal ▾

**Text style** contains presently four selections, **Normal** being default. **Custom** cannot be selected, but will be the result if selecting any text that has been changed in any way (e.g. bold, italics) compared to the other styles. **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.

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: **b**



The **Italic** button will change the text to italics; the **Underline** button will change the text to underlined. The **Center align** button will change the text to centered. They work the same way as the bold button when it comes to activation etc.



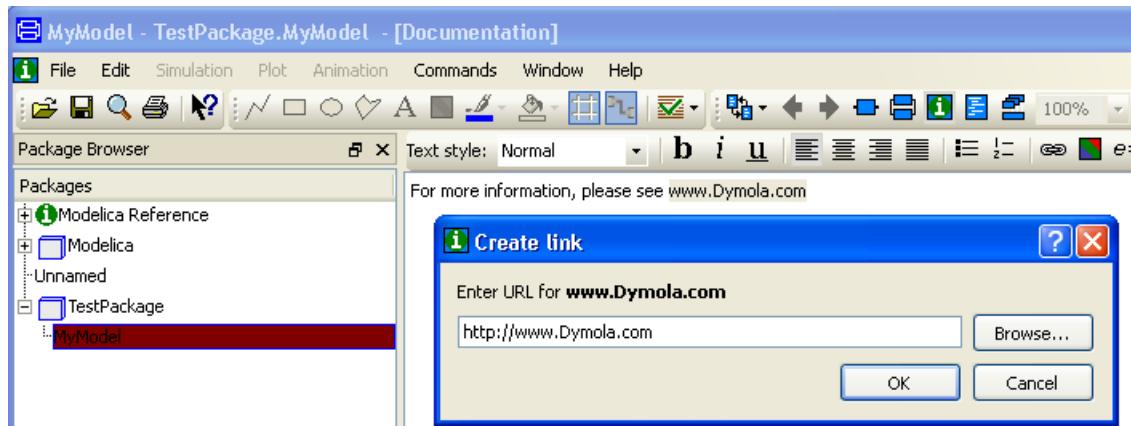
Four buttons are available for the alignment of selected text: they are in order **Left align**, **Center align**, **Right align** and **Justify**. The default is **Left align**; that button will be shown active by default.



**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 **Create link** button will open a menu for entering of an URI according to usual html rules. If a link is already present, that link will be shown.



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. 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.

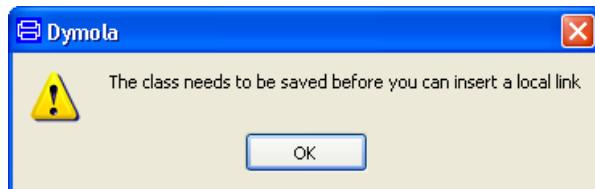
It is possible to select a formatted text when creating the link, as well as an image, and a mixing of text and image. Images will not be underlined when in links, however.

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 **Insert 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.

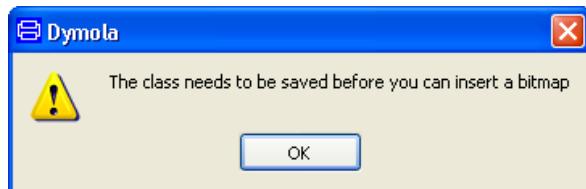
Using the recommended location for images above for an image “Illustration.png” for a top package “MyPackage”, the URI

```
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.)

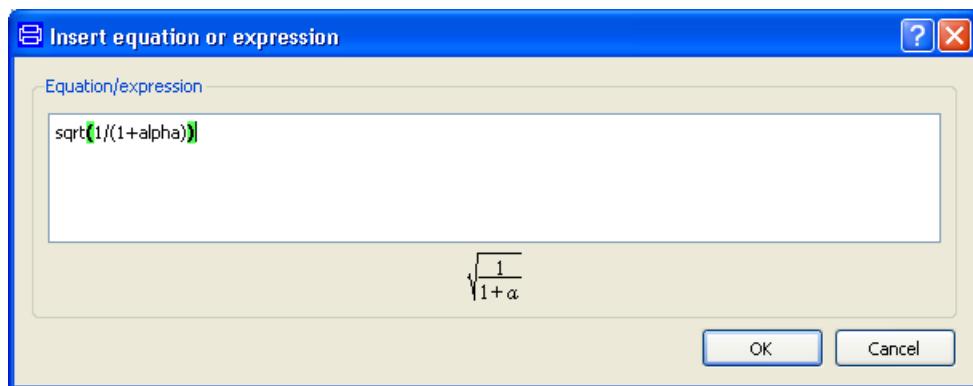
In order to be able to generate correct links, the model/package has to be saved (using e.g. **File > Save**) before an image can be inserted. If initial saving has not been done, a warning will be displayed:



If the link is invalid (image not found) it will be indicated with a symbol.

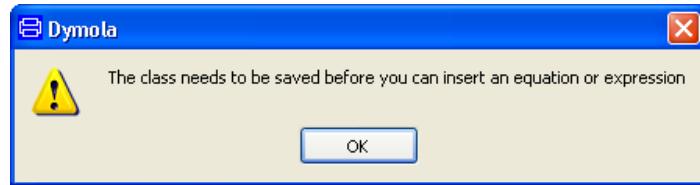
Clicking the button **Insert equation or expression** opens an editor for the creation of an equation/expression.

**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 **Insert equation or expression** button in the first place.

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. Taking up the context menu by right-clicking and selecting **Edit equation** 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 **Insert equation or expression** 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.)

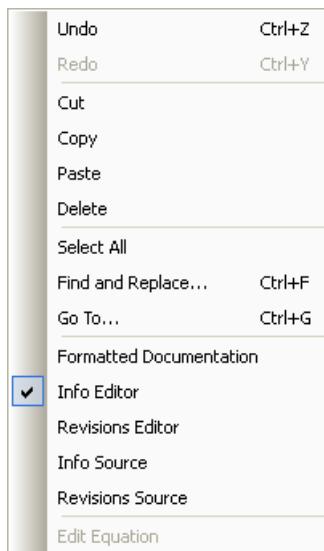
For some examples of the rendering of examples/expressions, please see next chapter, section “Model simulation”, sub-section “Documentation”.

Please note that it is still possible to enter simple expressions/equations by typing them in directly without using the button.

The revisions part of the information in the documentation layer can be edited using the same editor, but reached by right-clicking and selecting **Revisions editor**.

A context menu is available by right-clicking in the documentation editor:

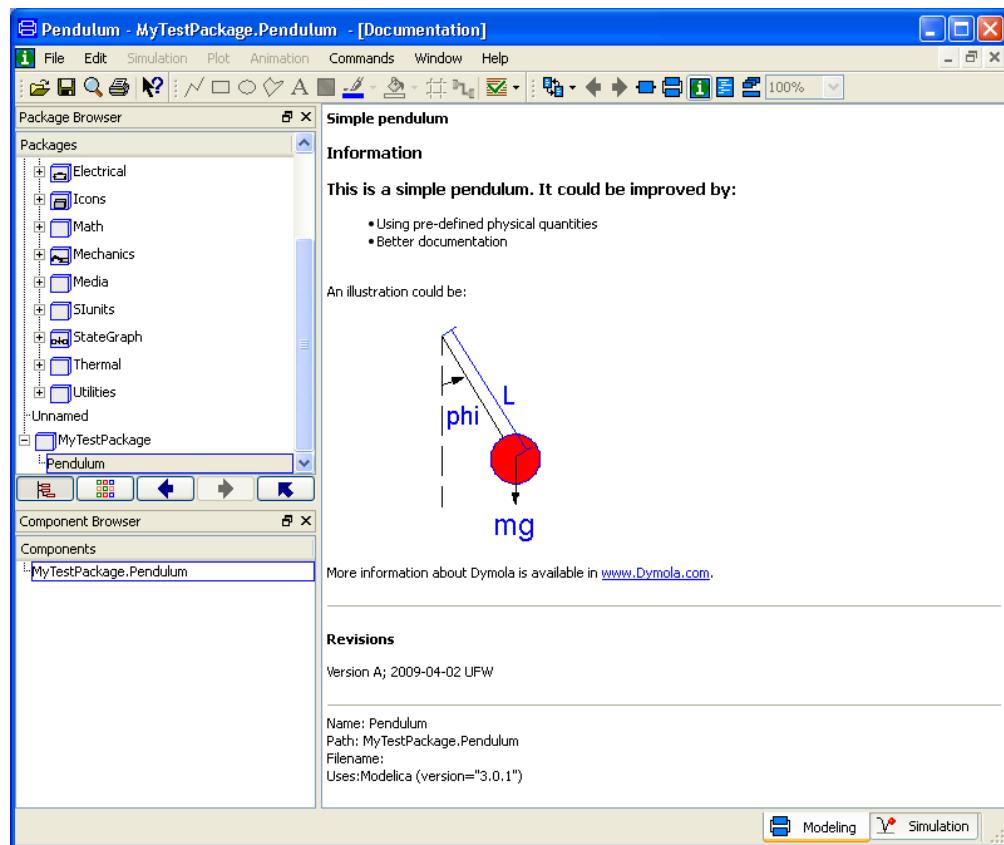
**Context menu of documentation editor.**



More information about this menu is available in the section “Context menu for editable info/revisions part” on page 333.

To go back to the formatted documentation, right-click and select **Formatted documentation**. 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 beside 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, e.g. tables.

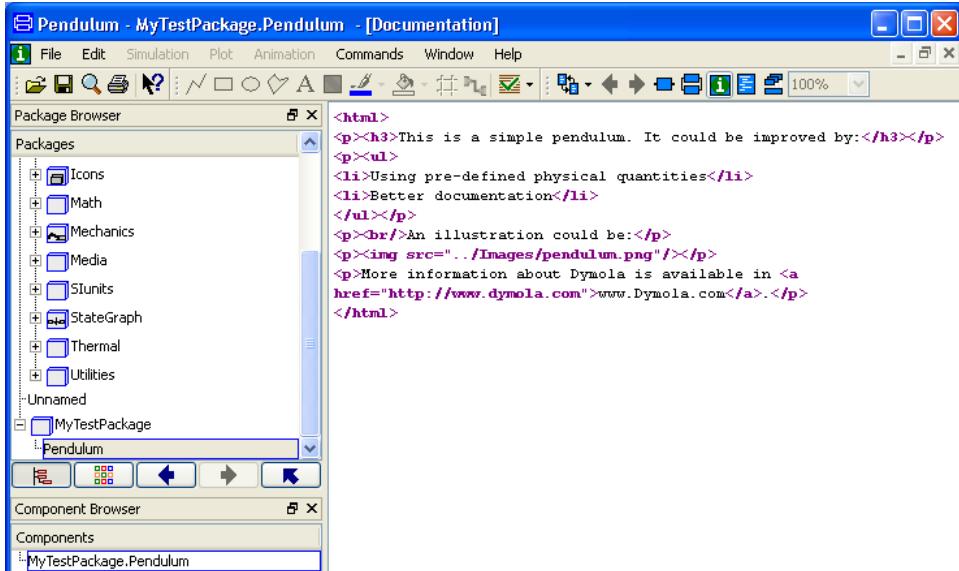
Please note that the documentation editor regenerates *all* HTML code when making *any* editing in it. This means that a construction that is not at all supported in the documentation editor (that is, not supported for rendering) will disappear completely when the HTML code is regenerated.

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.)

Tables are supported for rendering in the documentation editor. This means that tables can be constructed in the HTML editor, and then shown in the documentation editor. The cell content can be edited in the documentation editor, but no additional cells can be inserted – that must be done in the HTML source code.

The HTML source code for the information and revisions part of the documentation layer is reached (if the class is not read-only) by right-clicking and selecting **Info source** or **Revisions source** respectively. Selecting the formerly for the pendulum above will show:

**The corresponding html source code.**



The screenshot shows the Dymola Documentation Editor window titled "Pendulum - MyTestPackage.Pendulum - [Documentation]". The main area displays the following HTML code:

```
<html>
<p><h3>This is a simple pendulum. It could be improved by:</h3></p>
<p><ul>
<li>Using pre-defined physical quantities</li>
<li>Better documentation</li>
</ul></p>
<p><br/>An illustration could be:</p>
<p></p>
<p>More information about Dymola is available in <a href="http://www.dymola.com">www.Dymola.com</a>.</p>
</html>
```

The left side of the interface features a "Package Browser" pane listing packages like Icons, Math, Mechanics, Media, SUnits, StateGraph, Thermal, Utilities, and Unnamed, with "MyTestPackage" expanded to show its components. Below it is a "Component Browser" pane showing "MyTestPackage.Pendulum".

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, right-click and select **Formatted documentation**.

### 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 “File > Export... > HTML...” starting on page 300.

## **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. Please see page 306 for details (the setting **Generate HTML for referenced classes** in the **Advanced** tab reached by the command **File > 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 **File > Export... > HTML**. Please see the pages 301-309 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 (x86)\Dymola 2015 FD01\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 (x86)\Dymola 2015 FD01\Modelica\Library\Modelica 3.2.1\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      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 (UTF-8), 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.

In particular, the following has been tested for non-ASCII characters:

### **Using external files (read/write)**

- Modelica/.mos scripts in UTF-8. **Note**; a leading byte-order-mark is required.
- .mos scripts in UTF-16. **Note**; a leading byte-order-mark is required. (This was already supported in earlier Dymola versions since it is needed for scripting with non-ASCII file names.)
- Writing other files:
  - Encrypting Modelica files.
  - Save Total Model.
  - HTML-generation.

The reason for requiring a leading byte-order-mark in the files for UTF-8 and UTF-16 is that Modelica Standard Library and other libraries developed in Europe have European character for degree sign (°) and diacritical characters (ü å, ä, ö) that are outside of ASCII.

When Modelica/.mos scripts are created in Dymola, they are automatically created using UTF-16 encoding if any non-ASCII characters are present, otherwise using ASCII.

Creating and saving Modelica/.mos scripts using MS Notepad, a file containing non-ASCII characters must be saved using UTF-8 encoding.

### **The parameter dialog and the Edit > Variables command**

- Description strings (including the top ones for component and class).
- Tabs and groups.
- Choices descriptions.
- Tables.

### **The diagram layer**

- Texts.

- Tooltips for components.
- Connection lines with Unicode labels.

### **The text editor**

- Normal editing with e. g. comments.
- Mathematical notation.
- Insert equation with Unicode inside the equation.
- Tooltips for hidden parts.
- Storing text with errors.
- Error messages if using non-ASCII quoted identifiers will be displayed.

### **The description string for classes**

- As tooltip in the package browser (and icon view).
- Unicode characters can be used for new/duplicate model commands.
- Edit Attributes.
- Search (showing and matching).

### **The Commands menu**

- Description of commands.

### **Simulation and plotting**

- Description for initial values/plot selection.
- Description used in tooltips.
- Plot setup (also for legends, titles etc.)
- Plot scripts.

## 4.3 Advanced model editing

### 4.3.1 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:

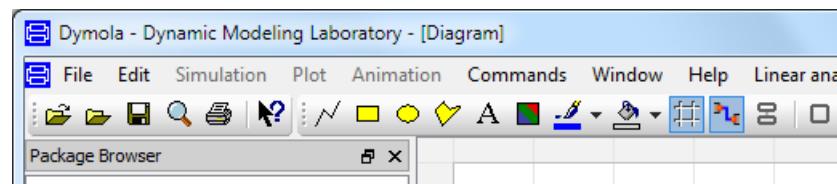
- For variables that are common between different states, the inner/outer concept has to be used, or alternatively graphical connections.
- State machines are clocked.

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 **Help > Documentation**, and the tutorial available by the command **File > Demos > Modelica Synchronous Tutorial**.

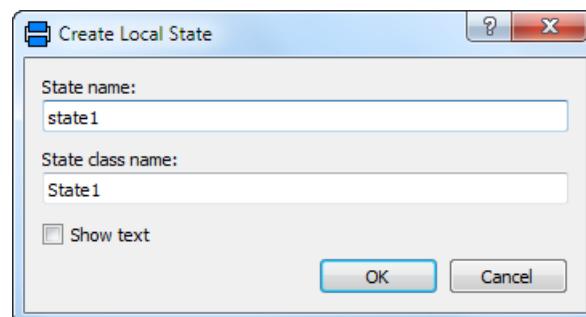
#### Creating and simulating a simple state machine

The below is a short description of the basic state machine editor features in Dymola; for more information see above.

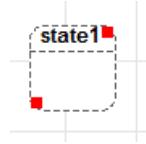
To create states and transitions the corresponding commands have been added to the **Drawing** toolbar (the rightmost ones in the figure below):



The **Create Local State** command can also be given as **Edit > 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")}))
```

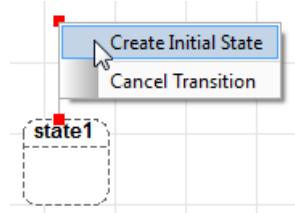
can be changed to

```
annotation(Diagram(graphics={Text(textString="%stateText")}))
```

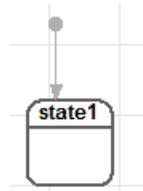


When the state is created, the **Transition Mode** is activated automatically; enabling creation of transitions by dragging, like dragging connections. You start dragging from the border of the state.

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:

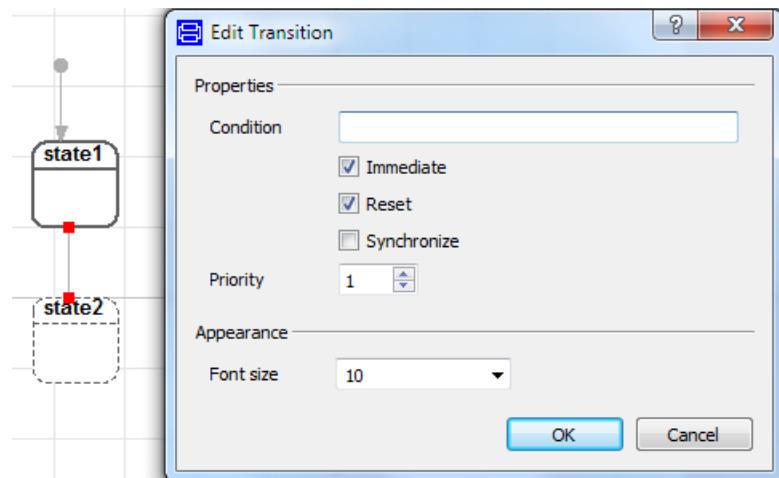


Selecting **Create Initial State** will give:



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”).

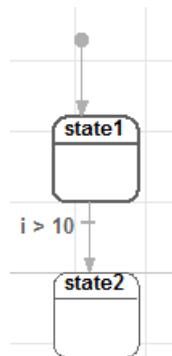
**Reset** should be selected if the target state (and states included in the target states, if any) should be reinitialized; i.e. included state machines are restarted in initial state and state variables are reset to their start values.

**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**:

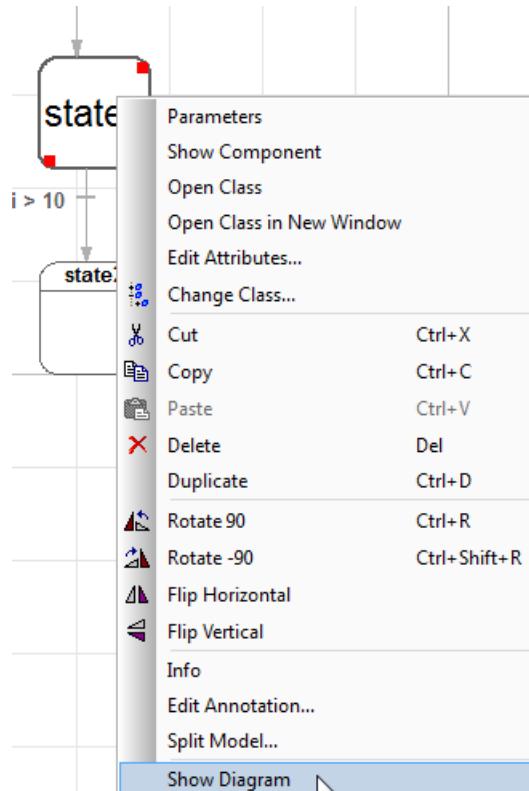


The transition is created as an arrow between the two states. It indicates the above selections

- The arrow direction is from the source state to the destination state.

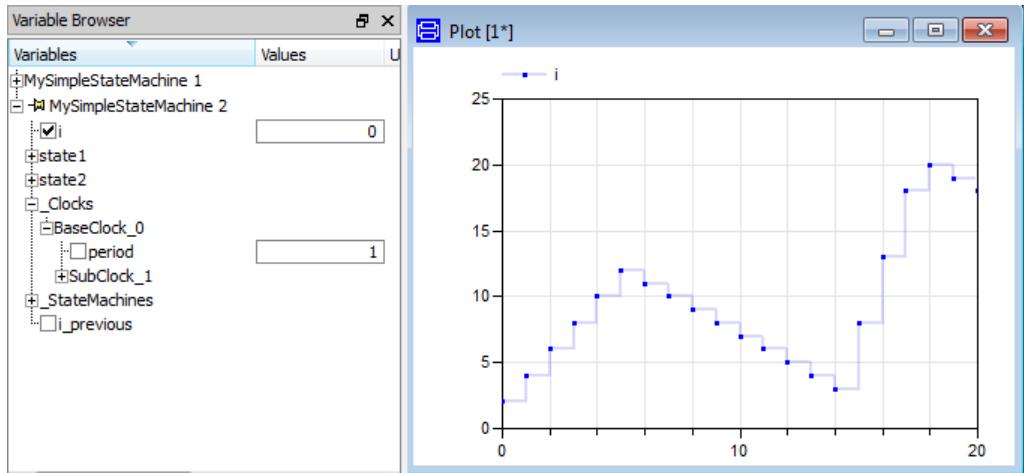
- The perpendicular bar of the transition is close to the destination state for an immediate transition, and close to the source state for a delayed transition.
- The arrow is filled for a reset transition, otherwise non-filled.
- A synchronize transition has an “inverted fork” at the source state.
- 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:

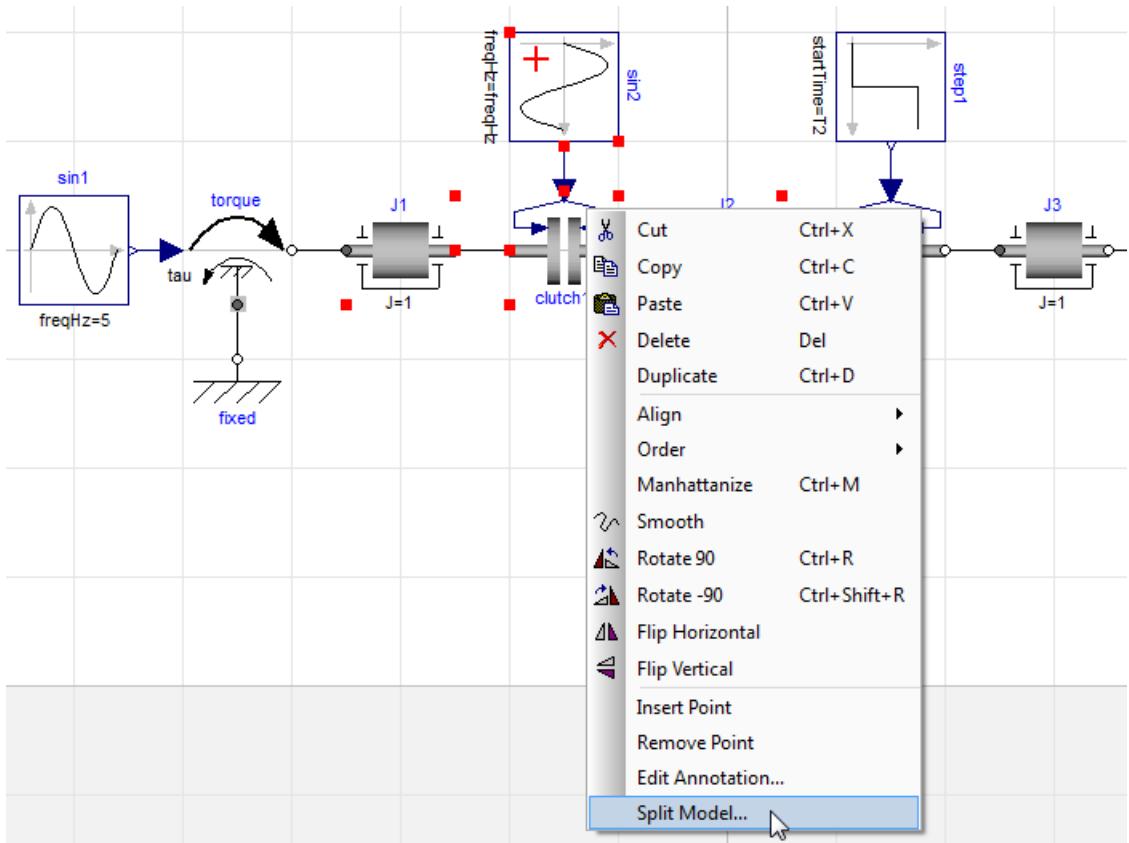


### 4.3.2 Splitting models

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, right-clicking and using the **Split Model...** command in the context menu.



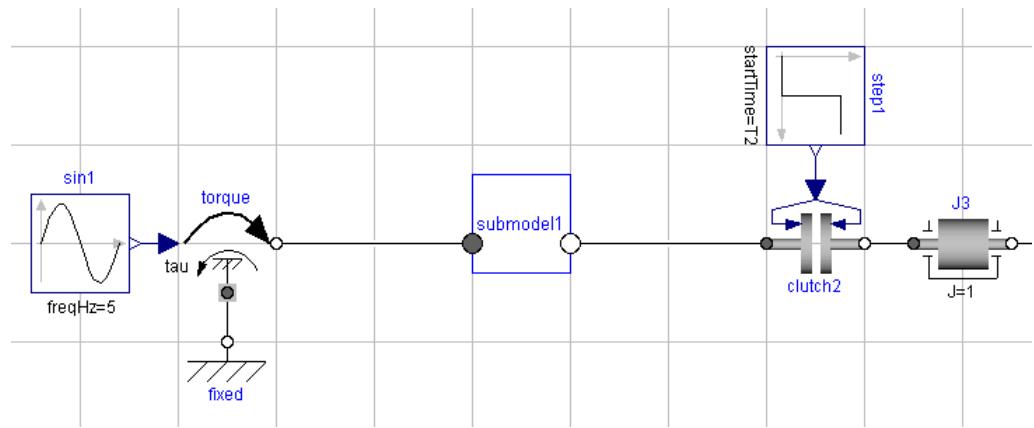
The selected components will be moved to a new class that is created automatically. That class will then be instantiated in the place of the selected components in the original model.

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.)

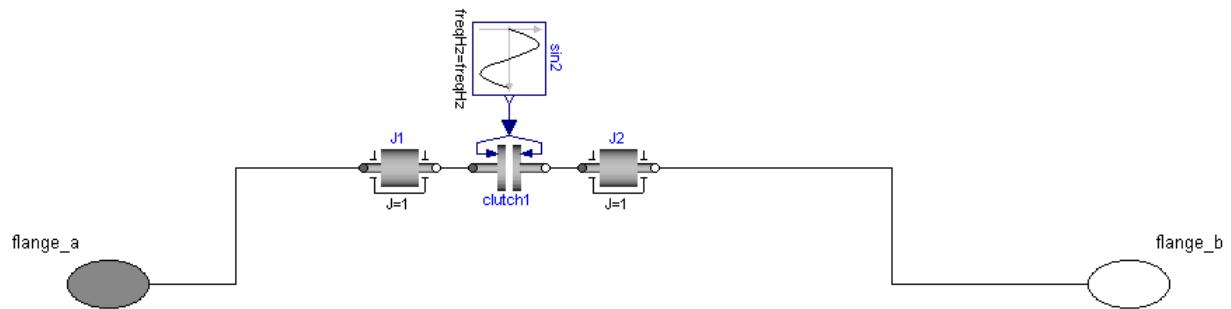
Parameters used in the components are copied to the subsystem when they (or components of them) are used ("=p" and "=p.x"), as well as when used in arrays/matrices/and simple expressions.

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:



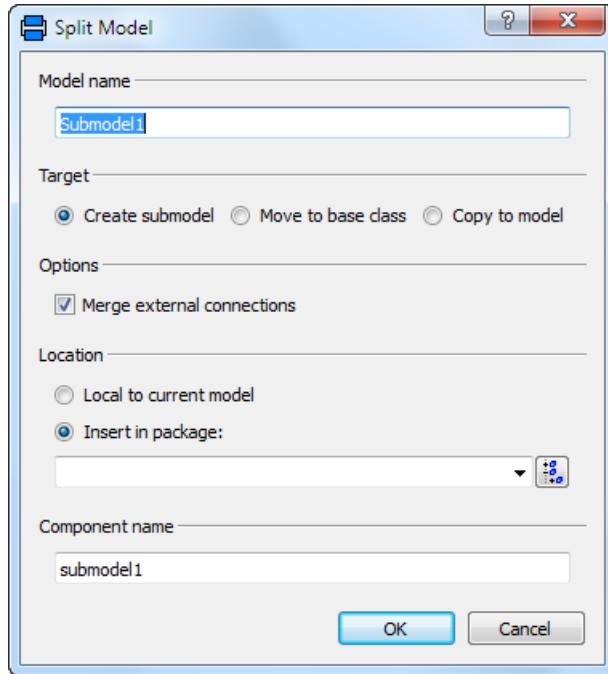
Connections between selected components are automatically included in the submodel.

### 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...** context command is:

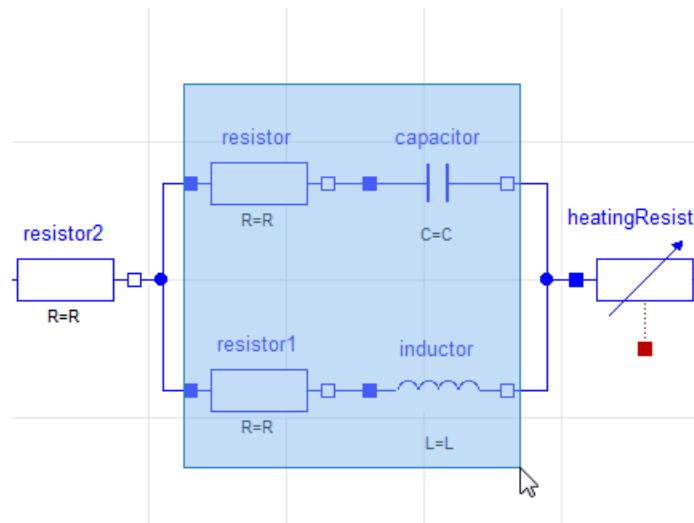


By default the submodel (class) created is public, and the user should select a package where the new submodel will be stored. Selecting **Local to current model** will store the submodel as a protected class in the package of the original model.

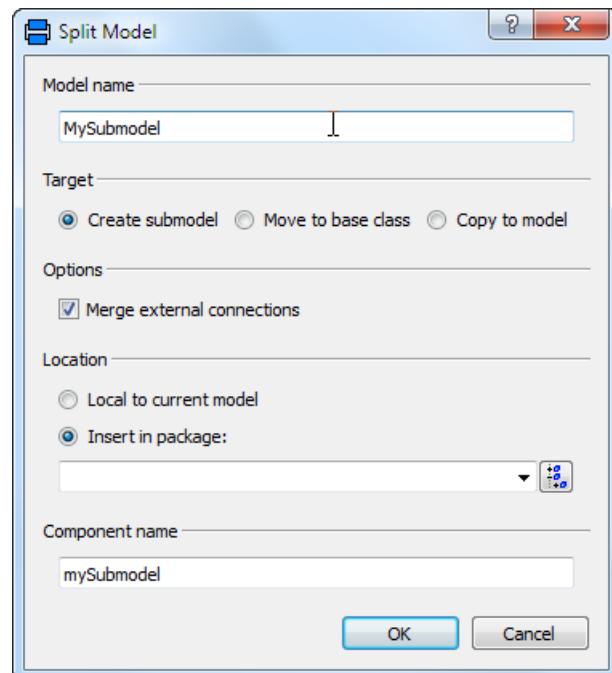
If the user changes the **Model name**, the **Component name** will automatically be changed simultaneously; the difference being that the component name will start with a lower-case letter. If the user changes the Component name, there is no automatic change of the Model name.

It is possible to merge external connections if a submodel is created. This increases the simplification of a diagram.

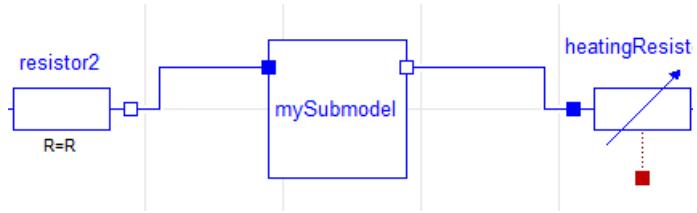
Having as a starting point selected the following:



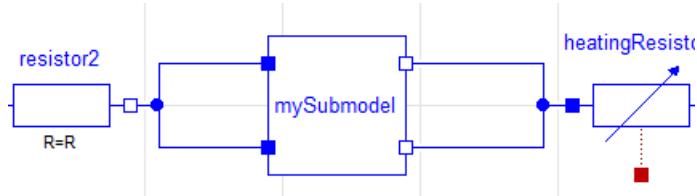
Right-clicking and selecting **Split Model...** from context menu, the following menu will appear:



Keeping the default setting of **Merge external connections** will give the result:

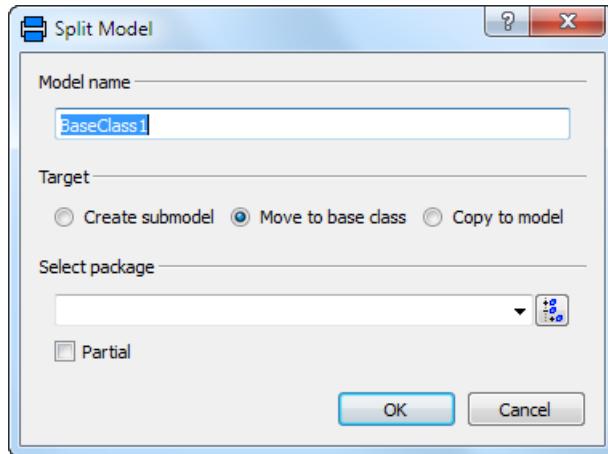


Unticking the setting will give:



### 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.)

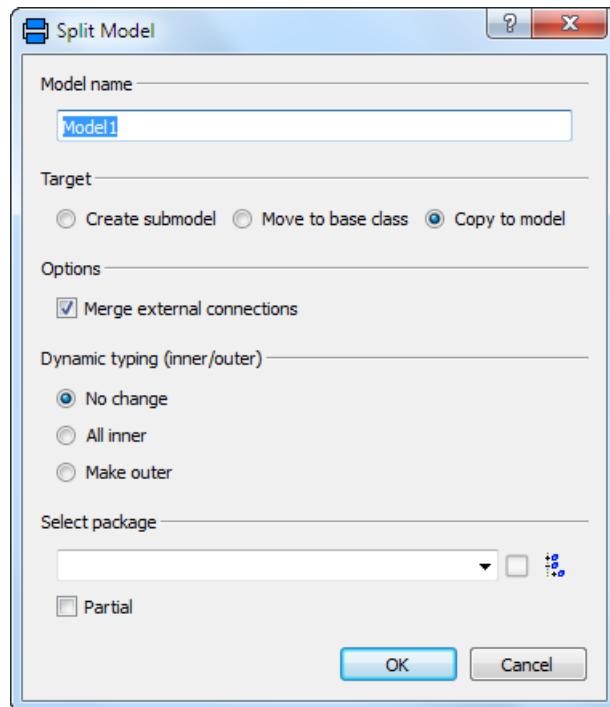
Parameters used in the components are copied to the subsystem when they (or components of them) are used ("=p" and "=p.x"), as well as when used in arrays/matrices/and simple expressions.

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 in 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.

### 4.3.3 Components and connectors

Nested connectors, overlapping (stacked) connectors, expandable connectors and array of connectors are supported. Please see next section.

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”.

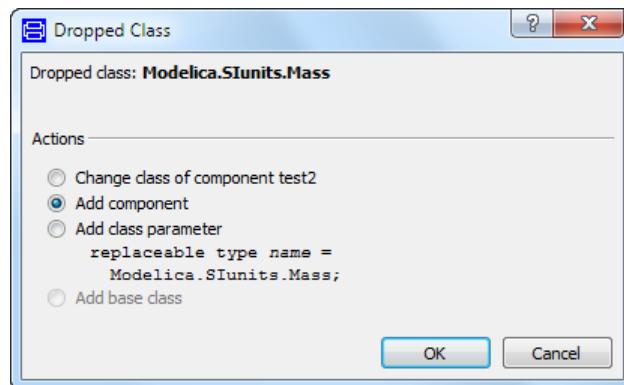
#### Inserting by dragging it to the Modelica Text layer of an edit window

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.SIunits 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.

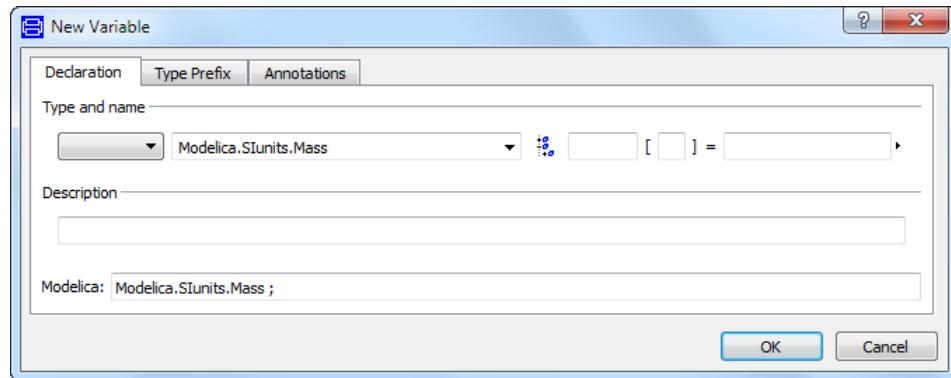
When dragging a component from the package browser on top of a component in the component browser, a **Change class of component** action (see below) is performed by default if possible. By dragging it between components, or if the default action is not possible, the following menu appears:



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.SIunits.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.)

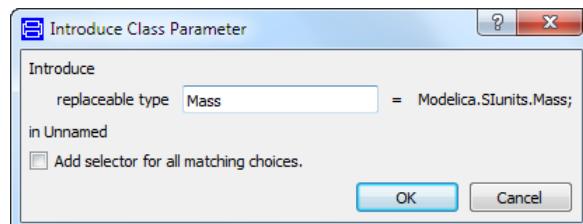
**Add component** Components of type, block, class and model can be added this way. If this alternative is selected, a new menu where variables can be declared will follow when **OK** is pressed. In the case of “mass” it will look the following:



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 261.

**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).

**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 (e.g. using the **Attributes...** in the context menu of it.)

### Re-declaring (replacing) one replaceable component

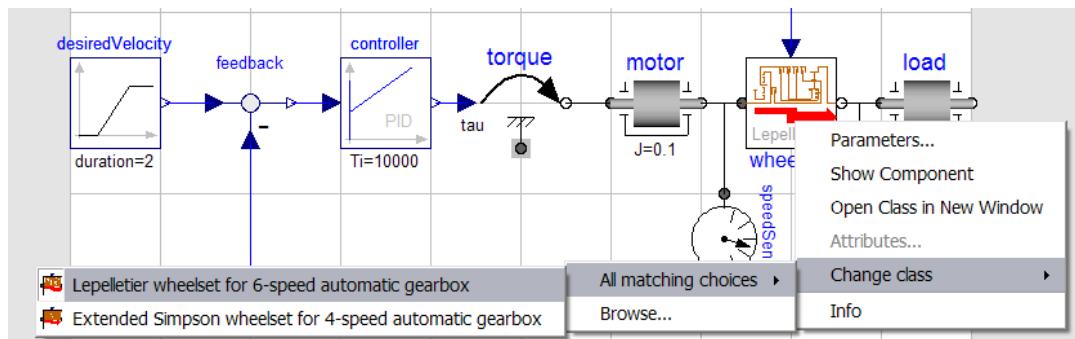
Please note that this section is closely related to the section “Building templates using replaceable components” on page 259. Please also see that section.

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 (re-declared), 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 re-declared class (below for Lepellier) instead of the original class. The Modelica Text layer and the documentation layer show the text of the re-declared class.



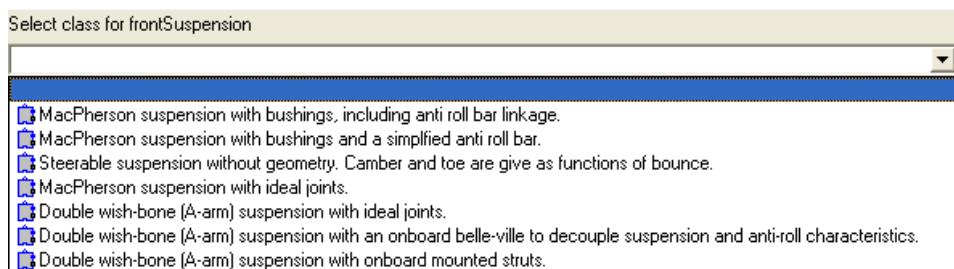
Selecting the menu choice **Change class > Browse...** instead displays a normal package browser. (This way of replacing a component can be used if the component is not replaceable as well; the **Change class...** context command will display the browser.)

#### Replacing (re-declaring) several replaceable components at the same time

It is possible to replace (re-declare) a number of components in a convenient way in the same time, without having to select them one by one. 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 re-declared components available. All matching classes are listed.

## Choices for select class.



## Replacing (re-declaring) a component already replaced (redeclared)

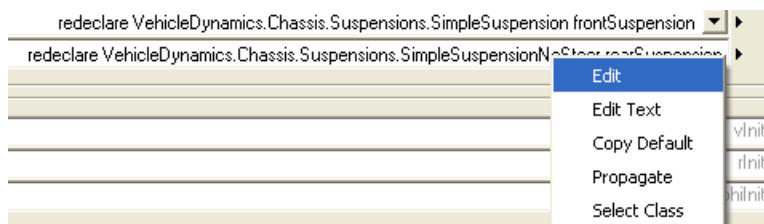
Please note (advanced users) that in the normal case a component that has been replaced (re-declared) cannot be replaced again. (That is, when the model in which the already replaced (redeclared) component is located, is being extended from or being used as a sub-component. As an example, a template 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

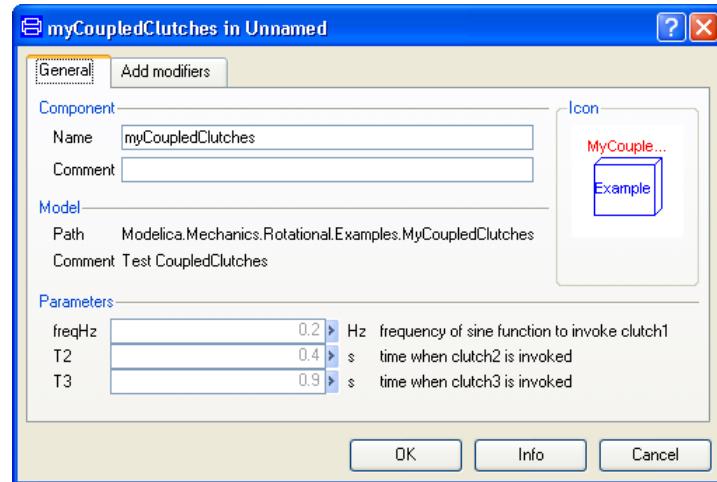
To modify and test the parameter values of a component of a read-only class the class can be extended. A component of the extended class can be manipulated using the parameter dialog.

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):

Parameters...  
Info

### 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.

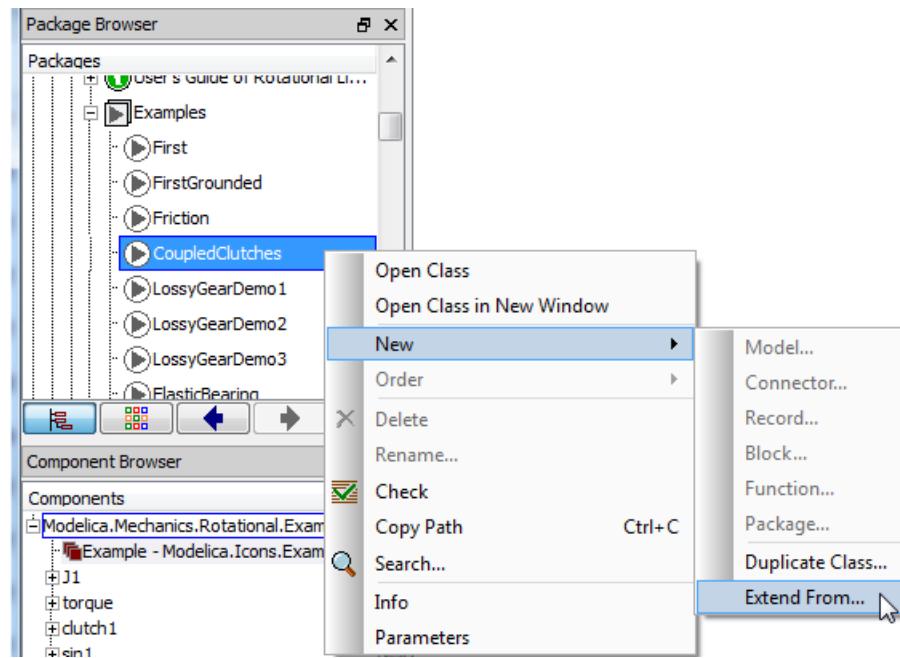
To experiment with a model such as CoupledClutches you can follow these steps. The experimental changes are then stored in a model.

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 **File > 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.



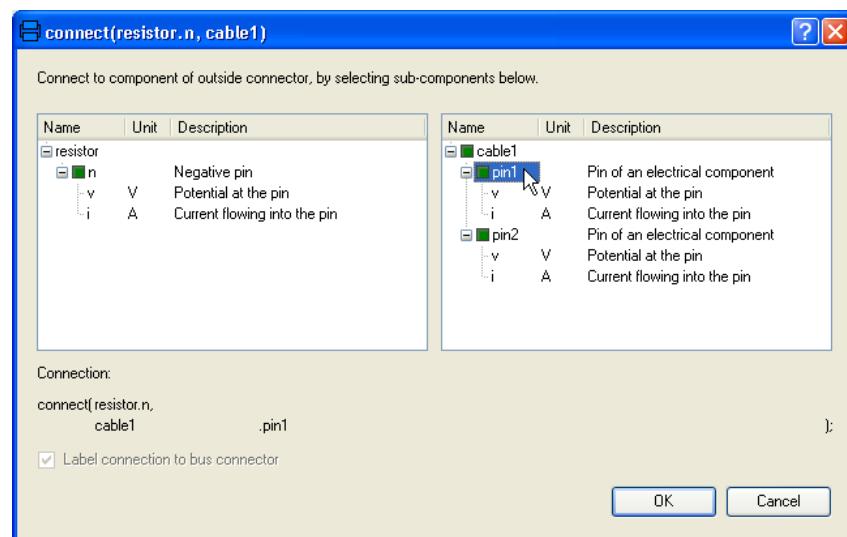
2. Instantiate the class by e.g. dragging a component from the package browser to the diagram layer of an edit window.
3. In the diagram layer select a component, several components, or no component and right-click to bring up the context menu, select **Parameters...** and modify the parameter values 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.

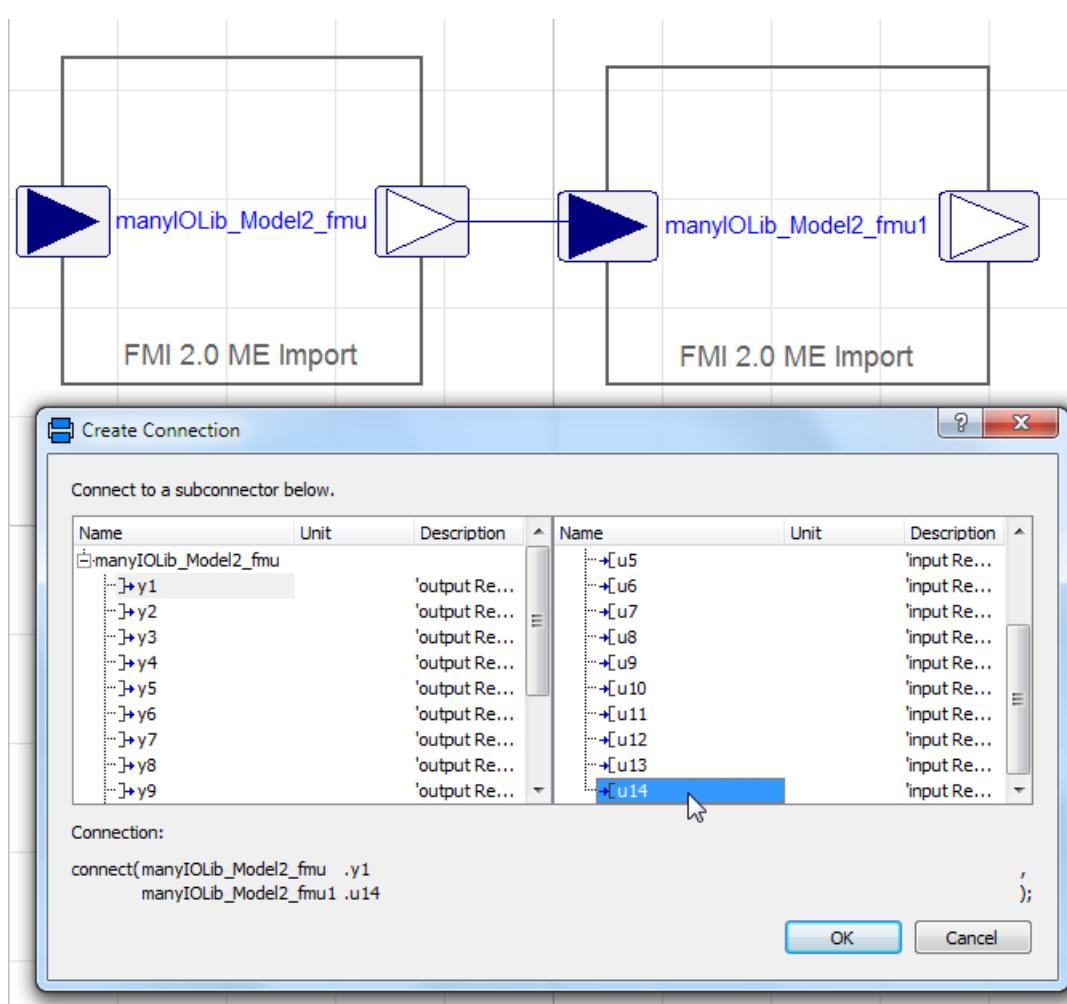


In this case a connection from the connector resistor.n to any of the two nested connectors (sub-connectors) in cable1 is possible. The one that should be used is selected by clicking on it. The resulting connection that will be the result if **OK** is pressed is shown in the lower part of the window.

## 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 y1 in the Real output group of the FMU to the left, to connector u14 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.

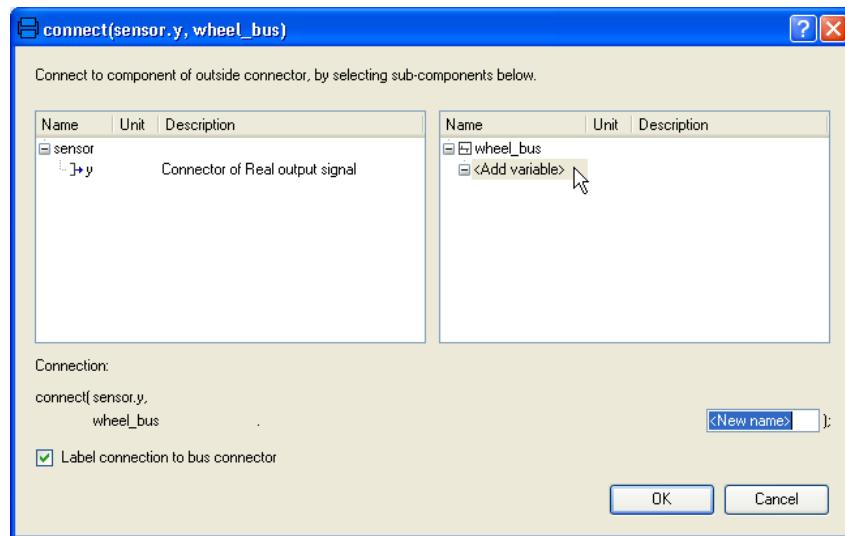
Note 3: The example is simplified, usually more groups are present.

## Expandable connectors

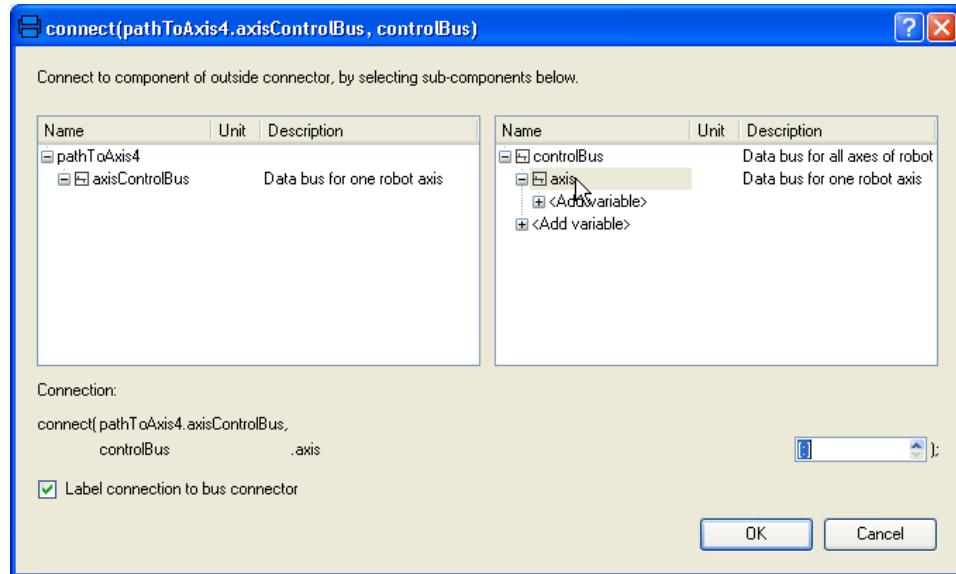
Dymola has support for expandable connectors as defined in Modelica Language. Components of expandable connectors are automatically treated as connectors.

In the example below we have connected from the output *y* of the component sensor to the expandable connector *wheel\_bus*. We are then given a choice of what variable in the *wheel\_bus* to connect to and can select **<Add variable>** to browse for existing connections to *wheel\_bus* in other models, or construct a new variable.

**Tree browser for connecting to bus.**



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 Message window. See next chapter, section “Message window”.

### 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, use **Edit >Flip Vertical** or **Edit > Flip Horizontal** on the connector.
- 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.)

## Presentation of components and connectors in the diagram layer

### Conditional components/connectors

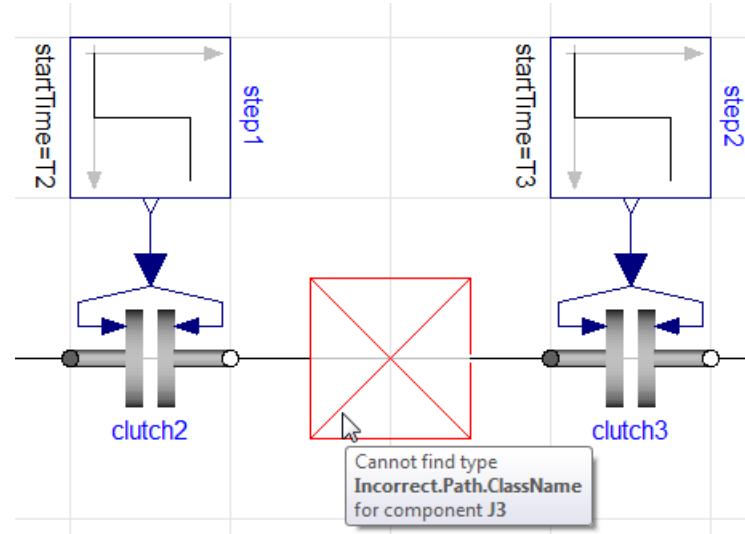
In a top-level view, conditional connectors and components (see previous section) are always shown regardless if they are enabled or disabled, since all cases must be taken into account by the model creator. When using the context command **Show Component**, disabled conditional components and connectors 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.

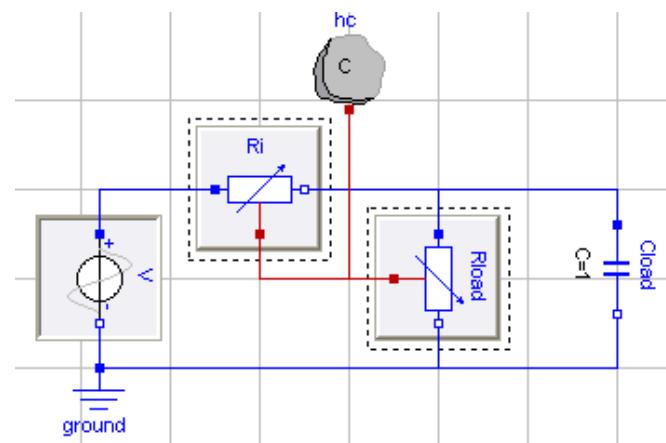
### Components of a non-existing class

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, re-declared and inherited components

The highlighting indication makes it easier to visualize model structure in the diagram layer of the edit window.



Replaceable and re-declared components:

- Highlights components and connectors which can be replaced. Replaceable components with the default value are shown sunken, as a socket. Re-declared components are shown raised, as mounted in the socket. Components of a replaceable/re-declared *class* are outlined with a dashed line.

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.

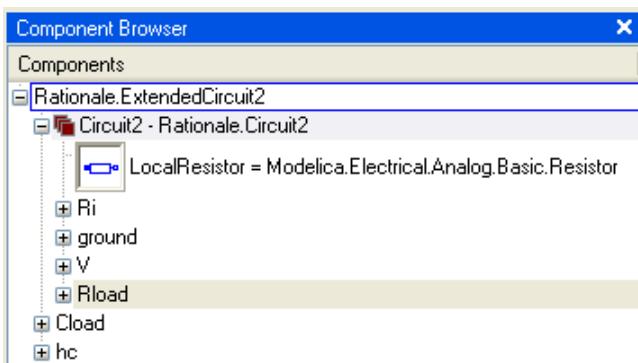
Highlighting can be turned on or off in **Edit > Options...**, for example when copying to clipboard or exporting images. It is always off during **File > Export... > HTML....** Re-declared components are always shown.

### **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. Re-declared 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.

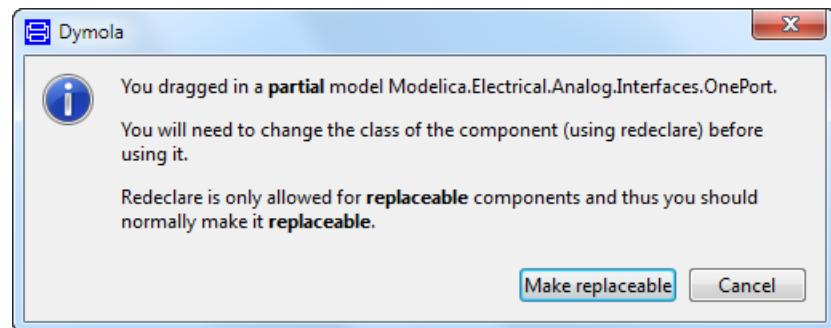
### 4.3.4 Building templates using replaceable components

#### Building templates

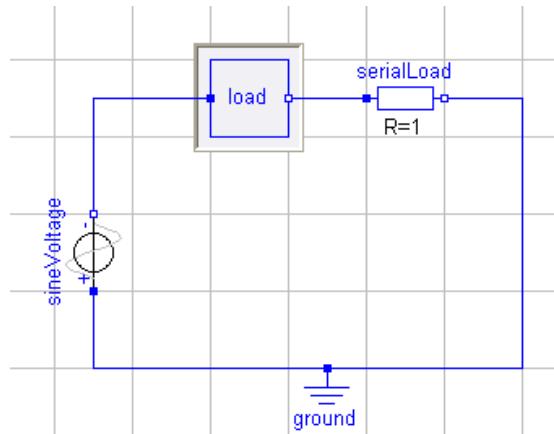
It is easy to build model templates using replaceable components:

- 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 context menu entry **Edit Attributes...** for component makes it possible to set and modify the constraining class (and replaceable attribute), both for the original declaration and a re-declare.
- 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 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

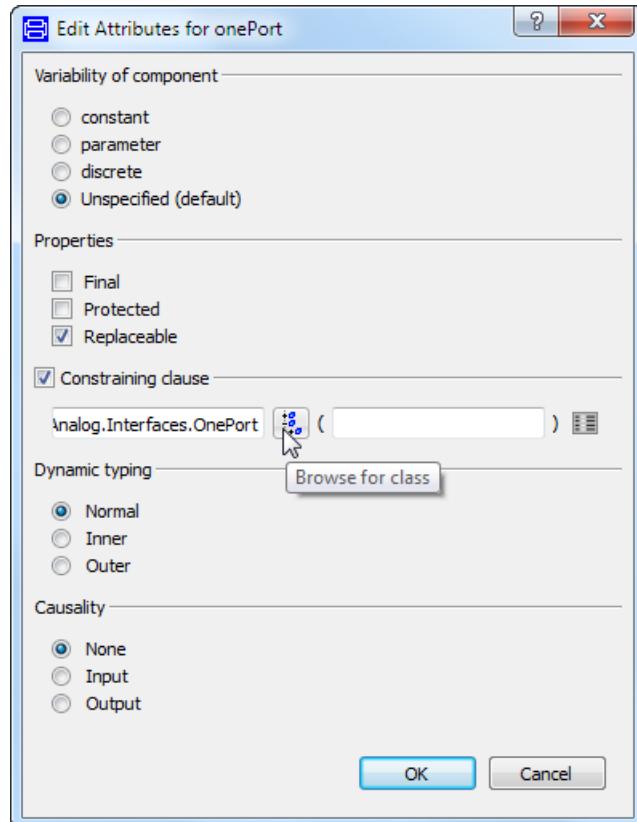


By building the circuit in the usual way one then gets

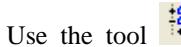


and can then change the default to Resistor using **Change class > All matching choices** in the context-menu of load.

Alternatively one can modify replaceable and constraining class using **Attributes...** in the context menu.



For more information about the context menu, please see section “Context menu: Components” on page 343.



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 **What's this?** text available 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.

## 4.3.5 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 chapter “Introduction to Modelica”, section “Initialization of models”, sub-section “Parameter values”.

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, using menus.
- 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.

### 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.

#### Double-clicking on a component

Double-clicking on a component or connector in the graphical editor (diagram layer of an edit window) displays a parameter dialog.

#### Using a context menu

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** in the context menu of the parameter input field in the General tab in the Parameter dialog will display a variable declaration dialog.

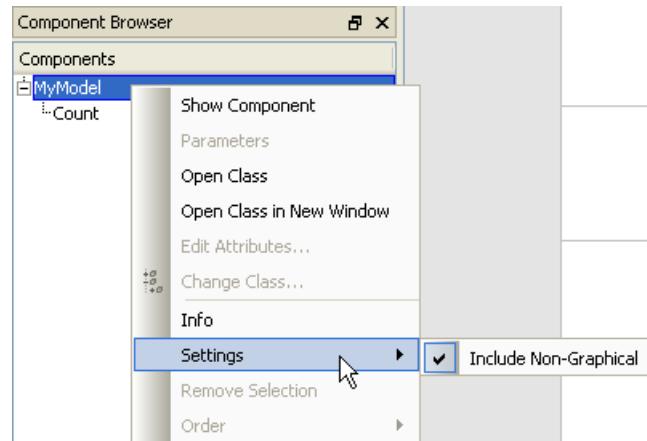
#### Using a menu

It is easy access to all variable declarations in a model using **Edit > Variables**. 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 “Edit > Variables” on page 318 for more information.



You can also use **Settings > Include non-graphical** in the component browser and then select **Parameters...** for a variable; this will display the variable declaration dialog.

**Component browser settings.**

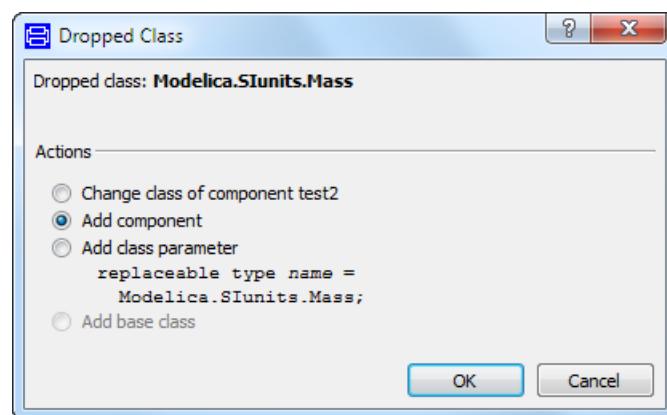


### Drag-and-drop operation

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

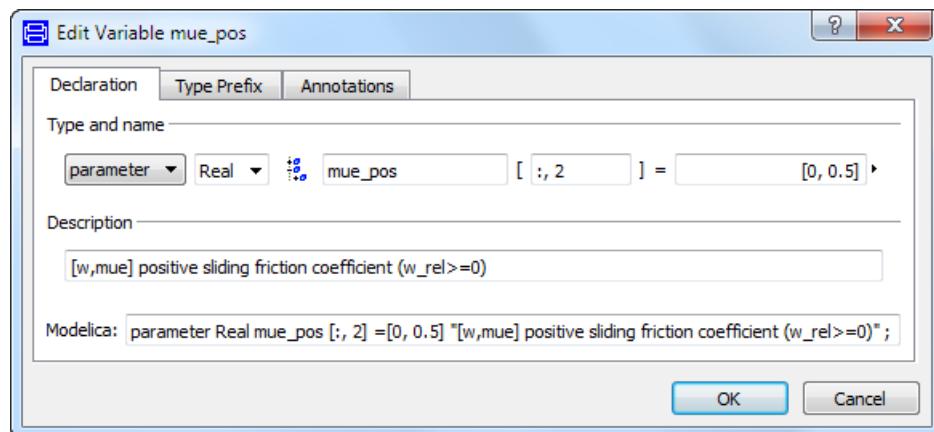
Parameters and variables can of course also be declared in the Modelica Text layer. Please see section “Programming in Modelica” on page 193.

## 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.



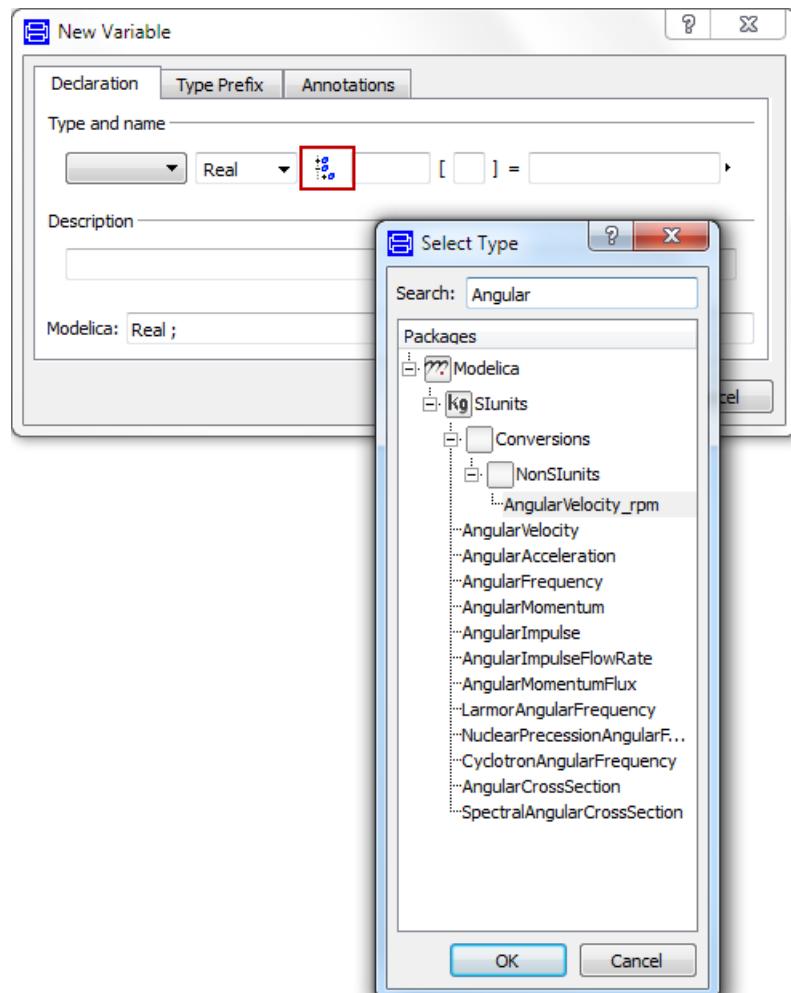
Depending on in which situation the variable declaration dialog is used, the available alternatives to select might differ slightly. The example above is from the parameter `mue_pos` in a clutch.

#### 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:



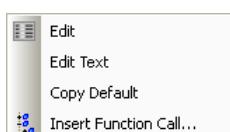
What is entered in the **Search** field will dynamically select what is shown in the Packages pane. The choice selected by default is marked with a light grey background. If the default selection is the wanted one, it is sufficient to click **OK** to get that type. Otherwise the wanted type can be selected 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



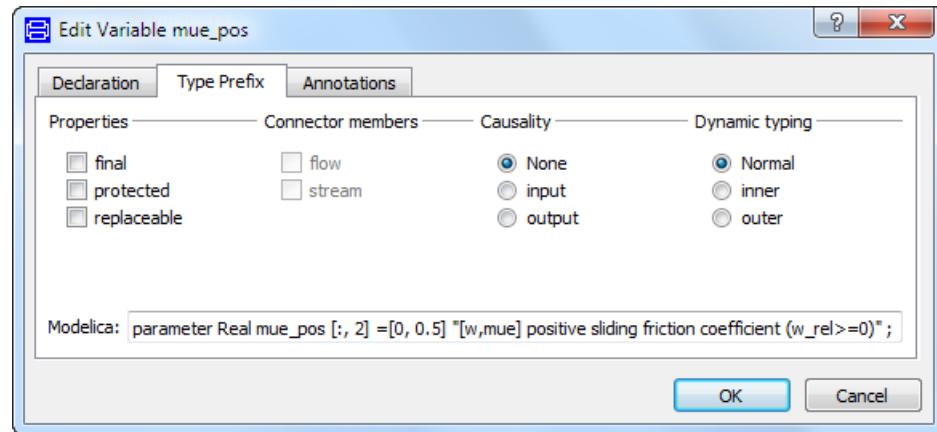
information about this menu, please see the section “Context menu: Variable declaration dialog; Value input field” on page 355.

### 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 re-declared 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.

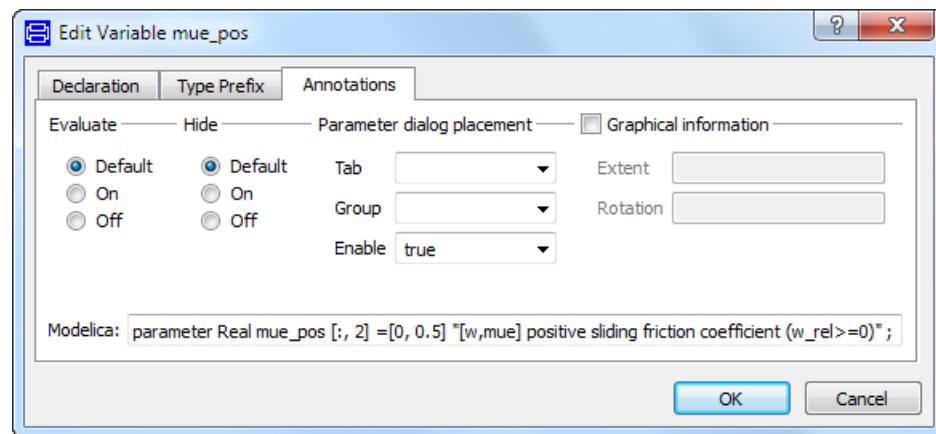
**Normal** – the variable has no dynamic type.

**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. 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.

**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.

### 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 user can edit the look of the dialog in other ways as well – see following sections. See also the chapter “User-defined GUI” in the manual “Dymola User Manual Volume 2”.

### Graphical information group

Position, size (extent) and rotation of graphical component.

### Parameter dialog – advanced

The basic handling of the parameter dialog is described in section “Editing parameters and variables” on page 170.

### 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 170.
- Replaceable classes in the model, which can be re-declared 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 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.

### 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)).
```

### 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.

### Parameter dialog – editing the dialog

The user can do some editing in the parameter dialog.

- Adding tabs.
- Adding groups.
- Hiding variables.
- 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 267. For more information, and a description of the other features, please see the chapter “User-

defined GUI” in the manual “Dymola User Manual Volume 2”. (Adding tabs and groups in tabs are also described.)

## 4.3.6 Matching and variable selections

### General

To simplify postprocessing by reducing the amount of variables saved and displayed in the variable browser in Simulate mode, variable selections can be used.

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.

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** reached by the command **Simulate > Setup...** in Simulate mode, the **Output** tab. Please see the corresponding section in next chapter for more information.

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” on page 276 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%`, and `%i%`.

The `%name%` corresponds to the part matching the regular expression and `%componentPath%<%name%>` is the full path of the variable. 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.

## Examples

The following examples illustrate some features.

In order to easily change the selections, they are put in separate models, e.g. `Selection1`.

They are applied, in these examples, on the model example `FullRobot`, by extending to a new model, `MyFullRobot`, looking 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

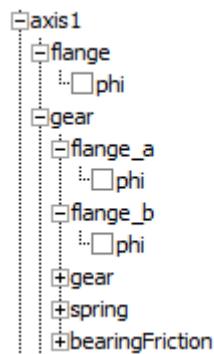
(The following models are to be extended like above.)

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="MySelection",
  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.

### Selections – introducing new names

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%"`.

```

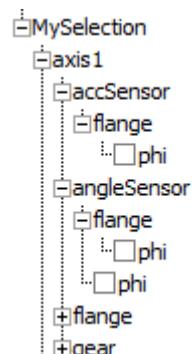
model Selection2

annotation (_Dymola_selections={
  Selection(name="MySelection",
    match={MatchVariable(name=".phi", newName="%path%")})});

end Selection2;

```

It will give the same result above, but *also* the following *additional* content of the variable browser:



Including several variables in the selection can be made by using “|” in the regular expression for name:

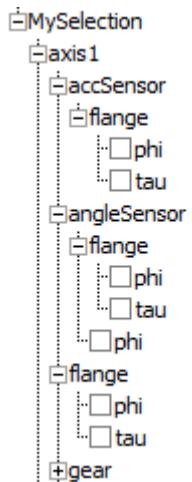
```

model Selection3

annotation (__Dymola_selections={
    Selection(name="MySelection",
        match={MatchVariable(name=".phi|tau", newName="%path%")})});
end Selection3;

```

resulting in:



Alternatively of using “|”, several matches can be used. They are or ‘ed.

```

model Selection4

annotation (__Dymola_selections={
    Selection(name="MySelection",
        match={MatchVariable(name=".phi", newName="%path%"),
            MatchVariable(name=".tau", newName="%path%")})});
end Selection4;

```

It is possible to build subtrees, by including a subtree name in the newName path:

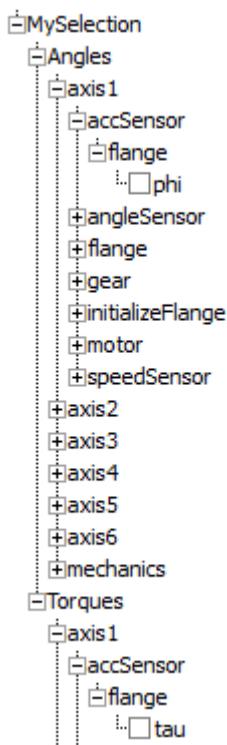
```

model Selection5

annotation (__Dymola_selections={
    Selection(name="MySelection",
        match={MatchVariable(name=".phi", newName="Angles.%path%"),
            MatchVariable(name=".tau", newName="Torques.%path%")})});
end Selection5;

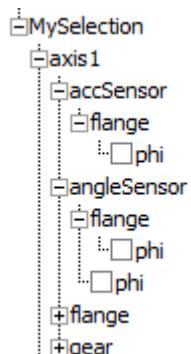
```

resulting in:



### Selections -matching within a scope

It can be noted that the annotation in the model Selection2 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`".

```

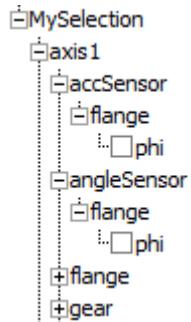
model Selection6

annotation (__Dymola_selections={
    Selection(name="MySelection",
        match={MatchVariable(className="Modelica.Mechanics.Rotational.Interfaces.Flange*",
            name="phi", newName="%path%") } ) });

end Selection6;

```

The result is then:



## 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

The below is an example of user interface in Simulate mode 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”.

In Simulate mode, 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:

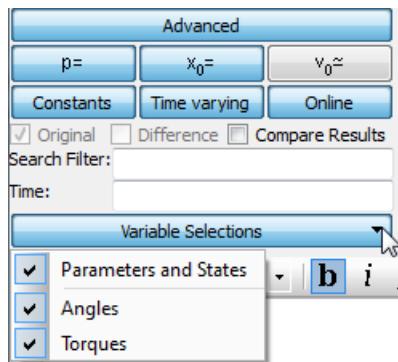
```

model Selection7

annotation (__Dymola_selections={
    Selection(name="Angles",
        match={MatchVariable(name="*phi", newName="%path%") }) ,
    Selection(name="Torques",
        match={MatchVariable(name="*tau", newName="%path%") }) });
end Selection7;

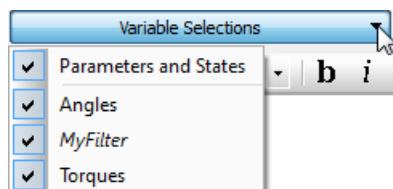
```

This corresponds to the following items in the advanced filter menu in the variable browser:



The selection **Parameters and States** is predefined, and corresponds to the interactive fields for setting parameters and initial conditions of states.

In the above example, the 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:



#### 4.3.7 Using data from files

Data files (tables etc.) can be read and written from Dymola in several ways. Common file formats are:

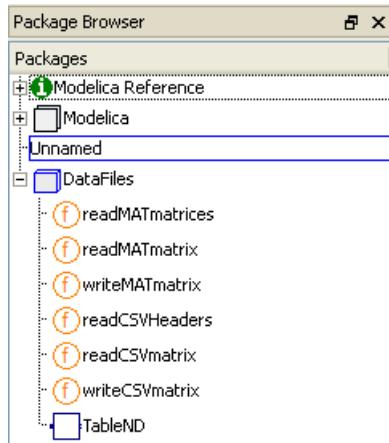
- Matlab 4 files (.mat).
- Comma separated values files. Comma separated values files (used by e.g. Microsoft Excel). We recommend the file extension .csv but .txt format is also supported. In the latter case there might also be a choice of separator characters.

#### 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 357. Matlab 4, CSV and text files can be handled.

## DataFiles package

The content of DataFiles package.



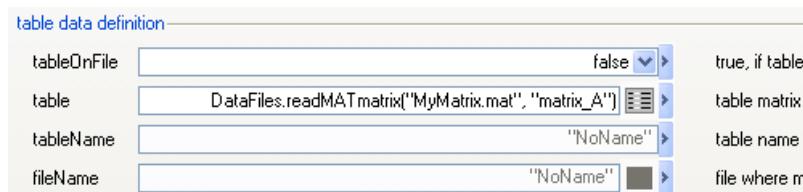
The package `DataFiles.mo`, located in `Program Files (x86)\Dymola 2015 FD01\Modelica\Library` contains functions for reading and writing data, and a block `TableND`.

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).

## 4.3.8 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 283). 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 172) 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 (x86)\Dymola 2015 FD01\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.)

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);

.....

// Kinematic Viscosity
defineUnitConversion("m2/s", "mm2/s", 1e6);

// -----
// 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);
```

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");
```

Some outcommented examples of default display units are present. If the user wants to present e.g. **bar** instead of **Pa**, the only thing that has to be done is to remove the comment signs **//**.

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)
```

## 4.3.9      Changing graphical attributes

### Editing graphics using Edit Annotation

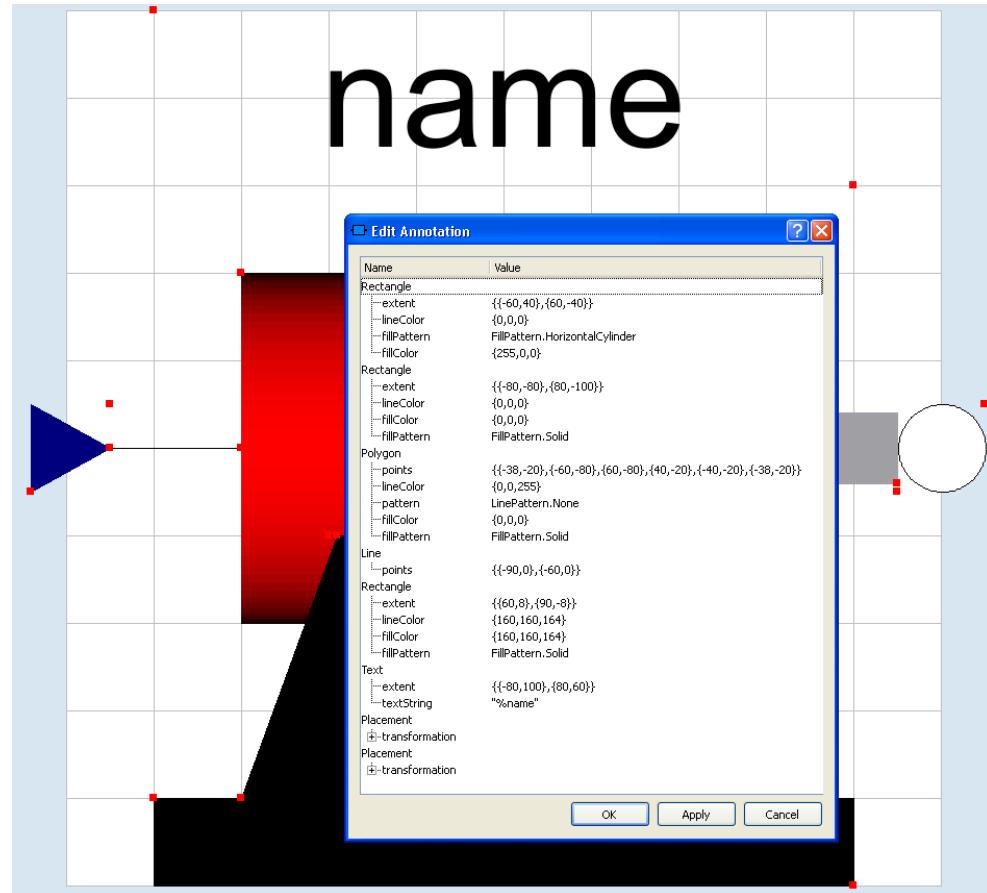
Selecting graphical object(s), right-clicking to get the context menu and then select **Edit Annotation...** give editing possibilities where attributes can be changed by text editing in the displayed window. The window can be said to display a form where editing can be done.

An important condition is that the feature that should be edited is already present; that is, the feature must be presented in the form – it is not possible to insert rows with new features etc.

This means that **Edit Annotation...** is of most value when it comes to editing attributes that are always present e.g. the coordinates of corners in a curve.

Selecting the objects present in the icon for the motor that was built in the chapter “Getting started with Dymola” and selecting **Edit Annotation...** will give the following:

**Example of Edit Annotation.**



Each object is represented in the form. Editing is done by clicking on the relevant line and then edit that line. Changes are implemented using **OK** or **Apply** (**OK** will also close the editor); **Cancel** is used to regret any change.

Please note that also the location of components is available as annotations (Placement – transformation).

#### 4.3.10 Viewing graphical attributes

Having no object selected and taking up the context menu, the entry **View Annotation** will display the annotations for the objects located in this view. Please compare with the previous section.

---

## 4.4 Checking the model

### 4.4.1 Commands

Please see section “Edit > Check” on page 314; 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.SIunits provides a large set of predefined quantities and it is recommended to use them whenever possible. See the Users Guide of package Modelica.SIunits.

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 fulfills 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.SIunits 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	$\mu$
$10^9$	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 <sup>2</sup>
pascal	Pa	N/m <sup>2</sup>
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 <sup>2</sup>
henry	H	Wb/A
degree Celcius	degC	K
lumen	lm	cd.sr
lux	lx	lm/m <sup>2</sup>
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 <sup>3</sup>
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

$$\omega = 2\pi \text{ rad}$$

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.SIunits.Mass m = 10 "Mass";
  Modelica.SIunits.Force f = 10 "Force";
  Modelica.SIunits.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;

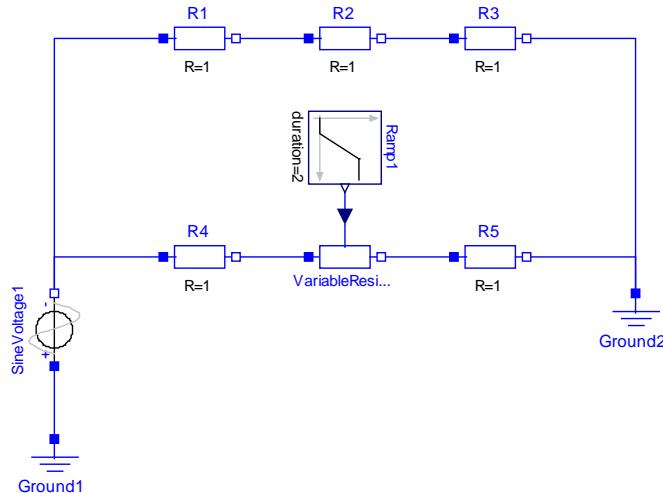
```

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

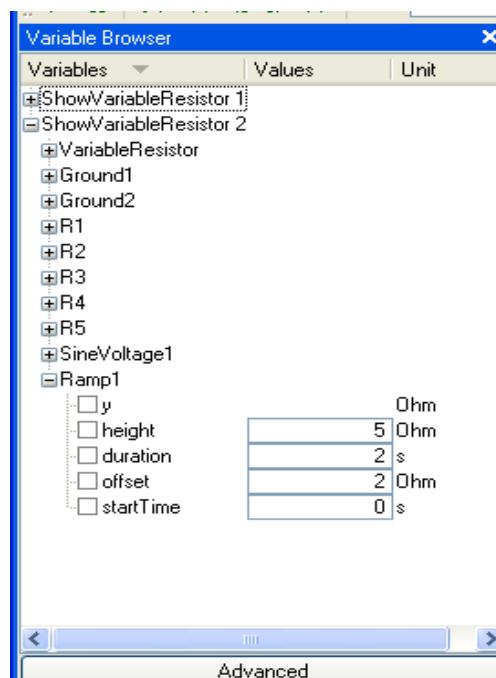
The Modelica.Blocks library includes general blocks to define sources and mathematical operations. Their inputs and output have of course no units specified. For user convenience, Dymola has introduced automatic deduction of units. Consider the example Modelica.Electrical.Analog.Examples.ShowVariableResistor.

#### 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**. This option is available using the command **Simulation > Setup...**, the **Translation** tab. 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.SIunits.

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](http://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](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

###### Dymola Main Window

Using the command **Window > Tools** the content of Dymola Main window can be changed, both when it comes to what sub-windows that should be shown and what toolbars should be shown. More information about this is given in section “Window > Tools” on page 327.

Using the command **Edit > Options... > Appearance** also influences what is shown in Dymola Main window. Please see section “Appearance tab” on page 319.

##### 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 **Edit > Options... > Appearance**. Please see section “Appearance tab” on page 319.

### **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 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

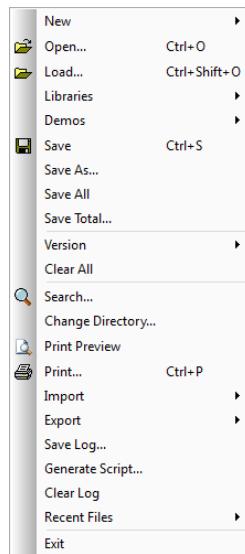
## Editor command reference – Modeling mode

The first sub-sections in this section cover the menu entries in Dymola Main window (File, Edit, Commands, Window and Help menus). A number of commands are either only available in Simulation mode, or generally more used in that mode. A corresponding section is to be found in that chapter.

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



#### Toolbar



The toolbar contains buttons open and save models, and to print. The final button enabled interactive help, see “Help > What’s This?” on page 329.

The toolbar can be displayed/hidden using **Window > Tools** and selecting/deselecting **File**. This toolbar is also shown in Simulation mode, but contains fewer possibilities in that mode.

#### File > New... > Model etc.

Creates a new model, connector, record, block or function. Dymola presents a dialog window where the user must 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.

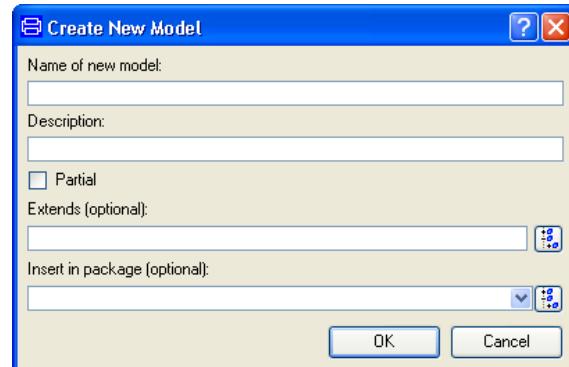
For creation and handling of functions, please see chapter “Simulating a model”, section “Scripting”.

The fourth 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.

**Drag from the package browser to extend or insert in a package.**

The fifth 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 are available in a drop-down list.

### **Creating a new model.**



### **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:

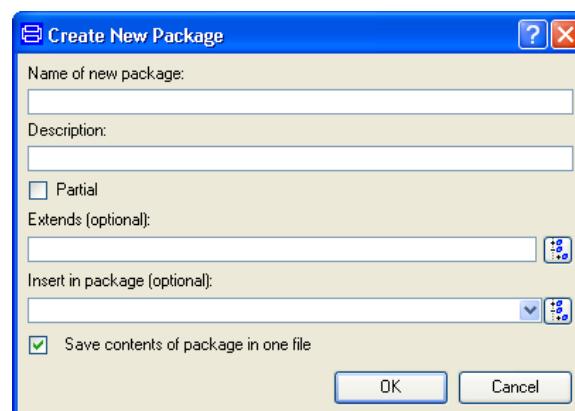
- Saving the package as a single file. All models etc. of the package are stored in this file. This alternative is recommended for small packages.
- Saving all models etc. of a package in separate files. The package itself is stored as a directory in the file system, plus a file called package.mo which 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 by **Edit > Attributes...** after the package has been created see “Edit > Edit Attributes...” on page 316 or, for a top-level package using **File > Save As...** and not changing the name, see “File > Save As...” on page 294.

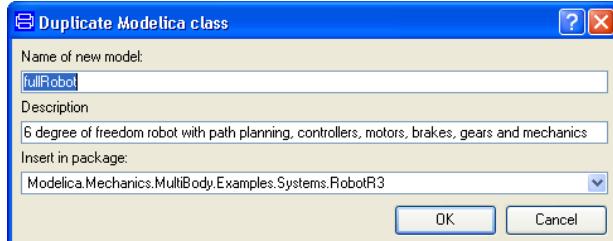
### **Creating a new package.**



## **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.

### 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.

## **File > Open...**

Reads the contents of a Modelica file. The name of all read classes is shown in the status bar; errors found when parsing the Modelica model are shown in the message window. The number of read classes and the number of errors and warnings are shown after the file has been opened.

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.

Dymola will automatically change the current working directory to the directory of the opened file. Compare the below command.

## **File > Load...**

This command works as the above command, the only difference is that the current working directory is *not* changed. Note that it is possible to set the wanted working directory when opening Dymola using a shortcut. See chapter 6 “Appendix – Installation” for more information.

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.

**The libraries menu is extended when optional libraries are installed.**

## **File > Libraries**

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 known libraries. Please see chapter “Appendix – Installation”, section “Installation on Windows”, sub-section “Additional setup”.

## **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 chapter “Appendix – Installation”, section “Installation on Windows”, sub-section “Additional setup”.

## **File > Save**

Saves the class in the current window.

- If the original model definition was read from a file, that file is updated.
- Otherwise, the user is prompted for a filename. The name of the class with file extension .mo is default.

If the name of the class is “Unnamed” (for a new class), Dymola will ask for another class name.

Dymola will ask the user to save a modified class before terminating.

## **File > Save As...**

Duplicates the current model and saves it to a new file. See also “File > New... > Duplicate Class” on page 293. 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).

## **File > Save All**

Saves all modified model classes with a File > Save operation. This operation can also be made in Simulation mode.

## **File > Save Total...**

Saves the class in the current window and all classes used by it. This creates a single file with the complete definition of a class, which is independent of other files and libraries.

## **File > Version**

This command is only available if the option Model Management is available. Please see the manual “Dymola User Manual Volume 2” chapter “Model Management”, section “Version Management”.

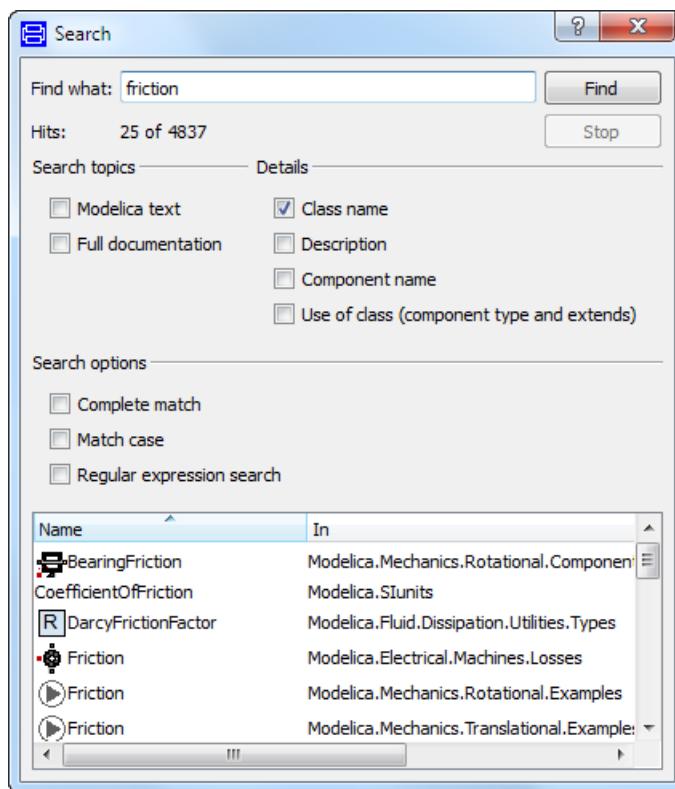
## File > Clear All

Performs a Save All operation and then removes all model definitions in Dymola, except the Modelica standard libraries.

## File > Search...



The search dialog.



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;
```

The search options control how the matching is done for each searched item:

**Complete match** Match the complete contents of topics. If not checked, the search pattern will match parts of the topics.

**Match case** Match upper and lower case exactly. If checked, the search pattern a will not match A.

**Regular expression search** 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.

For further search inside the class a “find” command (e.g. the command **Edit > Find**) can be used. Please see section “Edit > Find” on page 310 for more information about this command.

### **File > Change Directory...**

Display a dialog which allows the user to change the current directory. The current directory is used as the default location for opening files, and for saving simulation results.

Change Directory may also be given as a textual command. Entering `cd` (without directory specification) in the Command input line of a Command window prints the current working directory in the Command window.

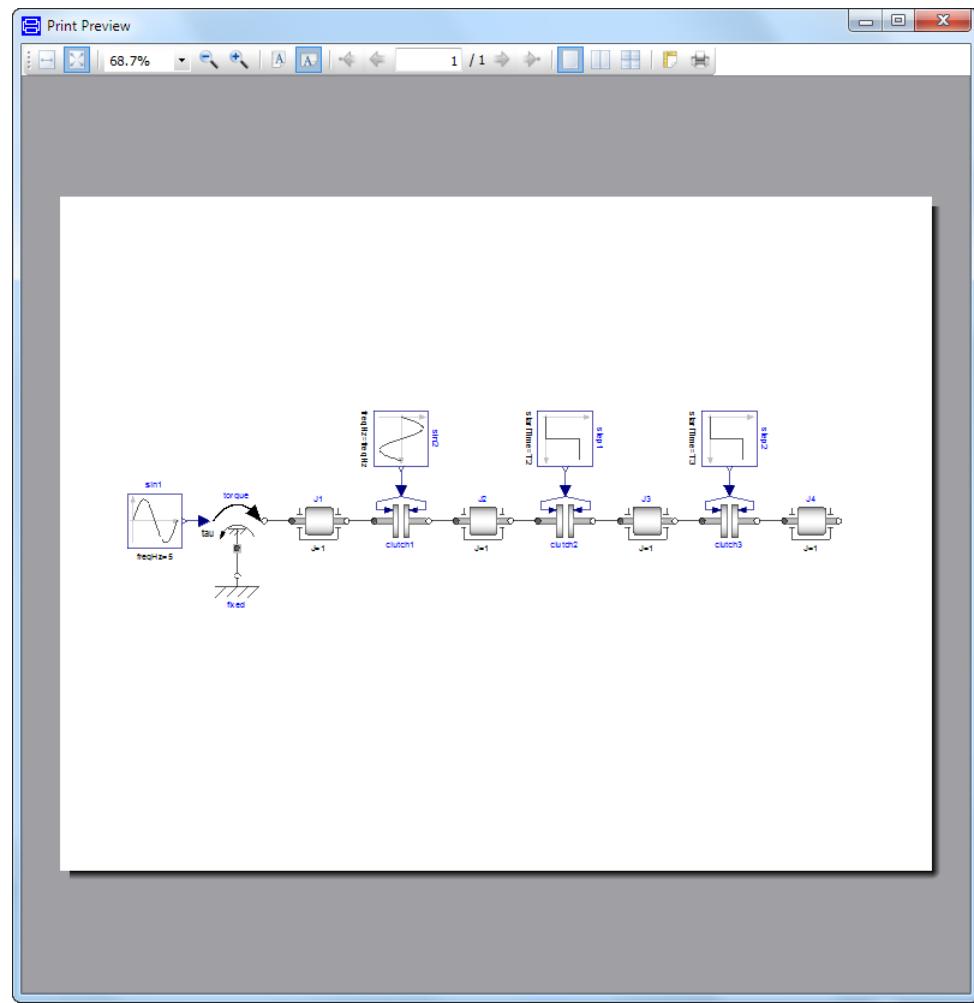
#### **Changing directory.**



On Windows, the current directory can be an UNC path. Note: Compilation is handled by copying to a temporary directory, which adds some time to the compilation.

### **File > Print Preview...**

Displays a print preview in a separate window.



A number of commands are available in the top bar, some particularly useful features are:

- Selection of portrait/landscape
- Print Setup
- Print command

### **File > Print...**

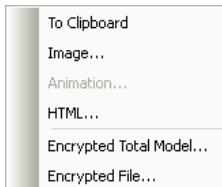
Prints the contents of the window on the current printer.

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 > Import > FMU**

This command corresponds to the built-in function `importFMU` of the selected FMU. Since the settings used in this command are described in the next chapter, please see this command in next chapter. Note that dragging an .fmu file into the Dymola main window will perform this command on it automatically.

## **File > 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.

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.

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.

This command is only available in the Windows version of Dymola.

## **File > 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.

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.

## **File > Export... > Animation...**

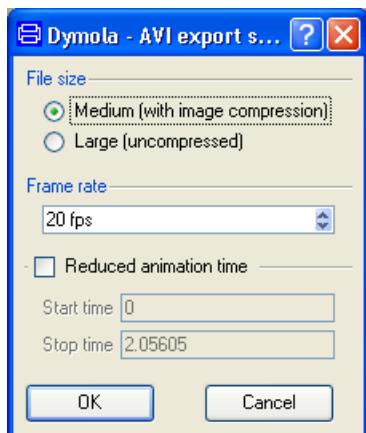
Saves the current animation window in any of the formats:

- 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.**



#### **File > Export... > HTML...**

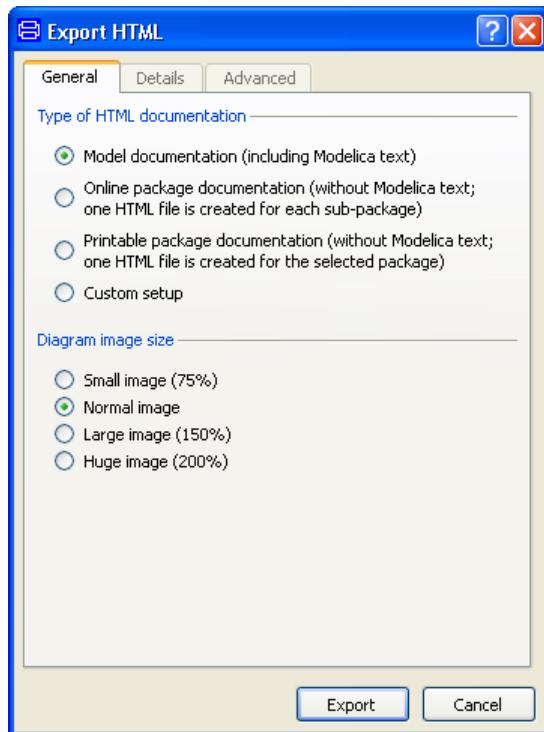
Exports comprehensive model and library documentation in standardized HTML format according to selected options and a CSS (Cascading Style Sheet) file. The generated HTML code is controlled with regards to documentation contents, references to other files and the size of graphics.

The CSS file is located in `Program Files (x86)\Dymola 2015 FD01\Insert` and has the name `style1.css`. Presently the file contains specifications of fonts, headings, colors and tables. The file can of course be adapted by the user, but do not change the name of it (and it might be wise to keep a copy of the unchanged file if you want to go back to the default setting). The file is referred to from inside Dymola.

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.

#### 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.

#### 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 corresponding to the package documented.

A Microsoft Word 2003 template for generation of printed library documentation is available in Program Files (x86)\Dymola 2015 FD01\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.

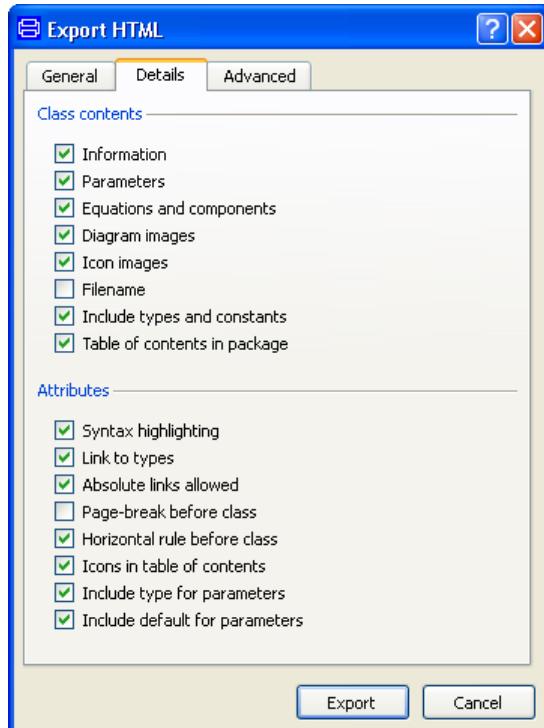
The **Diagram image size** group handles the fact that Bitmap images are not scalable and the best size depends on several factors, e.g., the number of components, complexity of graphics and the intended use of the documentation. Dymola uses a combination of heuristic sizing and user control.

Export

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.

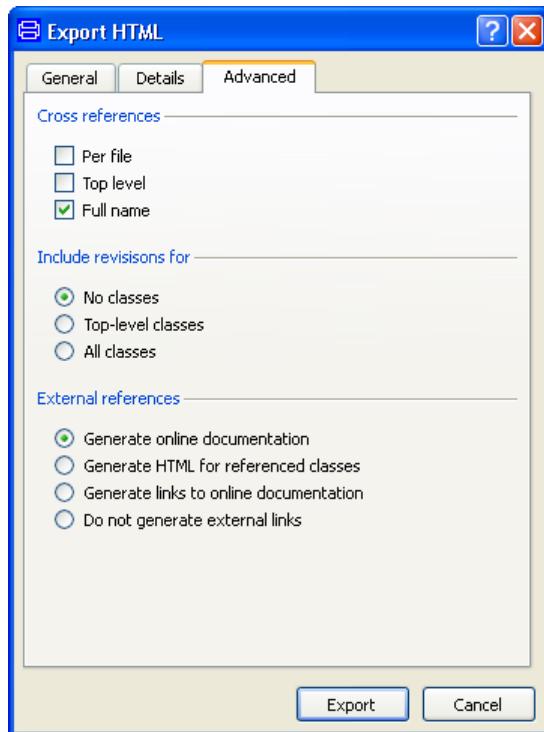
- **Include types and constants** Types and constants is included if this entry is checked.
- **Table of contents in package** Table of contents is generated with the description of the package if this entry is checked.

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.SIunits 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 if 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.

## 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. Please observe that the ModelManagement option is required for this. The following choices are then 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 223. 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` 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.

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` 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.

### **File > Export... > Encrypted Total Model...**

This command is only available if the option Model Management is available. Please see the manual “Dymola User Manual Volume 2”, chapter “Model Management”, section “Encryption in Dymola”.

### **File > Export... > Encrypted File...**

This command is only available if the option Model Management is available. Please see the manual “Dymola User Manual Volume 2”, chapter “Model Management”, section “Encryption in Dymola”.

### **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.
- Script file format (.mos), containing all commands given but without the output from the commands.

Since the command window is by default available in Simulation mode, and usually is mostly used in that mode, a more extensive description of the alternatives in **File > Save Log...** is found in next chapter.

### **File > Generate Script...**

Saves various settings in a script file. Please see corresponding section in next chapter for more information.

### **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 **File > Open...** (see section “File > Open...” on page 293) or **Simulation > Run Script...** (See also next chapter, section “Editor command reference”, sub-section “Main window: Simulation menu”, command “Simulation > Run Script...”). Selecting one of the files opens it again, typically to open models after **File > Clear All** or to re-run a script. 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.

## 4.6.2 Main window: Edit menu

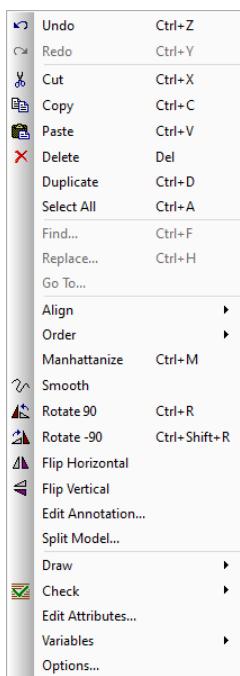
### Toolbar



The toolbar contains buttons for the drawing operations, to toggle gridlines, to toggle connect mode, and to handle transitions and states in state machines. The final button checks the model for errors.

The toolbar can be displayed/hidden using **Window > Tools** and selecting/deselecting **Drawing**. This toolbar is also possible to show in Simulation mode, but cannot be used in that mode.

Please see section “Creating graphical objects” starting on page 184 for more information about the buttons for drawing. Concerning handling of state machines, please see section “State Machines” starting on page 235.



### Edit > Undo and Edit > Redo

Undoes or redoes the previous editing operation.

### Edit > Cut

Copies the current selection to an internal clipboard and deletes them. Text is copied to the operating system clipboard.

### Edit > Copy

Copies the current selection to an internal clipboard without deleting them. Text is copied to the operating system clipboard.

### Edit > Paste

Pastes objects from the internal clipboard to the current window. The current selection is not replaced by the pasted objects.

## **Edit > Delete**

Deletes the current selection. The objects are not placed in the internal clipboard. See also “Deleting objects” on page 164 regarding deletion of connections.

## **Edit > 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.

## **Edit > Select All**

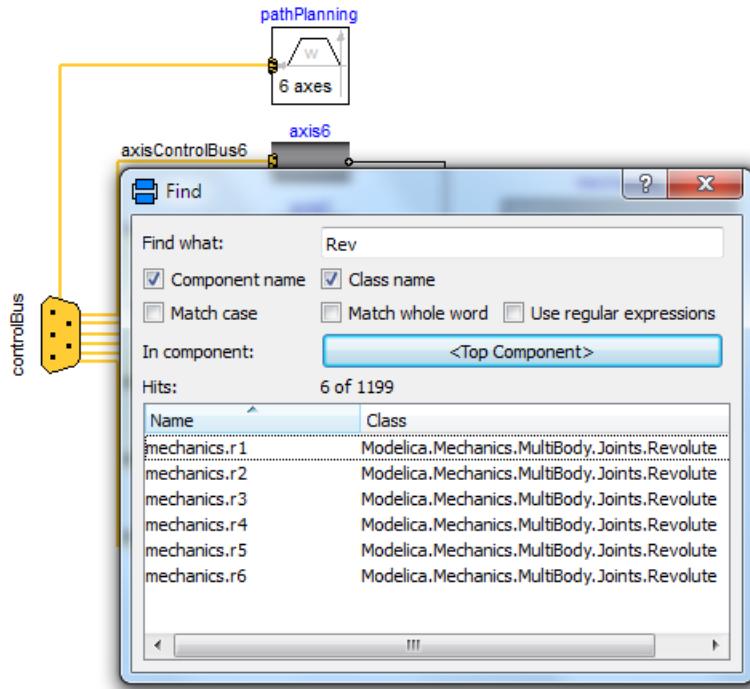
Selects all objects in the current window. Note that it also selects objects inherited from base classes.

## **Edit > Find...**

The result of the command **Edit > Find...** differs depending on what is displayed in the edit window when applying the command.

### **Edit > Find... - in diagram layer**

Being in the diagram layer, the command **Edit > Find...** will search for components in the diagram layer. In the example below, Rev is searched in the Robot model:



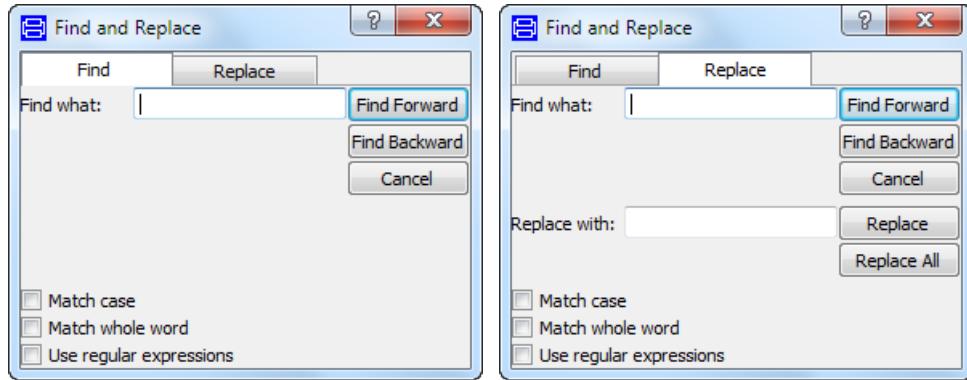
Some features of this command:

- 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 the **In component** button, or you can modify the search.

#### Edit > Find and Edit >Replace – in text layer

**Find and Replace** finds and replaces text in the text editor. Enter the text you search for and press **OK**.

The **Find and Replace** operation will automatically scroll the window to make the found text visible. If the hit is inside an annotation, that annotation will be expanded. **Find/Replace** in Modelica text gives warning if search string is not found.



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.

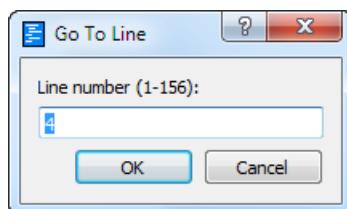
This command is used inside a class. To search for classes, the command **File > Search...** can be used. Please see section “File > Search...” on page 295 for more information.

### **Edit > Replace...**

Please see above.

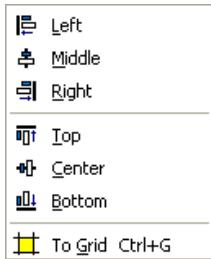
### **Edit > Go To...**

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.

## Edit > Align



The alignment 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.

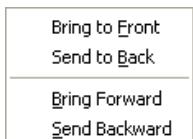
The majority of commands align graphical objects with each other. The first selected object is used as reference (and does not move).

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**.

## Edit > Order



The ordering operations are used to move graphical objects or components 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.

## Edit > 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.

## Edit > Smooth



Converts sharp corners to rounded corners if active. Smooth corresponds to an annotation. Please see the command Edit > Edit Annotation below.

## Edit > Rotate 90 and Edit > Rotate -90

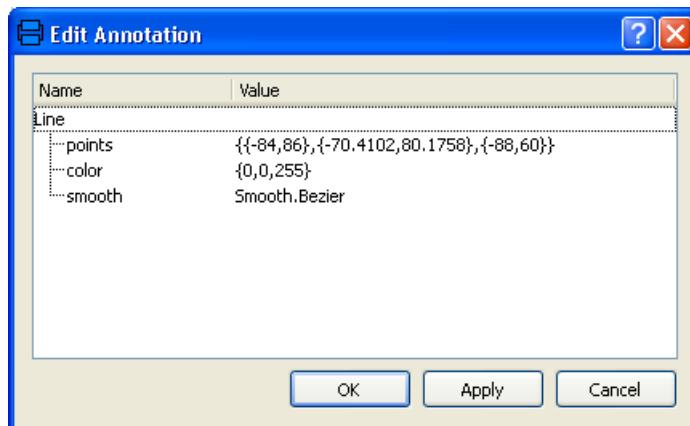
Rotates the selected components 90 degrees clock-wise (counter-clock-wise).

## **Edit > Flip Horizontal and Edit > Flip Vertical**

Flips the selected components left-right (up-down).

## **Edit > Edit Annotation...**

Makes it possible to look at/edit the annotations of the selected object(s). An example of a menu of a line:



The first line displays what type of object is selected.

**points** gives the coordinates for the points in the object.

**color** gives the RGB color of the object.

**smooth** will convert sharp corners to rounded corners if active (this is the case in the example, if not Smooth.None will be displayed).

Please see section “Changing graphical attributes” on page 281 for more information about this.

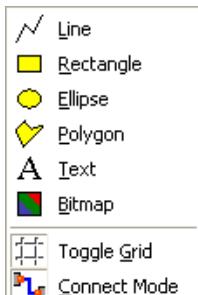
## **Edit > Split Model...**

Creates a submodel or base class from selected components. Please see section “Splitting models” starting on page 239 for more information about this.

## **Edit > Create Local State...**

Creates a state in a state machine. For more information about state machines, see section “State Machines” starting on page 235.

## Edit > Draw

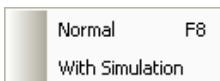


These are the commands to interactively draw new shapes, and related setup. See also “Creating graphical objects” on page 184.

- Drawing graphical shapes (line, rectangle, ellipse, polygon, text and bitmap).
- Show gridlines to make alignment of objects easier.

Toggle connect mode, see section “Connections” on page 177.

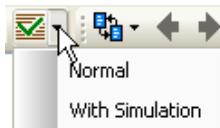
## Edit > Check



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.

A selection whether **Normal** or **With simulation** should be used can be done either by the command or by a drop-down list to the check button.

**Normal** is default when clicking on the **Check** button.



**With Simulation** means that associated commands will also be included in the check. The commands will be executed and the model will be simulated with stored experiment setup. Please see section “Commands > Add Command” on page 324 for more information on associated commands.

### 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 next chapter, the command “Simulation > Setup...”, the Translation tab.

### Handling of built-in functions

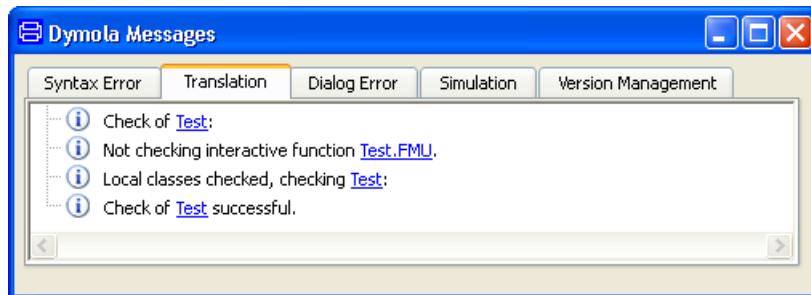
Check will warn for built-in functions not being defined in the Modelica Language Specification. Since a number of built-in functions not being defined in the Modelica Language Specification are present, an annotation `annotation(__Dymola_interactive=true)` can be used when wanting to use such functions 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
    annotation(__Dymola_interactive=true);
    algorithm
      translateModelFMU("Modelica.Mechanics.Rotational.Examples.
        CoupledClutches", true);
    end FMU;
end Test
```

The result of checking the package will be:



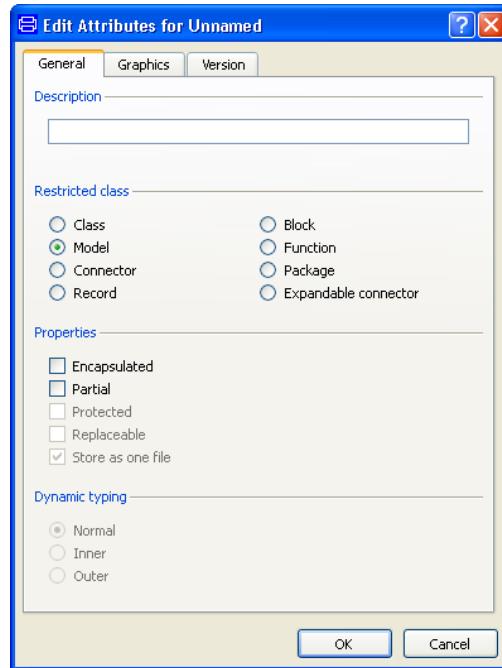
(Checking only the function separately will still report a warning.)

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.

**General model attributes.**

## Edit > Edit Attributes...

### General tab

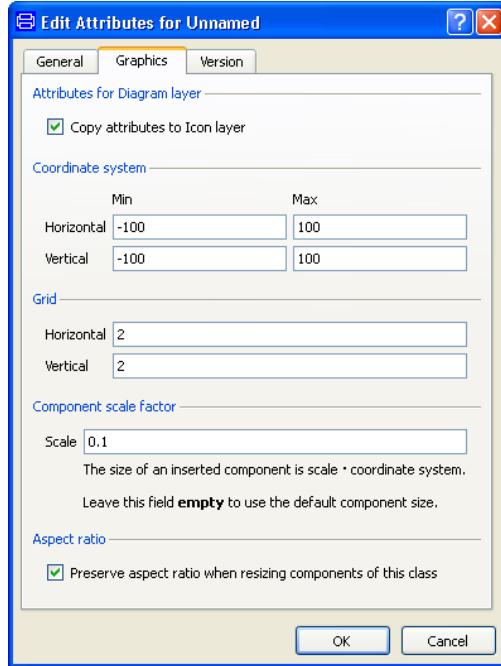


The **General** tab contains the (editable) description of the model and options related to model properties specified in the Modelica language.

- Restriction of the general class (e.g. model or connector).
- Aspects related to protection and redeclaration. It e.g. possible to select if the class is stored in one file or hierarchically.
- 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 **Edit 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).

The **Coordinate system** group is used for changing the coordinate system of the current class. See also “Coordinate system” on page 152.

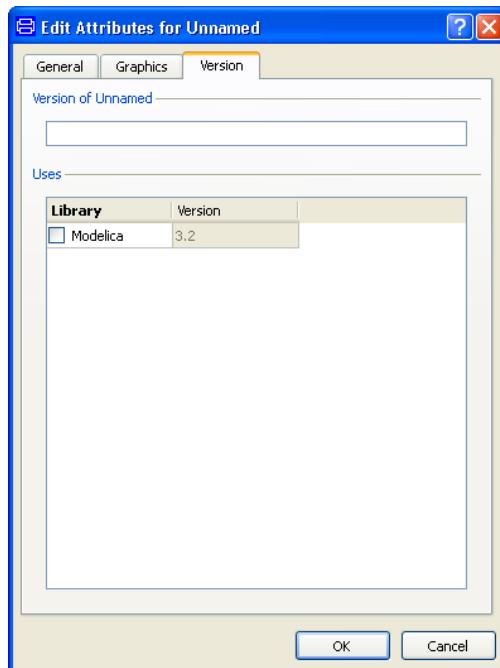
The **Grid** group is used for changing the grid size of the current class.

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.**



The **Version** tab is only available if the option Model Management is included in Dymola.

Please see manual “Dymola User Manual Volume 2”, chapter “Model Management”, section “Version management” for more information.

### Edit > Variables

Giving this command when a component is selected could give the following:



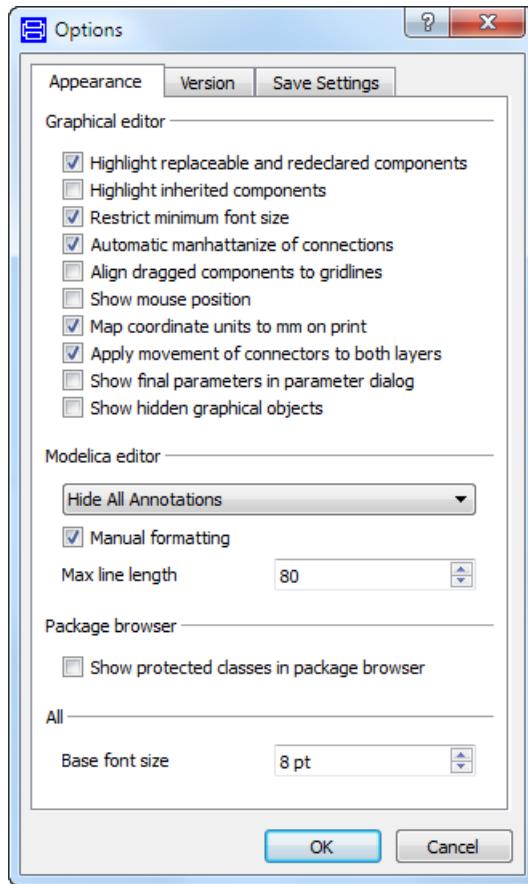
The command **New variable...** can be used to declare new variables of the built-in types Real, Integer, Boolean and String. The command will pop up a variable declaration dialog without any predefined variable. Please see section “Variable declaration dialog” starting on page 264.

The other entries are the variables in the selected component. By selecting any of these the variable declaration dialog for that variable is shown.

## Edit > Options...

### Appearance tab

#### The drawing toolbar.



The **Appearance** tab contains a number of options that affect the operation of the graphical editor.

#### Graphical editor 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. Re-declared components are shown raised, as mounted in the socket. Components of a replaceable class are outlined with a dashed line.

**Highlight inherited components** Components and connectors inherited from a base class are highlighted if this entry is checked. Such components cannot be deleted, moved or replaced.

**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 Manhattanize of connections** If enabled, connections are manhattanized immediately when created, moved or reshaped.

**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.

**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.

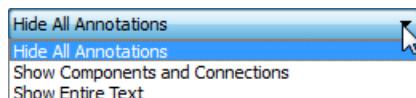
**Apply movement of connectors to both layers** will, if enabled; result in that movement of connectors will be applied to both the icon layer and the diagram layer if the position of the connectors is initially the same in both layers.

**Show final parameters in parameter dialog** will, if enabled, show final (read-only) parameters in the parameter dialog.

**Show hidden graphical objects** will, if enabled, show hidden graphical objects (e.g. hidden graphical connections).

#### Modelica editor group

The expand mode is selectable using a drop-down menu:



**Manual formatting** decides whether the formatting of the class author is used in the Modelica editor, or whether automatic formatting is used.

**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. For information about how to change the max line length value locally in a Modelica text editor window, see “Adjustable maximum line length” on page 203.

#### Package browser group:

The following option affects the package browser.

**Show protected classes in package browser.** 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.

#### All group:

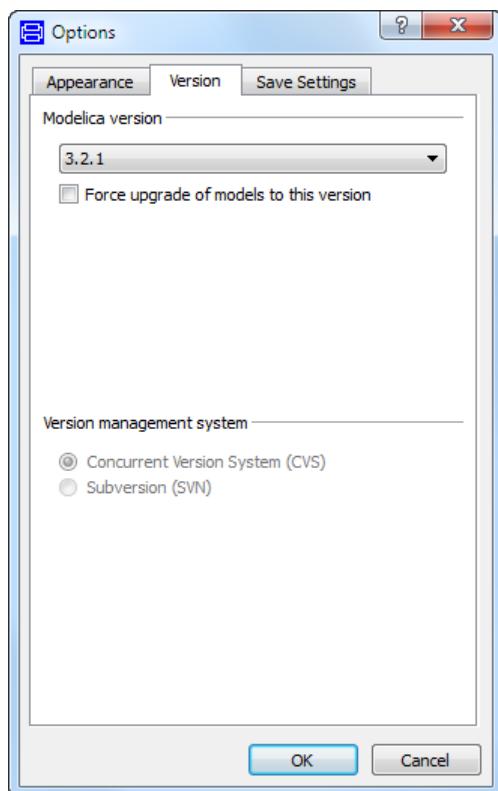
**Base font size** Dymola by default uses the system font size for text in menus, Modelica text etc. If this size is too small when for example presenting Dymola for an audience, the base font size can be changed.

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 (x86)\Dymola 2015 FD01\bin\Dymola.exe" -  
fixedfont "Lucida Console"
```

#### Version tab

##### Version handling.



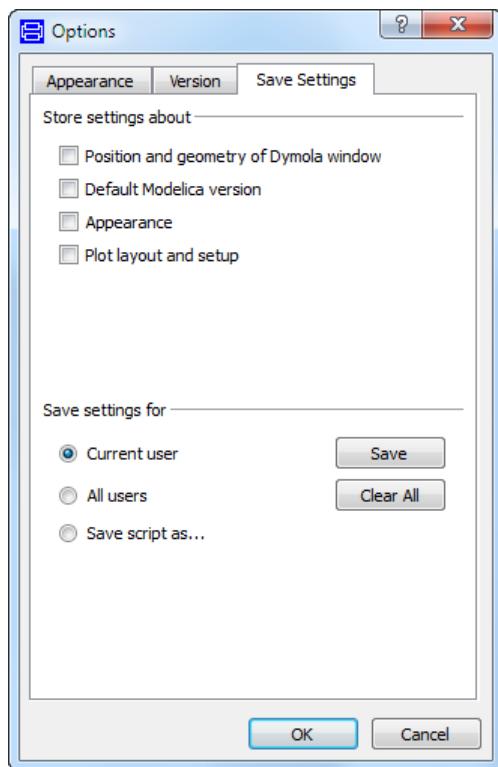
Sets the default version of the Modelica Standard Library and whether models should be upgraded to this version. This allows the user to quickly switch between some Modelica versions (e.g. Modelica 2.2.2 and Modelica 3.2.1). By setting the "Force" flag, models are required to be updated to the currently selected version of Modelica; if not set, the version of

Modelica that the model uses is loaded on demand. (If the version of the model is older than Modelica 2.2.2 Dymola suggest updating to the latest Modelica version available. If update to e.g. Modelica 2.2.2 is wanted, please select Modelica 2.2.2 and set the "Force" flag before opening the model.)

If the “Model Management” option is available, Dymola supports version control of models using two commonly used systems, Concurrent Version System (CVS) and Subversion (SVN). Please see “Dymola User Manual Volume 2”, chapter “Model Management”, section “Version management”.

### Save settings tab

#### Settings for saving.



The command **Save** will save the selected settings in a file associated with the user. The next time Dymola is started, the saved settings are used. The following can be ticked to be saved: layout information includes:

- **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.
- **Default Modelica version.**
- **Appearance.** This includes all settings in the **Appearance** tab, except the following:

- **Map coordinate units to mm on print.**
- **Apply movement of connectors to both layers.**
- **Show protected classes in package browser.**
- **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 **File > Generate Script...** and the built-in function `saveSettings`. Please see next chapter for more information.

The setup information is by default stored in a system directory associated with the user. On Windows, according to the recommendations of Microsoft, the name is typically

```
C:\Documents and Settings\<user>\Application Data\Dynasim\
setup.mos
```

The directory Dynasim is automatically created by Dymola if needed. It is also possible to store the settings in a directory read by all users, or to save the settings as a local script file chosen by the user.

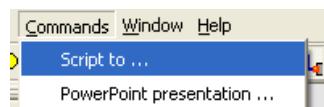
The command **Clear All** will erase the settings file for the current user or all users. Defaults will apply until settings are saved.

### 4.6.3 Main window: Commands menu



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.

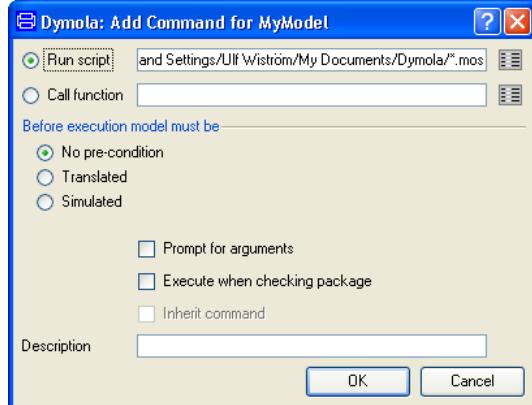
It is possible to include calls to scripts and opening of documents in such commands



Commands in the Commands menu can be inheritable to models extending from it. The command must either specify 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.

## Commands > Add Command

### Adding a command.



This command is used to associate commands with models.

**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 **File > Generate Script...** the script file can be saved as a command ticking **Store** in model as command. Please see section “File > Generate Script...” on page 307 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.

These commands are added to the **Command** menu for the current model.

Demands on the state of the model can be specified:

**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.

**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.

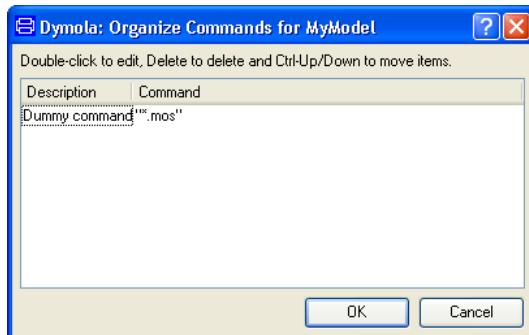
**Execute when checking package** specifies that the commands should be part of check. Please see the section “Edit > Check” on page 314 for more information on how to perform such a check.

**Inherit command** allows extended classes to inherit the command. This requires that the function/script does not explicitly use the name of the model.

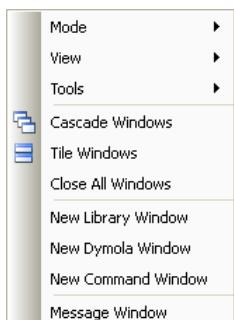
**Description** will be the name of the command/script in the Command menu.

### Commands > Organize commands

#### Organizing commands.



To edit a command, double-click on it. To delete it or to move it up/down, click on the command and use the **Delete** key to delete it or **Ctrl+Up** or **Ctrl+Down** to move the command to a higher or lower position in the list.



### 4.6.4 Main window: Window menu

#### Toolbar



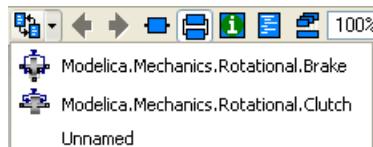
The toolbar contains buttons showing recent models, previous and next model viewed in the window and for viewing different layers of the model (see also “Class layers” on page 146). The last control sets the zoom factor.

The toolbar can be displayed/hidden using **Window > Tools** and selecting/deselecting **View**. This toolbar is also possible to show in Simulation mode, but then contains fewer possibilities.

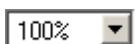
## Recent Models



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.

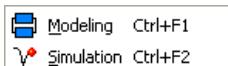


## Zooming



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. Scrollbars then appear to make it possible to move around in the model. For more alternatives for moving and zooming, please see section “Moving and zooming in the diagram layer” on page 401.

## Window > Mode > Modeling (Ctrl+F1)



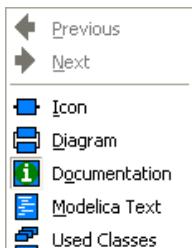
Changes window to modeling mode. This mode is used to compose new models or to change existing models.

## Window > Mode > Simulation (Ctrl+F2)

Changes window to simulation mode. This mode is used to set up an experiment and simulate a model. The results can be plotted and visualized with 3D animation.

## Window > View > Previous

Shows the previous model viewed in the editor.



## Window > View > Next

Shows the next model selected, if **Window > View > Previous** has been used before.

## Window > View > Icon

Displays the icon layer of the class.

## Window > View > Diagram

Displays the diagram layer of the class.

## Window > View > Documentation

Displays the description and documentation of the class (the documentation layer of the class).

## **Window > View > Modelica Text**

Displays the declarations and equations of the class (the Modelica Text layer of the class).

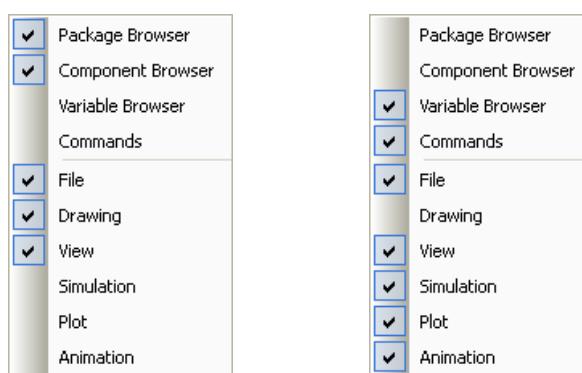
## **Window > View > Used Classes**

Displays the base classes, classes used for components and connectors, and the Modelica text for the class. Note that this text cannot be edited.

## **Window > Tools**

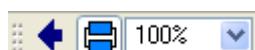
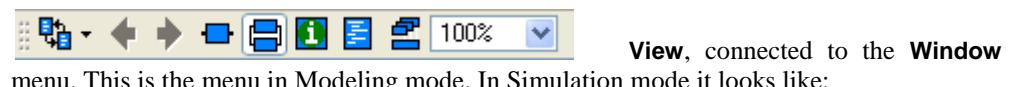
This menu controls what should be shown in the Dymola Main window when it comes to sub-windows and toolbars. By default different selections are made in Modeling and Simulation mode (the latter correspond to the figure to the right below).

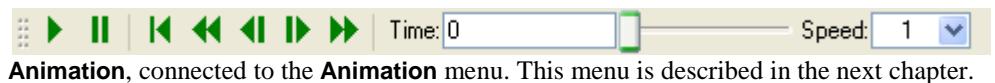
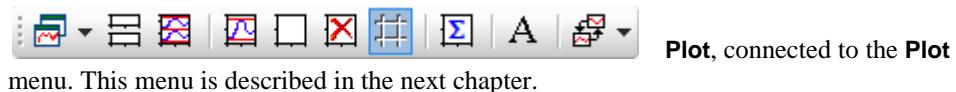
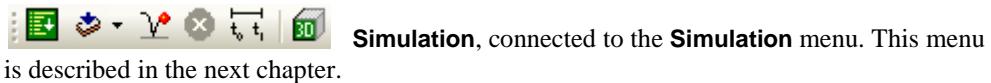
### **Window tools in Modeling and Simulation mode.**



The first part selects what sub-windows should be shown in the Dymola Main window. More information about these windows is given in the beginning of this chapter and the next chapter.

The second part selects which toolbars should be visible in the Dymola Main window. More information about these can be obtained by looking at the corresponding menu (see the list below) in the reference part of this and the next chapter. Please observe that they can contain different selections in Modeling and Simulation mode, respectively. The toolbars are:





**Line up** will arrange the toolbars by e.g. eliminating space between them.

### **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 Windows**

Closes all windows in the Edit window.

### **Window > New Library Window**

Creates a new library window, see “Library window” on page 158.

### **Window > New Dymola Window**

Creates a new modeling and simulation window.

### **Window > New Command Window**

Creates a new command window.

### **Window > Message Window**

Displays the message window.

## **4.6.5 Main window: Help menu**



Please note that Help can be achieved in a number of ways. Please see the chapter “Getting started with Dymola”, section “Help and information”.

### **Help > What's This?**



Displays interactive help for the graphical user interface. After selecting **Help > What's This?** the cursor changes. Now click on the item in the user interface you want information about. Please note that other ways are available. Please see the reference above.

### **Help > Documentation**

Opens a web browser window with the root of the online documentation, which contains the Dymola manual, articles and links to other resources.

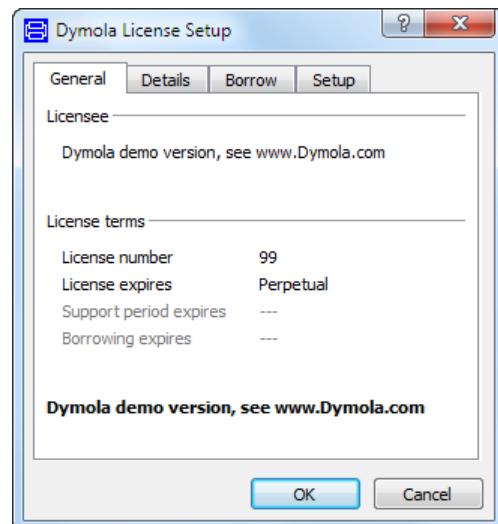
### **Help > Dymola website**

Opens a web browser window with the Dymola webpage.

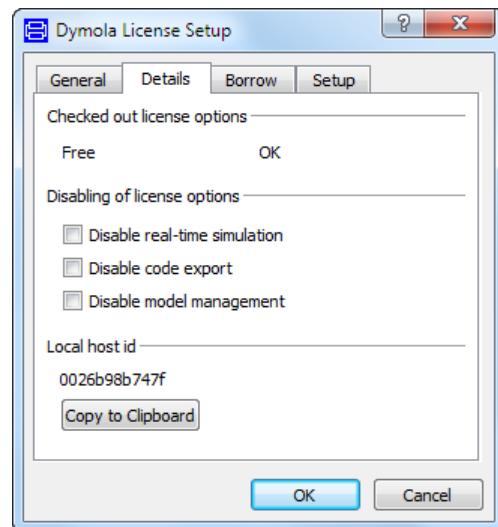
### **Help > License...**

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 demo version of Dymola installed.

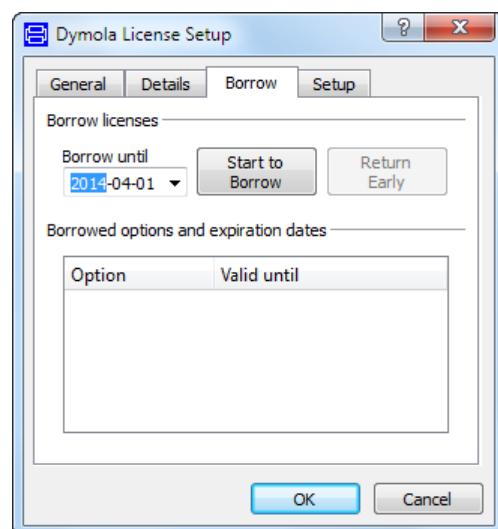
#### **General tab**



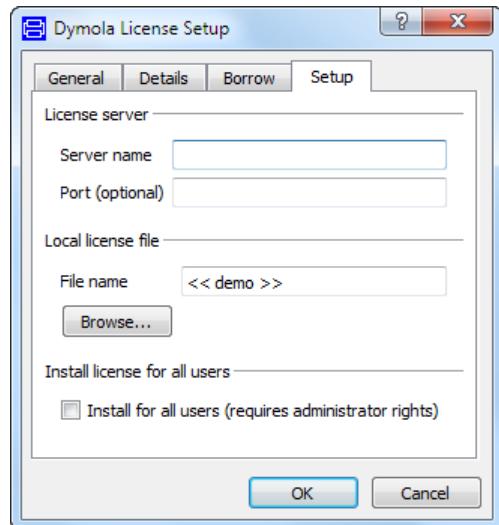
### Details tab



### Borrow tab



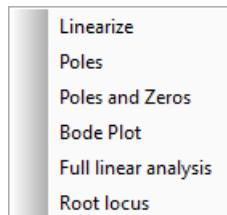
### Setup tab



### Help > About Dymola

Displays version, copyright and license information about Dymola.

## 4.6.6 Main window: Linear analysis menu



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.

### Linear analysis > Linearize

Linearize a model and return the linearized model as StateSpace object.

### Linear analysis > Poles

Linearize a model and plot the poles of the linearized model.

### Linear analysis > Poles and Zeros

Linearize a model and plot the poles and zeros of the linearized model.

### **Linear analysis > Bode Plot**

Linearize a model and plot the transfer functions from all inputs to all outputs of the linearized model as Bode plots.

### **Linear analysis > Full linear analysis**

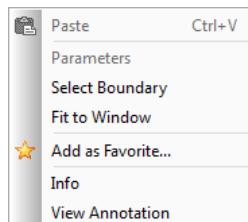
Linearize a model and perform all available linear analysis operations.

### **Linear analysis > Root locus**

Compute and plot the root locus of one parameter of a model (= eigenvalues of the model that is linearized for every parameter value).

## **4.6.7 Context menu: Edit window (Icon layer)**

The icon layer of the Edit window is introduced in “Icon layer” on page 147.



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:

**Paste** will paste the content in the clipboard, if any.

**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 152.

**Fit to Window** will set the zooming to 100% (default). For more information about zooming, see section “Zooming” on page 161.

**Add as Favorite...** adds the open class to favorites. See the section “Creating a favorite package” on page 164 for details.

**Info** shows the information of the package/component that the diagram layer is a part of.

**View Annotation...** makes it possible to view the annotations of all objects present in the icon layer. For more information about annotations, please see section “Editing graphics using Edit Annotation” on page 281.

## **4.6.8 Context menu: Edit window (Diagram layer)**

The diagram layer is introduced in “Diagram layer” on page 148.

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 menu looks exactly as the one in the previous section and is analog to that.

## 4.6.9 Context menu: Edit window (Documentation layer)

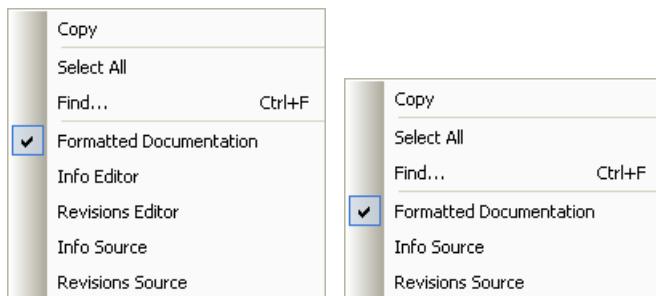
The documentation layer is introduced in “Documentation layer” on page 149. More information, including how to edit the information, is available in section “Documentation” starting on page 215.

### Context menu when viewing formatted documentation

Right-clicking in the documentation layer of the edit window presents a context menu with common text operations.

The commands available depend on whether the documentation layer is editable or not.

**Context menu for the Documentation layer; editable and non-editable.**



**Copy** copies text from the documentation layer to the clipboard.

**Select All** selects all text in the documentation layer.

**Find...** finds and replaces text in the text editor. Enter the text you search for and press **OK**.

**Formatted documentation** is the default setting, showing the result of any editing.

**Info editor** (available only if not read-only) switches to a documentation editor for viewing/editing the information part of the documentation.

**Revisions editor** (available only if not read-only) switches to a documentation editor for viewing/editing the revisions part of the documentation

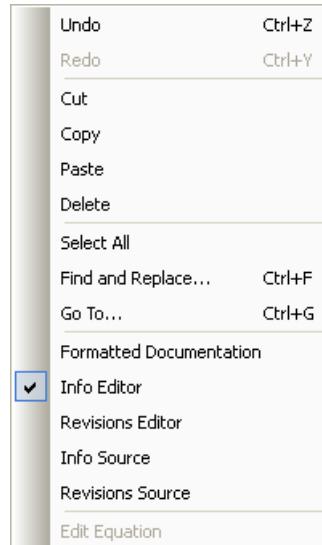
**Info source** will display the HTML code for the information part for viewing (or, if not read-only) editing.

**Revisions source** will display the HTML code for the information part for viewing (or, if not read-only) editing

### 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 **Edit** menu.

**Context menu for  
editable info/revisions  
part of documentation  
layer.**



**Undo**, **Redo** undoes and redoes the last editing operation in the editor. Same as **Edit > Undo** and **Edit > Redo**.

**Cut**, **Copy**, **Paste** copies text between the clipboard and the editor.

**Delete** deletes selected text.

**Select All** selects all text in the documentation layer.

**Find and Replace...** finds and replaces text in the text editor. Enter the text you search for and press **OK**.

**Go To...** sets the cursor at a specified line and scrolls the window if necessary.

**Formatted documentation** switches to showing the result of any editing; that is, the default documentation layer mode.

**Info editor** switches to a documentation editor for viewing/editing the information part of the documentation.

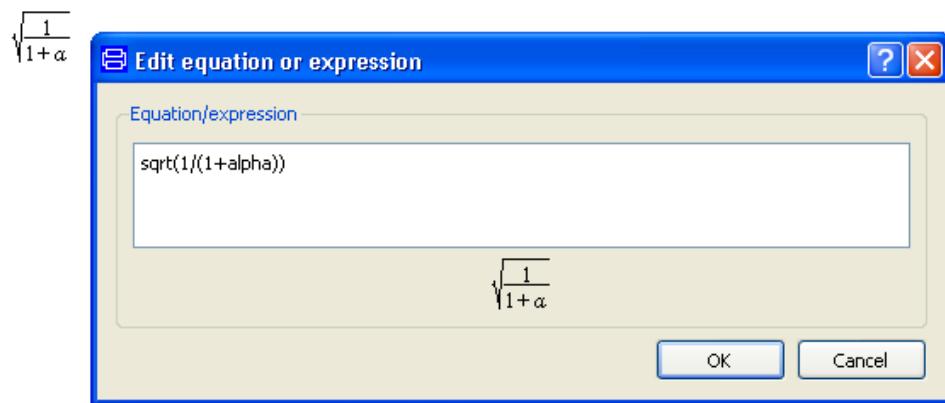
**Revisions editor** switches to a documentation editor for viewing/editing the revisions part of the documentation

**Info source** will display the HTML code for the information part for viewing or editing.

**Revisions source** will display the HTML code for the information part for viewing or editing

**Edit equation** enables editing an equation or expression created using the button **Insert equation or expression**. If such an equation/expression is selected, this entry will pop an editor for the equation/expression.

**Edit equation on a square root expression.**

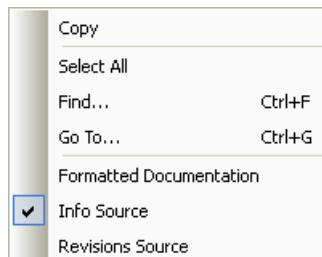


The result of the editing is shown below the editing pane. If an equation/expression is not concluded, the text Incomplete equation/expression is displayed. Clicking **OK** the edited equation/expression will replace the previously selected one.

#### **Context menu for read-only 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:

**Context menu for read-only HTML source code for info/revision part.**



For information about the entries, please see previous section.

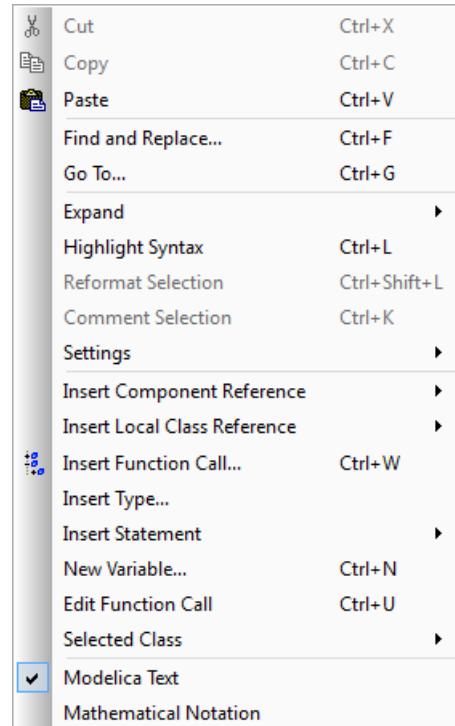
#### **4.6.10 Context menu: Edit window (Modelica Text layer)**

The Modelica Text layer is introduced in “Modelica Text layer” on page 150. When it comes to editing, please see “Programming in Modelica” on page 193.

If **Use 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 **Edit** menu.

**Context menu of the Modelica Text layer.**

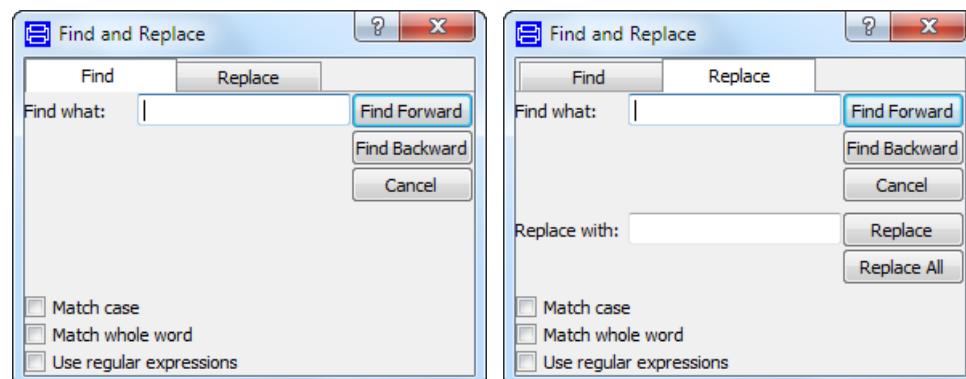


**Cut, Copy, Paste** copies text between the clipboard and the editor. It is possible to copy text with hidden annotations to clipboard.

**Select All** selects all text in the Modelica text layer.

**Find and Replace** finds and replaces text in the text editor. Enter the text you search for and press **OK**.

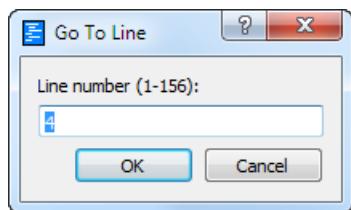
The **Find and Replace** operation will automatically scroll the window to make the found text visible. **Find/Replace** in Modelica text gives warning if search string is not found.



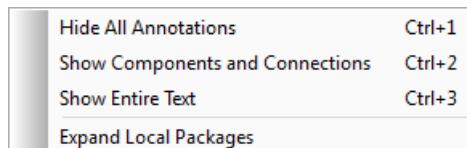
Regular expressions can be used in the search if this checkbox is checked. Special symbols in the regular expression are

*	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.



**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 “+”. See also section “Local packages” on page 197.

**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 message window.

**Reformat Selection** makes it possible to auto-format (pretty print) a selection of the text.

**Comment Selection** can be used to comment out selected rows.

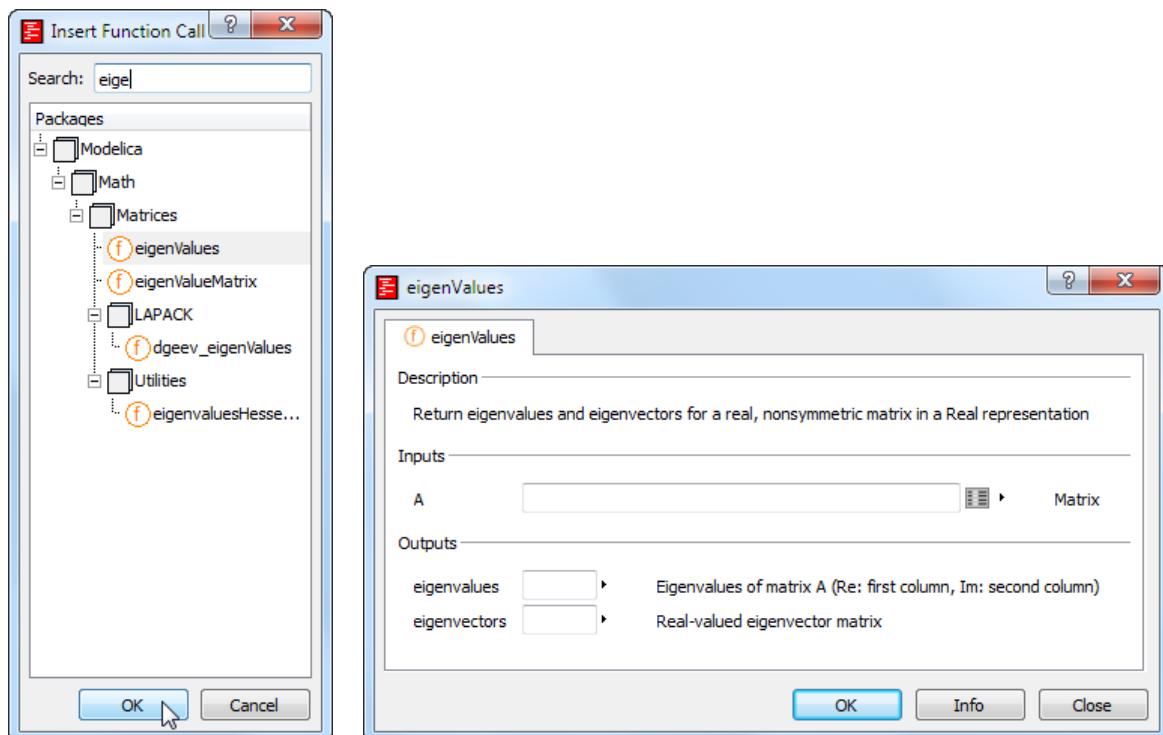


**Settings** The only entry here, **Manual formatting**, is by default activated and presents the documentation according to formatting implemented by the writer of the class documentation. If automatic formatting should be used instead, this alternative should be unchecked.

**Insert Component Reference** enables inserting a component reference in the text.

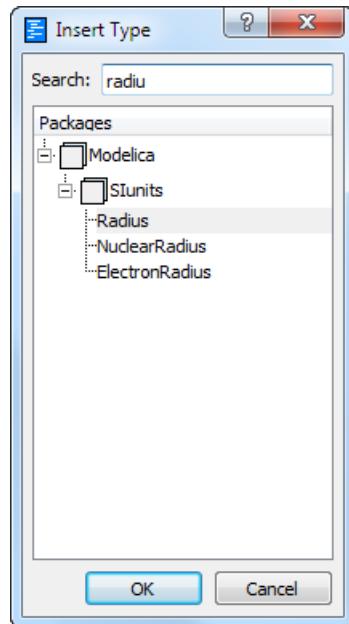
**Insert Local Class Reference** enables inserting a local class reference in the text.

**Insert 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):



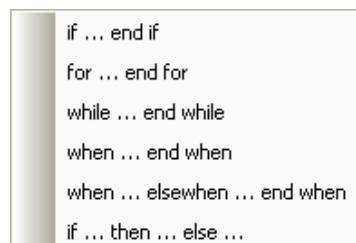
For more information how to use this function, please see section “Handling of function calls” starting on page 210.

**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.SIunits.Radius` will be inserted as text.

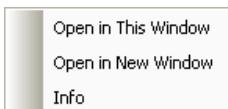


For more information how to use this function, please see section “Parameters, variables and constants” starting on page 207.

**Insert Statement** presents a sub-menu with common Modelica constructs. This makes it easier to enter Modelica code.



New Variable... allows you to define a new variable using the variable declaration dialog. For details of this dialog, see section “Variable declaration dialog” starting on page 264.



**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.

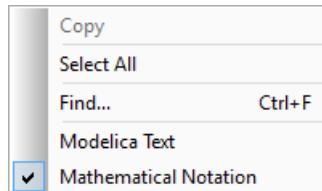
**Selected Class** allows you to open the selected class, or its documentation, by selecting the class name (or putting the cursor inside the name), right-clicking and then selecting the command.

**Modelica Text** is the default selection; presenting in plain text.

**Mathematical Notation** will change the editor to present the content in mathematical notation; formulas etc. will be presented as formulas instead of plain text. The context menu is presented in next section. Please note that this mode is only for presentation, no editing can be done in this mode.

#### 4.6.11 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 198.



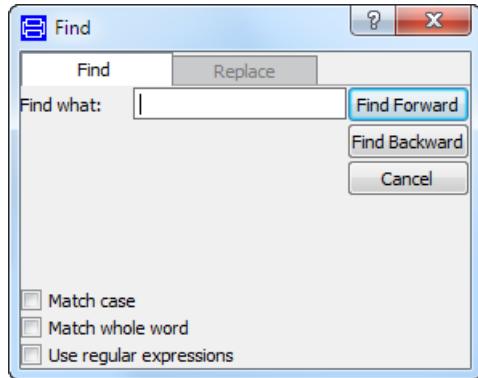
**Copy** will copy selected parts to the clipboard. Please note that there is no **Paste** in this menu, copied parts can be inserted in e.g. Microsoft Word for further elaboration. Text parts will be inserted as text in Microsoft Word, equations will be inserted as a picture.

(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.

**Find...** finds and replaces text in the text editor. Enter the text you search for and press **OK**.

The **Find...** operation will automatically scroll the window to make the found *text* visible. **Find/Replace** in Modelica text gives warning if search string is not found.



**Please note** that **Find...** only works in the text part of the editor – equations are treated as pictures and cannot be searched. Also, the **Replace** tab is not available – the layer is read-only displaying mathematical notation.

Regular expressions can be used in the search. Please see section “Context menu: Edit window (Modelica Text layer)” starting on page 335.

**Modelica Text** will deactivate the presentation mode and return to plain text.

## 4.6.12 Context menu: Edit window (Used Classes layer)

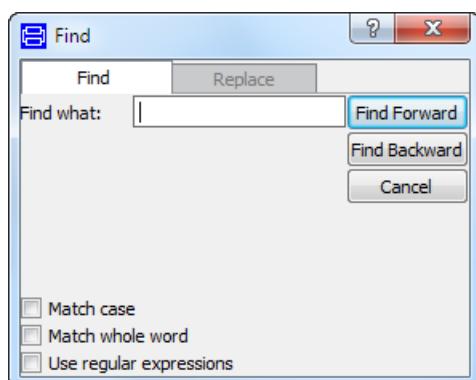
The Used Classes layer is introduced in “Used Classes layer” on page 151.

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.

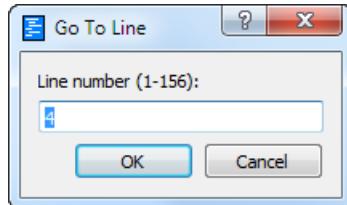
**Select All** – selects all text.

**Find...** – search functionality for a specific text (the **Replace** tab is not available – the layer is read-only). The menu looks like:

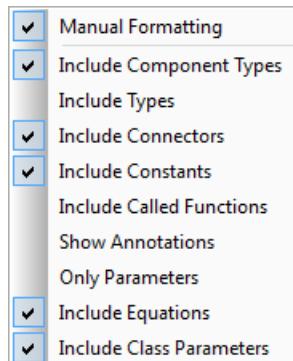


The **Find** operation will automatically scroll the window to make the found text visible. Regular expressions can be used in the search. Please see section “Context menu: Edit window (Modelica Text layer)” starting on page 335.

**Go To ...** – use of the following menu. The number of the current line is shown as default.

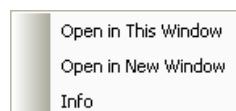


**Settings** – selects what should be seen:



**Manual formatting** is by default activated and presents the documentation according to formatting implemented by the writer of the class documentation. If automatic formatting should be used instead, this alternative should be unchecked.

**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.



**Used Classes** – (default) Displays base classes, classes used for components and connectors, and the Modelica text of the class.

**Flat Modelica** – Displays the Flat Modelica text if the selected item is a model or block, otherwise Used Classes as above (in spite of Flat Modelica being ticked).

## 4.6.13 Context menu: Package browser

The package browser is introduced in “Package and component browsers” starting on page 154.

Right-clicking in an empty space below the components in the package browser gives the menu where the user can decide which windows and layers should be displayed. For information about this menu, please see the section “Window > Tools” on page 327.

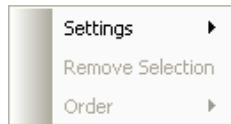
A much more common case is using the context menu of a component in the package browser. Please see the section “Context menu for components in the package browser” below.

Finally, right-clicking **the header** “Packages” in the package browser displays a context menu that will collapse all packages in the package browser if selected.



## 4.6.14 Context menu: Component browser

The component browser is introduced in section “Package and component browsers” starting on page 154.



Right-clicking in an empty space below the components in the component browser gives the context menu. This menu is part of the context menu in the much more common case that a component is selected when right-clicking. Please see the next section for description of the menu entries.

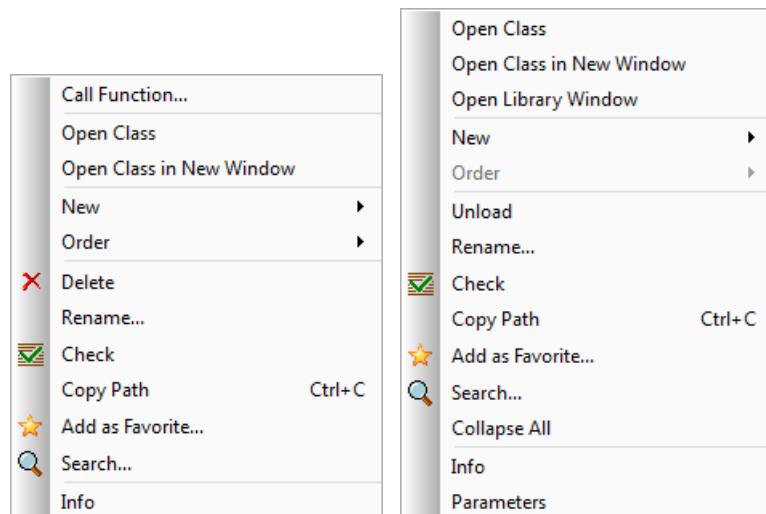
## 4.6.15 Context menu: Components

### Context menu for components in the package browser

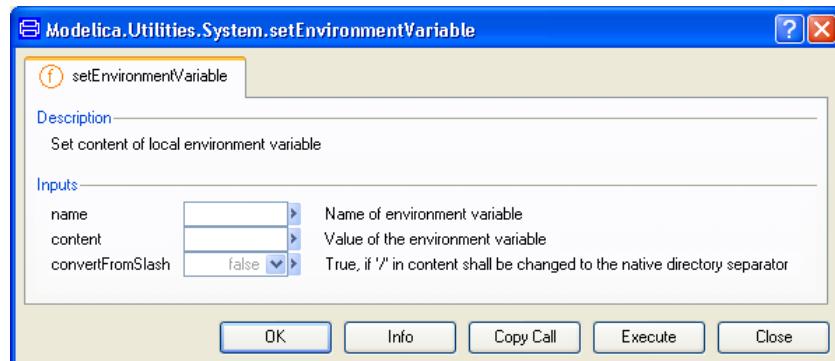
The package browser is introduced in “Package and component browsers” starting on page 154.

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.

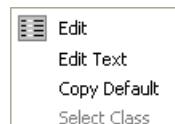
**Context menu for non-protected function and top-level package.**



**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.

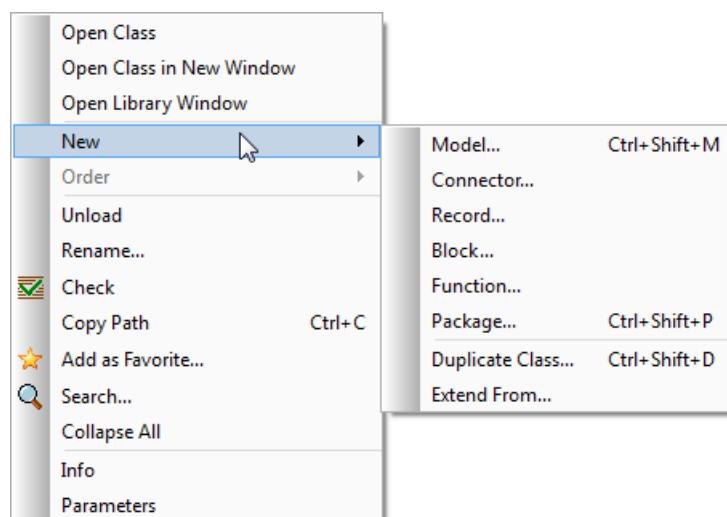
The **OK** button will execute the changes and close the dialog window, while the **Execute** button will execute the changes without closing the dialog window. (Executing the function usually also changes the mode from Modeling to Simulation.) The **Info** button will display information about the function 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.

**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.

**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 291.

**Duplicate Class** Duplicates a class, see “File > New... > Duplicate Class” on page 293.

**Extend From...** creates a new class, which extends from the selected class. See also “File > New... > Model etc.” on page 291.

**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.

**Unload** unloads the class from its enclosing package.

**Rename...** renames a class. This can move the class to a different package. For changing the order of classes within a package use **Order**. Renaming classes (using package context-menu **Edit > Rename**) automatically update all references in all loaded classes to reflect the name-change. There is no undo command for this.

Note that dangling references to the unloaded or renamed class may not be detected. Make **Edit > Check** on the enclosing package to detect some potential problems, see “Edit > Check” on page 314.

Note also that directories pointed to by LibraryDirectory and IncludeDirectory are handled, and the references updated. Other resources, e.g. images, are not copied.

**Check** checks the selected class for errors. See “Edit > Check” on page 314.

**Copy Path** will copy the Modelica path to the class to the clipboard, e.g. MyPackage.MyModel1.

**Add as Favorite...** adds the selected class to favorites. See the section “Creating a favorite package” on page 164 for details.

**Search...** Searches the selected class for all classes or components matching a search pattern. See “File > Search...” on page 295.

**Collapse All** closes an open hierarchy and all lower levels (a package and all sub-packages). Clicking on the “+” for the package in the package browser will close the package, but when “+” is clicked again all sub packages will be open as before. The “Close” will close all sub packages so that they will not open when “+” is clicked.

**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.

**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 170.

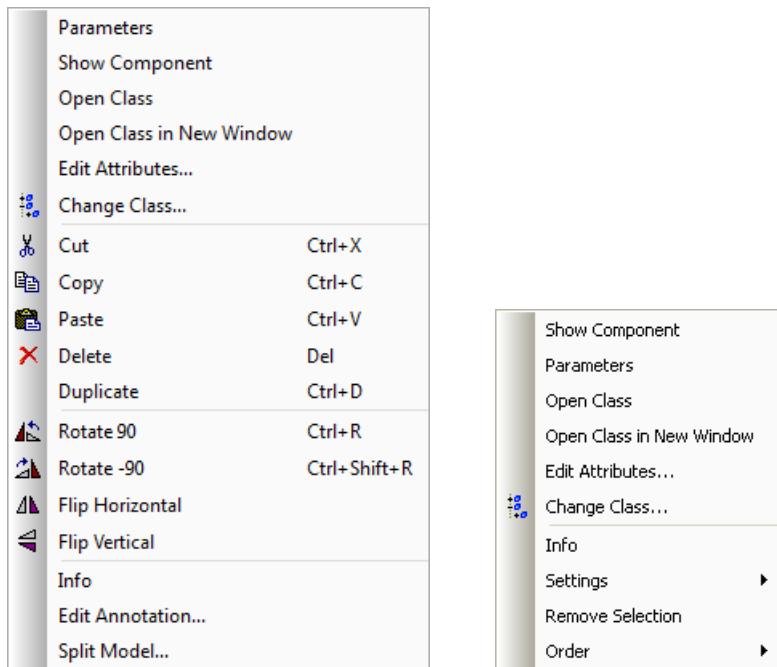
### **Context menu for components in the diagram layer and the component browser**

The context menu of an object in the diagram layer is introduced in “The context menu” on page 169. The component browser is introduced in “Package and component browsers” starting on page 154.

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.**



**Parameters** opens the parameter dialog. For information about the parameter dialog, please see section “Editing parameters and variables” on page 170.

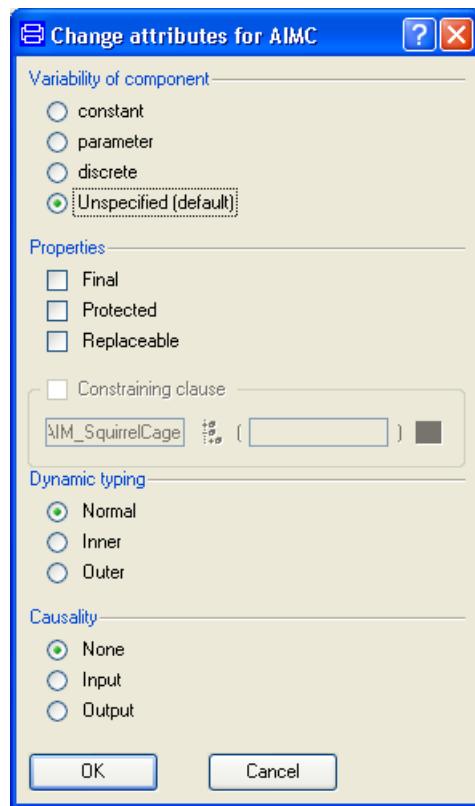
**Show Component** shows the class of the component or connector in this window. Press the Previous button to go back.

**Open Class** opens the class in the present window.

**Open Class in New Window** opens a new main window with the class of the component or connector.

**Edit Attributes...** opens a **Component Attribute dialog**. These attributes reflect component properties specified in the Modelica language. The illustration below shows the default settings.

The component attributes dialog.



The group **Variability of component** is also described when it comes to declaring a variable. Please see the section “Declaration tab” on page 264.

For information about the most of the entries in the groups **Properties**, **Dynamic typing** and **Causality** please see section “Type prefix tab” on page 266. In the **Properties** group the **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.



**Change Class...** It is possible to change class of replaceable component using the context menu for the component. Please see section “Working with replaceable components” on page 247 for more information.

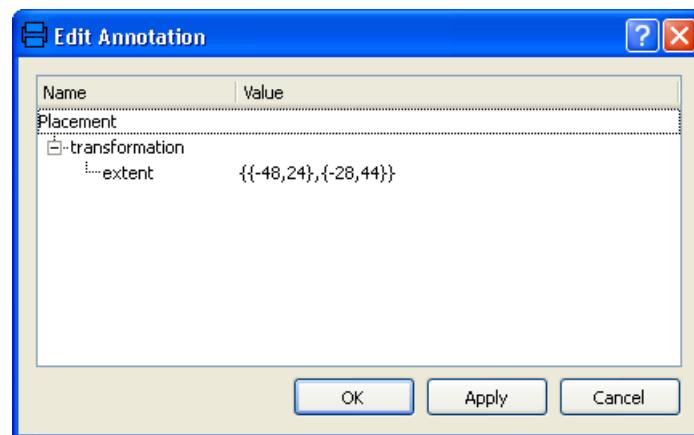
**Cut, Copy, Paste, Delete** and **Duplicate** corresponds to the commands **Edit > Cut**, **Edit > Duplicate** etc. Please see these commands in section “Main window: Edit menu” starting on page 308.

**Rotate 90, Rotate -90, Flip Horizontal** and **Flip Vertical**; please see the above reference.

**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.

**Edit Annotation...** will display a dialog that can be used to change the graphical representation of the object. The example below shows the transformation extent, that is, the location of the two red handles of the object. By clicking on the figures and changing them the extent of the object will be changed when pressing **OK**.

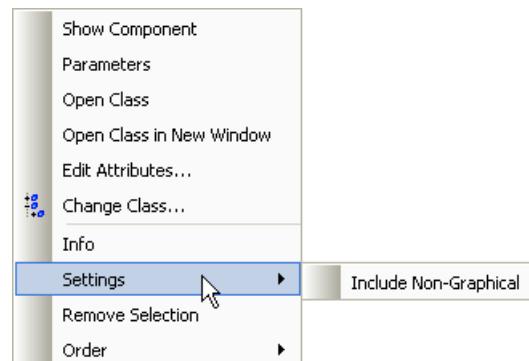
**Edit Annotation of the object.**



**Split Model...** - makes it possible to create a submodel or base class from selected components. Please see section “Splitting models” starting on page 239 for more information.

**Settings** - resting the cursor on this entry displays the following:

**Settings in the component browser.**

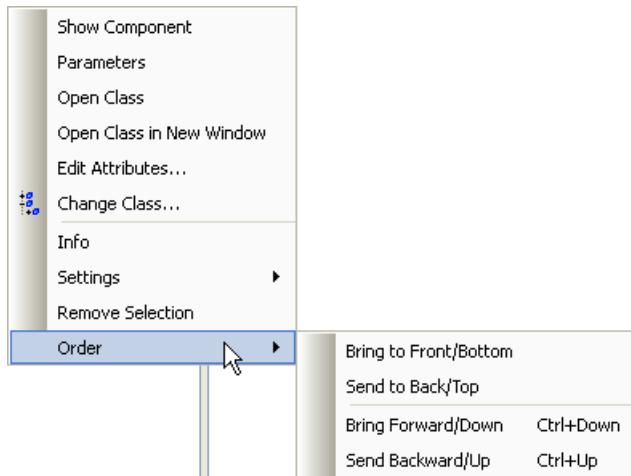


By activating **Include non-graphical** also components without graphical elements (e.g. real variables and constants) will be shown in the component browser. Please note that only such components in the top level of the tree are shown. A checked checkbox will be shown in the menu if this setting is activated.

**Remove selection** This action will remove selected parts of the tree (including all components) in the component browser. If multiple selection should be made the only way to do this is to use multiple select in the diagram layer.

**Order** Resting the cursor over this entry will display:

**Using the Order functionality.**



These commands will work on the component browser as well as the diagram layer (as an example **Bring to Front/Bottom** 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.

#### 4.6.16 Context menu: Coordinate system boundary

The coordinate system is introduced in section “Coordinate system” on page 152.

Right-clicking when the coordinate system boundary is selected displays the context menu.

**Edit Attributes...** displays the Graphics tab of the command **Edit > Edit Attributes....**. See section “Graphics tab” on page 317 for more information.

**Fit to Window** will zoom to 100 %.

#### 4.6.17 Context menu: Connection while connecting

Connections are introduced in “Connections” starting on page 177.

Right-clicking or double-clicking while the connection is being drawn 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.

## 4.6.18 Context menu: Graphical objects

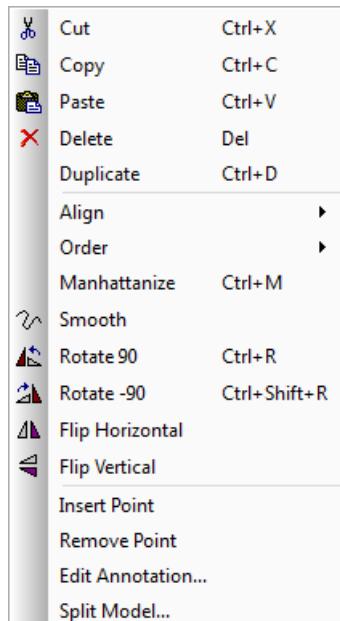
Editing of graphical objects is introduced in “Changing graphical attributes” on page 190.

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 193 for more details.

### General entries in the context menu of graphical objects

Most entries in the context menu of any object are the same as reached by the command **Edit**. As an example, the context menu for a line:



Most commands are the same as reached by the command **Edit**. For such commands, please see section “Main window: Edit menu” starting on page 308.

## **Specific commands in the context menu of various graphical objects**

### **Line**

The commands specific for the line context menu are:

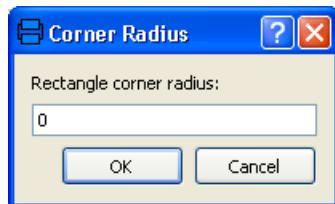
**Insert Point** will insert a new point in the selected object. The point is inserted in the point closest to the cursor in the selected object.

**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 185 for more details.

### **Ellipse**

The command specific for the context menu of an ellipse is **Elliptical Arc...** that displays the following:

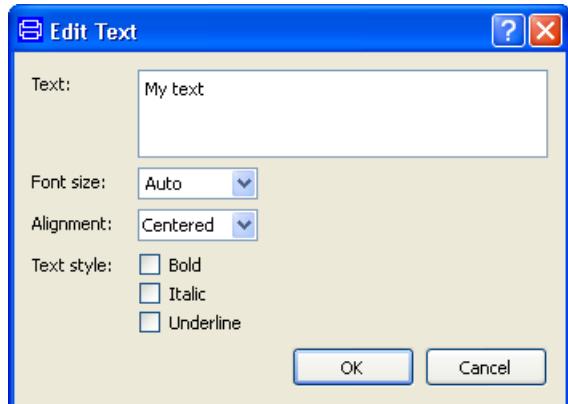


### **Polygon**

The commands specific for the context menu of the polygon are the same as for the line; please see above.

### **Text**

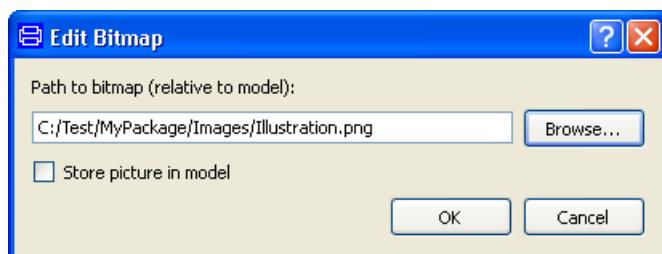
The command specific for the context menu of the text object displaying e.g. “My text” is **Edit text...** that displays the following:



For more information, please see section “Text” on page 186.

### 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 189.

## 4.6.19 Graphical object menus: Line and fill style

Editing of graphical objects is introduced in “Changing graphical attributes” on page 190.

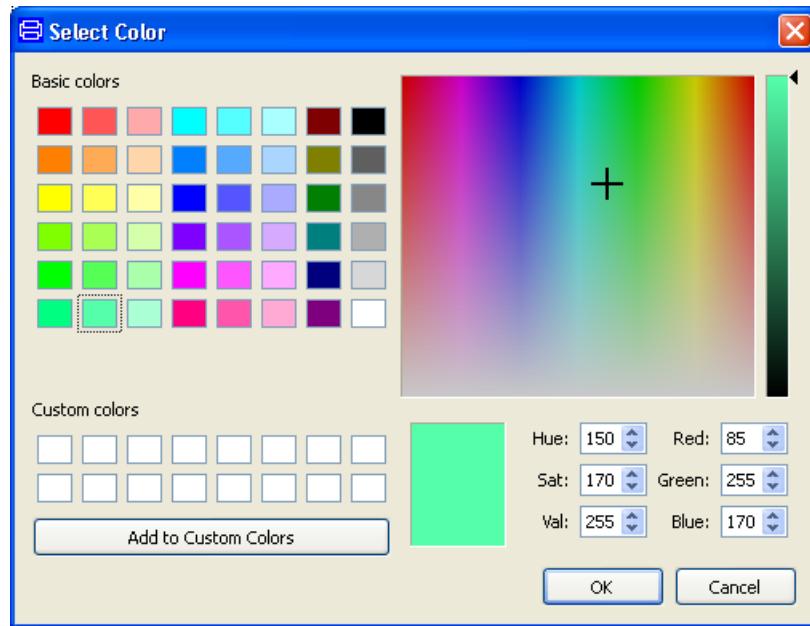
### Line style

The line style attributes are

**No Color** means that the line will be invisible; can be used e.g. to draw a filled area without a border.

**Colors...** Selecting this alternative will give a color palette:





The color can be set in two ways; either by clicking on a color in Basic colors or Custom colors, or by defining the color by specifying hue/saturation/value, (alternatively red/green/blue). 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.

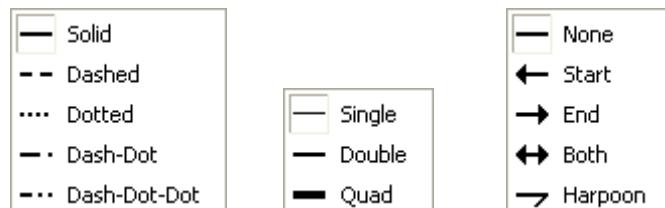
The color selection dialog has a choice of standard colors. Dymola supports true RGB color for all graphical objects, offering improved capabilities to design icons.

**Line style** the alternatives are displayed in the figure below to the left.

**Thickness** the alternatives are shown in the middle figure below.

**Arrow style** – please see the figure below to the right.

**Line style, Thickness and Arrow style.**



**Smooth** converts sharp corners to rounded corners.

### Fill style

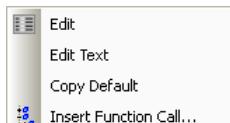


**Fill pattern and Gradient.**



### 4.6.20 Context menu: Variable declaration dialog; Value input field

The variable declaration dialog is described in “Variable declaration dialog” starting on page 264.



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.

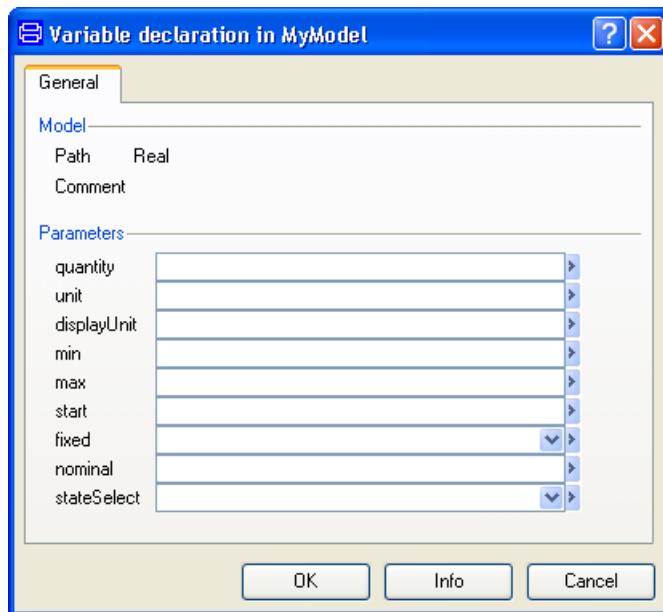
The **Edit** command (or pressing the **Edit** button symbol) will open an editor for the variable value. The editor will look different depending on if the value is just a single value or if it is a matrix of values (in the latter case the number of rows and columns are present in the **Array dimensions** 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.

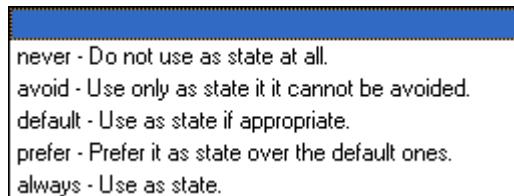
**start** – the start variable of the variable.

**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

**stateSelect** – this gives the preference state selection. (For more information about state selection please see the manual “Dymola User Manual Volume 2”, 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.

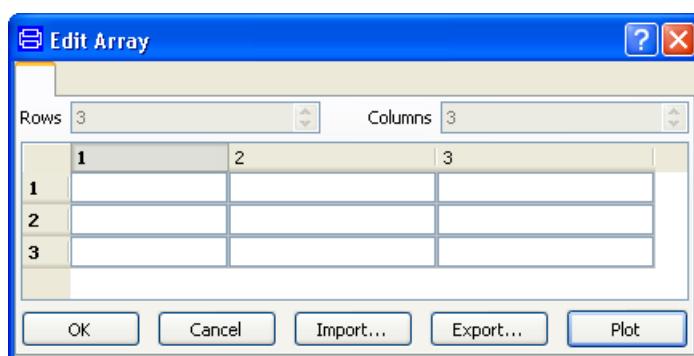
For handling of large matrices (more than 10000 cells) in the matrix editor, see “Handling of large matrices (more than 10000 cells)” on page 361.

For general information about using data from files, see section “Using data from files” starting on page 277.

#### 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 (when the window has been adapted by the user):

**Matrix editor – fixed  
size of matrix.**



Concerning the values, only numbers can be entered – no attributes of the values can be given.

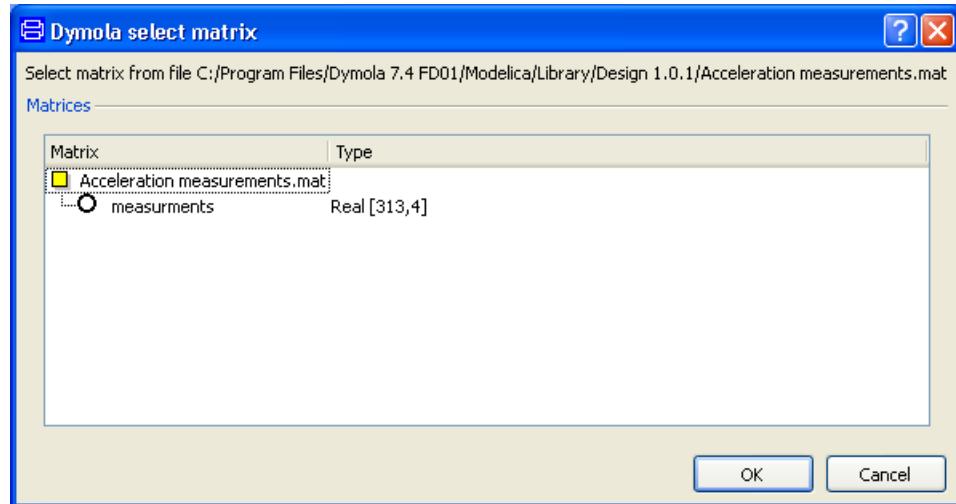
A context menu is available by right-clicking in any value field. That menu is described in the next section (where more alternatives can be used).

Data for the table can be loaded using the **Import...** button. The data can be imported from external Matlab 4 (.mat files), 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:

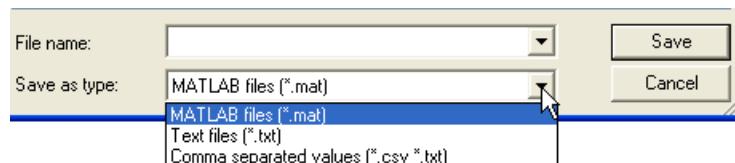
#### Matrix selector.



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....** (A matrix name has also to be specified if saving in .mat format.)

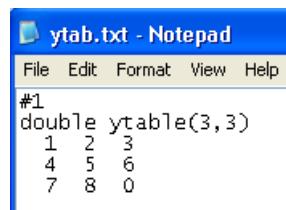
#### Menu for selection of file format when saving.



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 **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:

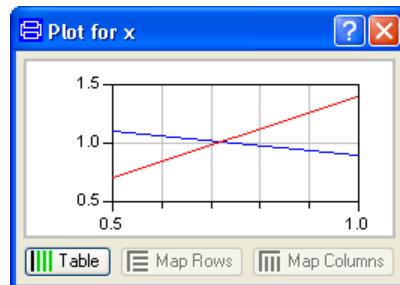
**Example of the specific Text files format.**



```
#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.

**Matrix editor plot.**



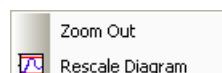
The contents of the table can be interpreted in three different ways when plotted. The type of plot is selected by pressing the corresponding button. Maps are available if the matrix size is greater than 2x2.

**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.

A context menu is available by right-clicking in the window:

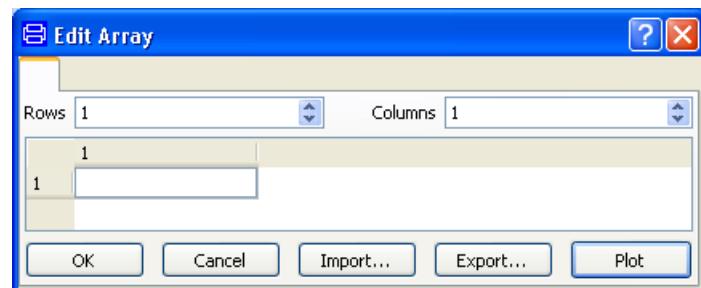


For information, please see section “Context menu: Parameter dialog; plot window” on page 362.

### Variable size of the matrix

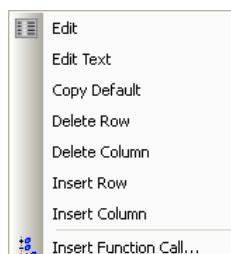
If the case the matrix is of variable size (e.g. a parameter declared `[:, :]`), the edit menu will look like the following when entered the first time:

**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.

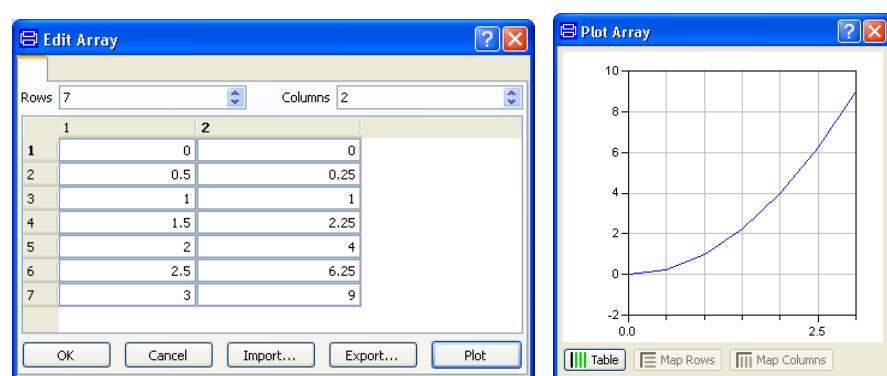
**Insert Row** and **Insert Column** will insert a row/column before the row/column where the cursor is located when displaying the context menu.

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:

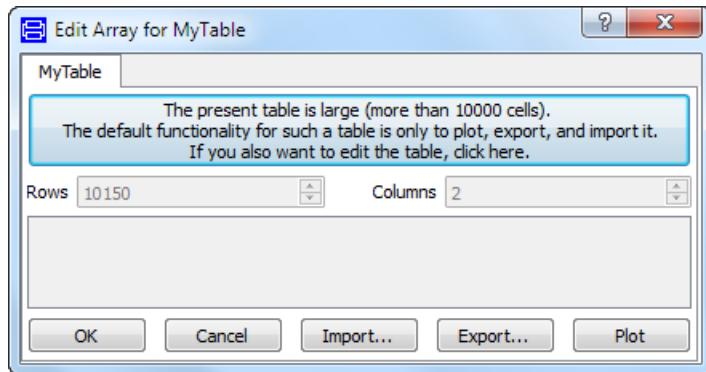
**Example of matrix and corresponding plot.**



### Handling of large matrices (more than 10000 cells)

If a matrix is large (more than 10000 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.

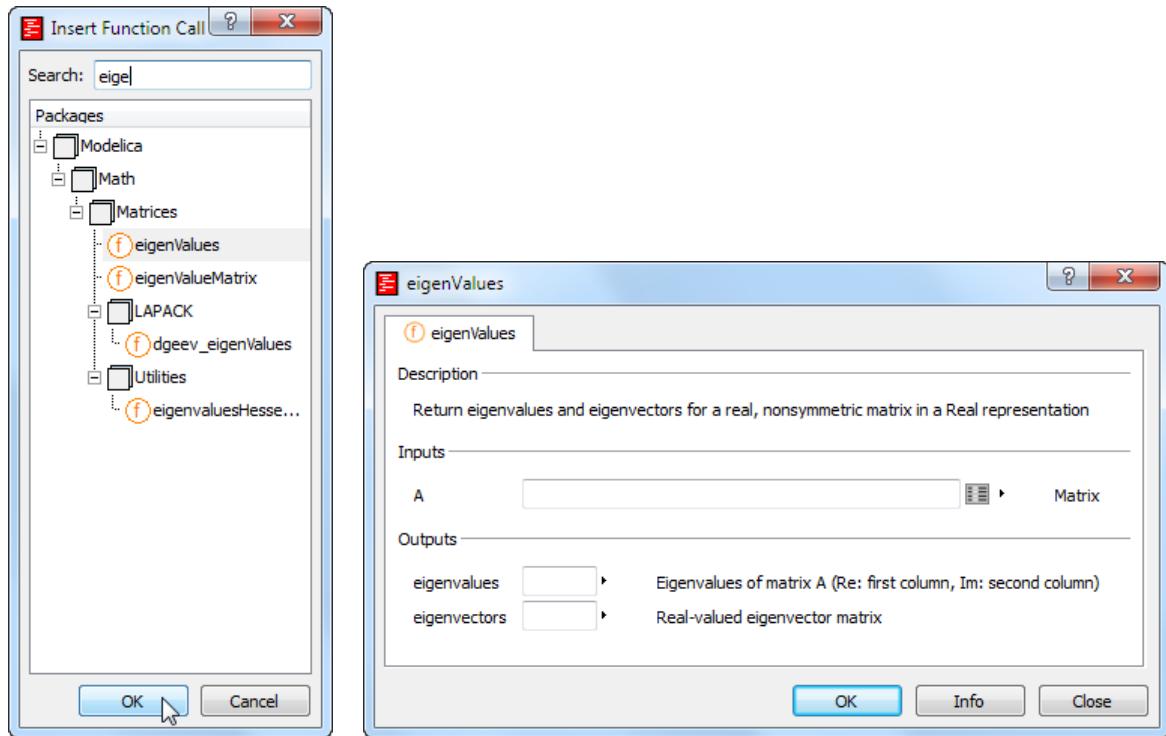


### Edit Text, Copy Default and Insert Function Call

**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.

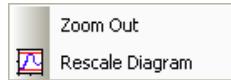
**Insert Function Call...** enables searching (or browsing) for a function and then entering the arguments, allowing easy insertion of a function call. As an example, entering `eige` in the search input field (in order to find `Modelica.Math.Matrices.eigenValues`) will give the following result (followed 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 210.

## 4.6.21 Context menu: Parameter dialog; plot window

This context menu is a sub-menu of the context menu in the previous section.



A context menu is available by right-clicking in the plot window in the parameter dialog:

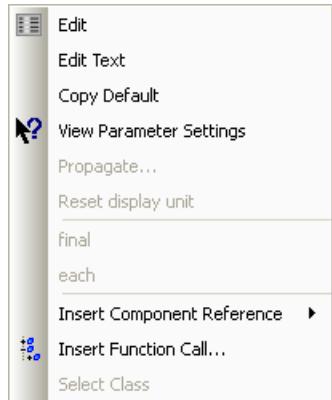
By using **Zoom out** the diagram is zoomed out if earlier been zoomed in into by dragging a rectangle.

**Rescale** will reset the curve to the default size.

## 4.6.22 Context menu: Parameter dialog; parameter input field

The parameter dialog is introduced in “Components and connectors” starting on page 168.

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.

**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 355.

**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.

Table Editor for setup.tuners					
name	Value	active	min	max	
1	0	true	-1E100	1E100	
2	0	true	-1E100	1E100	
3	0	true	-1E100	1E100	

**View Parameter Settings** will show the parameter settings, e.g. the following window:

**View parameter settings.**

Parameter settings.

The values higher up take precedence and the **actual value** is in bold.

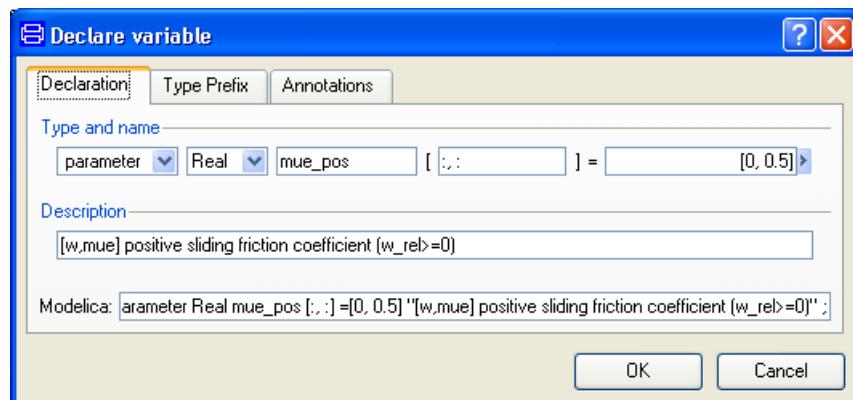
Modified in	Modified to
/input field	
mue_pos	<b>[0, 0.5]</b>

*Advanced users:*

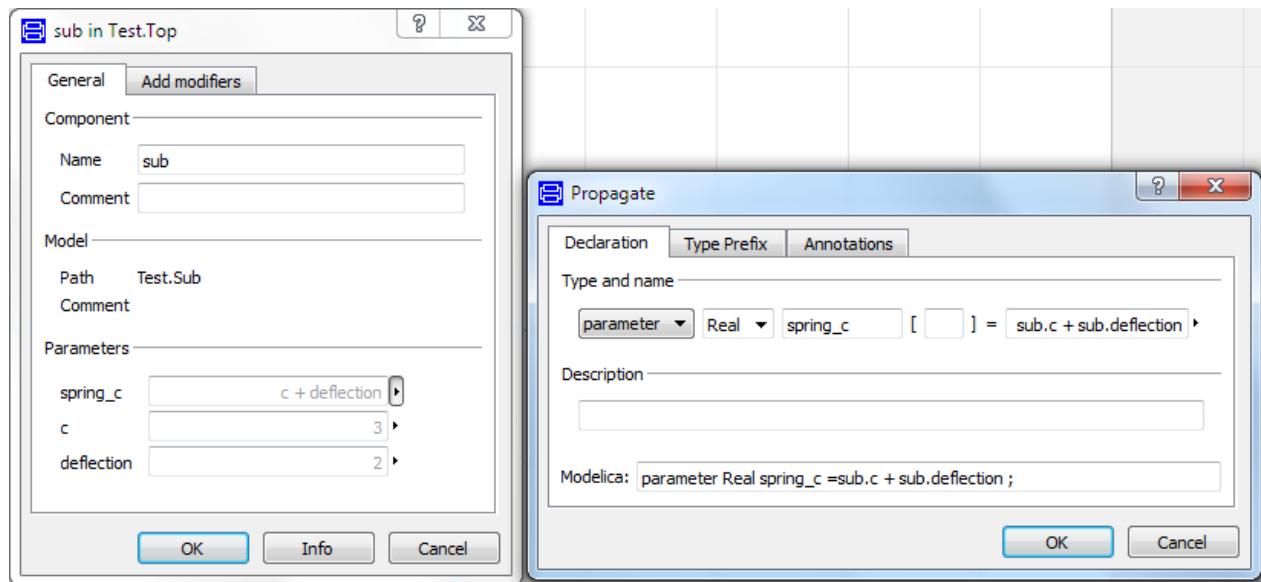
If the modifier is more complex than a normal parameter setting you can give the complete modifier line (including the variable name).

The window shows where the default originates from, and all modifiers applied between the original declaration and the current one.

**Propagate...** 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.



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 264.

**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. 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.

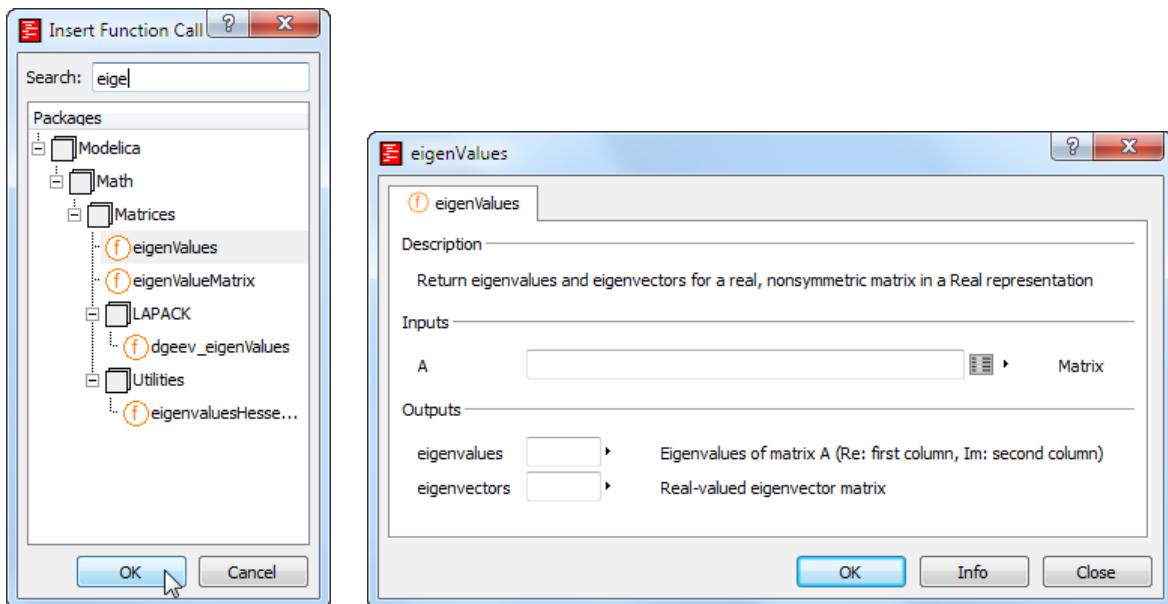
**Insert Component Reference** inserts a reference to a component in the model at the insertion point of the input field.

**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 210.

# **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 in Simulation mode**” starting on page 370 the windows available in Dymola in Simulation mode are described as an overview. More information about the use of some windows is given in the following sections.

Section 5.2 “**Model simulation**” starting at page 389 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. (See also 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 in simulation mode for browsing signals, plotting, animation, scripting and documentation. The section is concluded by a section about simulation settings.

Section 5.3 “**Editor command reference – Simulation mode**” starting on page 462 is a reference section, describing the menus available in Modeling mode. The menus are ordered 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.

Section 5.4 “**Dynamic Model Simulator**” on page 520 describes Dymosim, the executable generated by Dymola.

Section 5.5 “**Scripting**” starting at page 529 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.

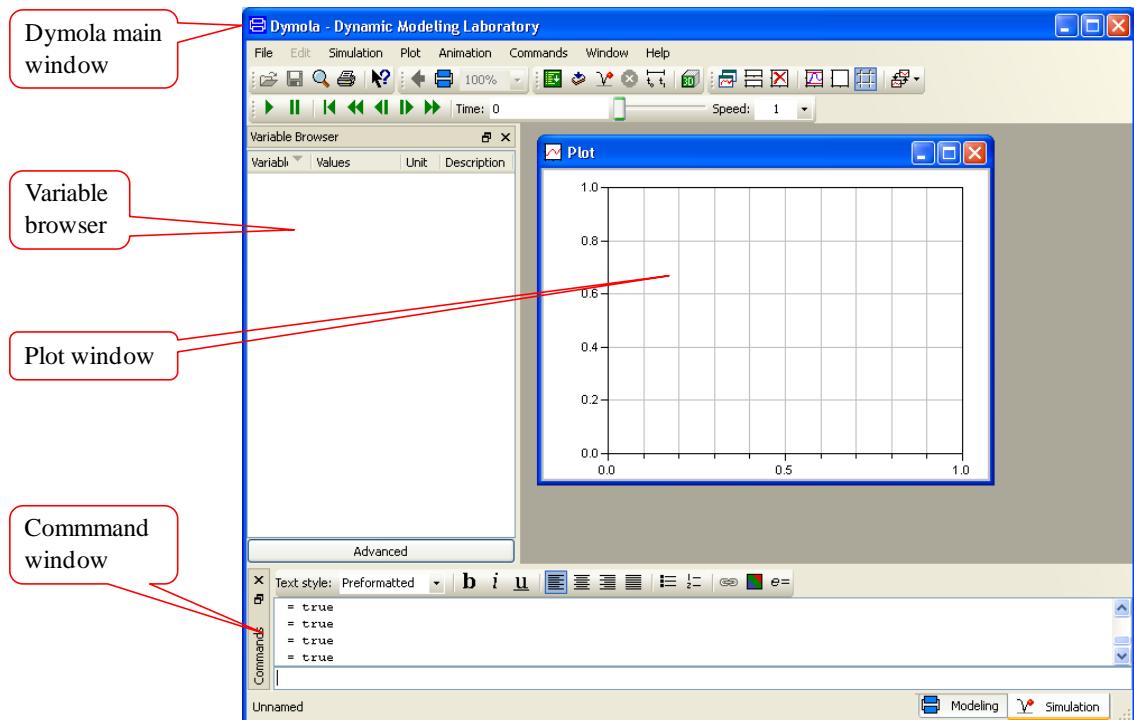
Section 5.6 “**Debugging models**” starting on page 583 lists some important tips for avoiding errors, as well as a number of useful tools how to find errors in the models.

Section 5.7 “**Improving simulation efficiency**” starting on page 604 describes methods to improve simulation efficiency (dealing with e.g. chattering, profiling and inline integration).

Section 5.8 “**Handling initialization and start values**” starting on page 617 describes how to handle the **Continue** command, over-specified initialization problems and discriminating start values.

## 5.1 Windows in Simulation mode

A number of window types are available in Dymola in Simulation mode. If Dymola is started and put in Simulation mode without opening any model it will look like:



What is seen is the Dymola main window which contains a number of sub-windows. The sub-windows available depend partly on what mode Dymola is in – Modeling or Simulation.

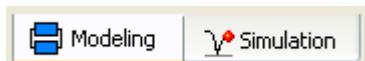
## 5.1.1 Dymola Main window

### Default content in Dymola Main window in Simulation mode

What sub-windows that are shown when entering Simulation mode 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.

Dymola Main window contains by default the following in Modeling mode (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 – Simulation mode” starting on page 462.
- Three sub-windows – one variable browser, one plot window and one command window. They are presented below.
- Two window mode tabs (to the lower right) that is used to decide whether Dymola should be in **Modeling** mode or **Simulation** mode. These tabs are available in both modes.



(Alternatively the keyboard shortcuts **Ctrl+F1** and **Ctrl+F2** or the **Window > Mode** menu can be used. Please see previous chapter, section “Editor command reference”, section “Main window: Window menu”).)

### Additional content available

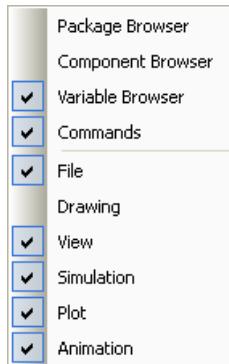
Additional content available depends on:

- The model being simulated; if it contains animation or visualization, animation or visualization windows will appear when the model is being simulated.
- User selection of content; by using the command **Window > Tools** the content of Dymola Main window can be changed. Features by default only available in Modeling mode can be added, and what toolbars are shown can be selected. (This menu is also available using the context menu of the main window, see next section for additional information.)
- User selection of commands; the diagram layer window is automatically opened (if not present) if demanded by a command (e.g. **Show Component** in the plot window context menu or a **Display** command in the variable browser context menu). Also a message window can be displayed using the command **Simulation > Show Log**. Finally, if the user selects a command for creating or opening a script, the script editor is displayed.
- Progress of translation/simulation. If errors are detected a message window is displayed.
- When using any **Info** button or **Info** command an information browser appears.

## Context menu

Right-clicking in an empty space in the commands/tool bar section of the main window will display a menu where the content of the main window (available sub-windows and toolbars) can be changed.

**Context window of  
main window in  
Simulation mode.**



This works also by right-clicking in an empty space in the gray header of any sub-window. For more information about this command, please see chapter “Developing a model”, section “Window > Tools”.

## 5.1.2 Variable browser

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.

What variables that 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.

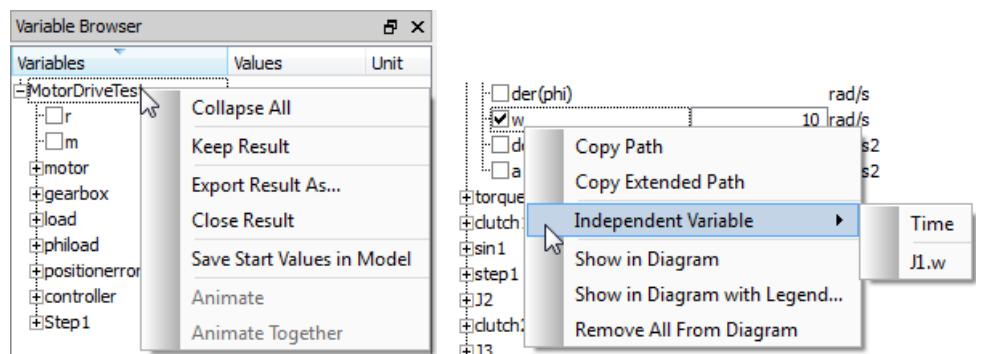
Modified initial conditions and parameter values entered in the variable browser in Simulation mode can be saved to the model.

Please see the section “Variable browser interaction” on page 392 for more information.

**The variable browser (expanded to show the description).**

Variables	Values	Unit	Description
CoupledClutches 1			
freqHz	0.2	Hz	frequency of sine function to invoke clutch1
T2	0.4	s	time when clutch2 is invoked
T3	0.9	s	time when clutch3 is invoked
J1			
J	1	kg.m <sup>2</sup>	Moment of inertia
initType			Type of initialization (defines usage of start values below)
phi_start	0	rad	Initial or guess value of rotor rotation angle phi
w_start	10	rad/s	Initial or guess value of angular velocity w = der(phi)
a_start	0	rad/s <sup>2</sup>	Initial value of angular acceleration a = der(w)
stateSelection			Priority to use phi and w as states
phi		rad	Absolute rotation angle of component (= flange_a.phi = flange_b.phi)
der(phi)		rad/s	der(Absolute rotation angle of component (= flange_a.phi = flange_b.phi))
flange_a			
flange_b			
w		rad/s	Absolute angular velocity of component
der(w)		rad/s <sup>2</sup>	der(Absolute angular velocity of component)
a		rad/s <sup>2</sup>	Absolute angular acceleration of component
torque			
clutch1			
sin1			
Advanced			

By right-clicking context menus can be used. The context menus are different depending on where the cursor is 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 504 and 506, respectively, for more information.



### 5.1.3 Diagram layer window

A diagram window can be opened by the user or automatically by any command demanding it.

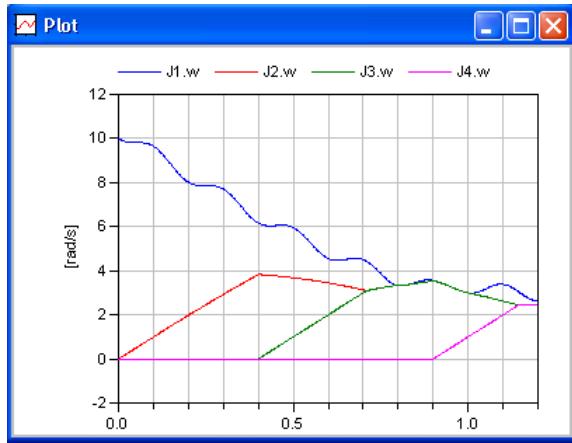
It can be used for selecting what should be visible in the variable browser and/or for presenting and changing values. For more information, please see section “Using the diagram layer to select what should be visible in the variable browser” on page 392 or section “Using the diagram layer” starting on page 401, respectively.

## 5.1.4 Plot window

Dymola supports plotting of any variable. In Simulation mode, multiple plot windows may be created using the command **Plot > New Plot window**. 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.**



The user can interact in the plot window, e. g.:

- Display several diagrams and work with them in the same plot window.
- Display Boolean and enumeration signal.
- 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.
- Change time unit on x-axis.
- Change the display unit of a signal.
- Display signal operators.
- Plot general expressions.
- Select multiple curves in a diagram (for e.g. copying the curve values to Excel).
- Display a table instead of curves.
- Create tables from curves.

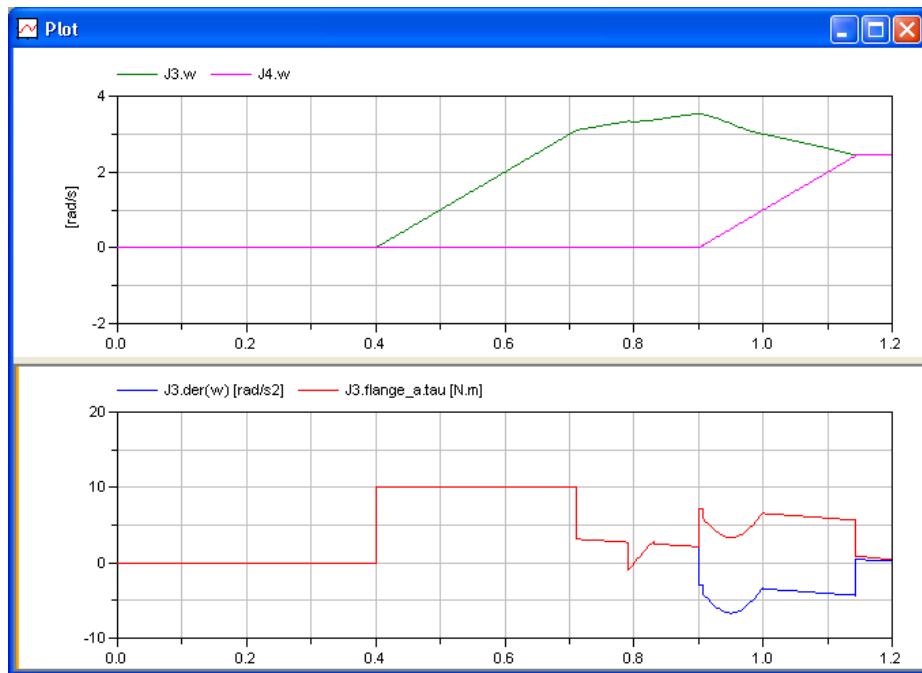
- Plot parametric curves.
- Change color, line/marker style and thickness of a signal.
- Set diagram heading and axis titles and ranges.
- Insert and edit text objects in the plot.
- Change the appearance, layout and location of the legend. (The legend can be located in several different places, for example inside the diagram.)
- Display the component where the signal comes from in a diagram layer.
- Display a time line in plots for an animation
- 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 405 for more information on interaction in the plot window.

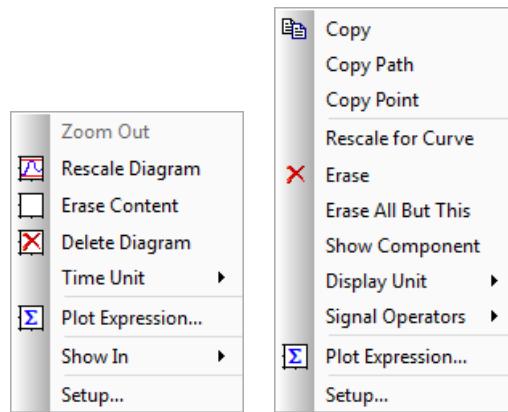
For an overview of where to find information in the manual concerning plot functionality, please see “Plotting” on page 404.

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.**



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.

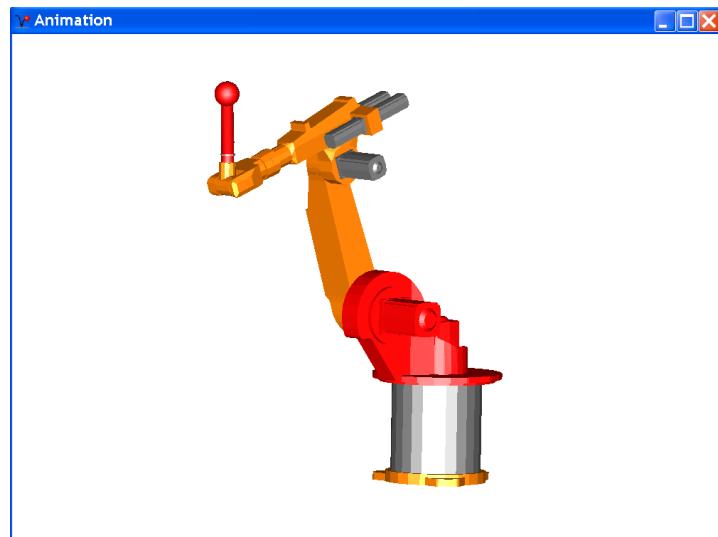


These context menus are described in the sections “Context menu: Plot window” on page 507 and “Context menu: Plot window – curve and legend” on page 508. Other context menus related to plot features are described after them (e.g. the context menu for a text object in a plot, context menus for signal operators and parametric curves, context menu for table window).

### 5.1.5 Animation window

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.

#### The animation window.



## 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

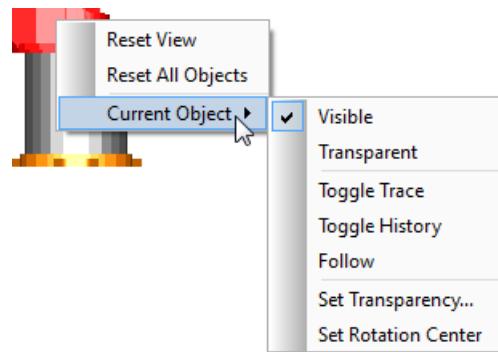
Parameters such as size can be specified. Coordinate systems can be defined by a complete 3-dimensional transformation (3x3-matrix+translation). The information can either be parametric or depend on the dynamics of the model.

For more information about the graphical objects, please see section “Defining Graphical Objects for the animation window” starting on page 442.

### Interaction

Please see the section “Animation window interaction” starting on page 444.

A context menu is available in the window:

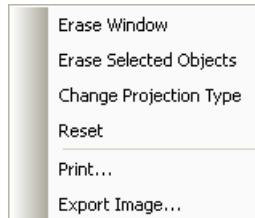


For more information about this menu, please see section “Context menu: Animation window” on page 511 for more information.

## 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 manual “Dymola User Manual Volume 2”, chapter “Visualize 3D”.

A context menu is available in the visualizer window:

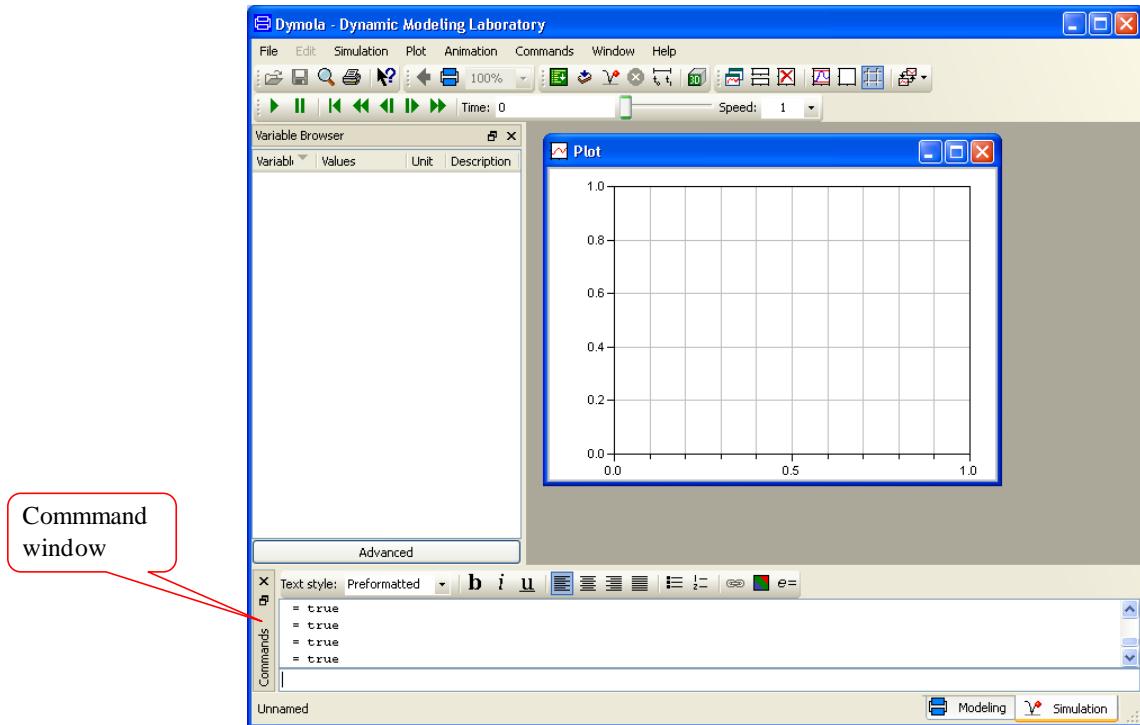


For more information about this, please see the section “Context menu: Visualizer window” on page 512.

## 5.1.7 Command window

### General

When entering **Simulation** mode the command window is shown by default:



The command window is important for scripting, documenting simulations and simulation of models.

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.



The command window can be displayed also in Modeling mode, and as a separate window as well. Please see section “Displaying the command window” starting on page 447 for more information about this.

### The command log pane

The command log pane displays the command log, containing commands given and the result of these commands, plus additional content added by the user.

Basically the user can work with the command log pane in two modes, the default mode of a documentation editor or a simpler text editor activated by a flag.

The documentation editor (and options that can be added in this mode) 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.

The basic text editor reflects looking at the command log primarily as a list of commands and the result of these. (The commands can be activated by command buttons, scripts or by typing in command in the command input line.) What can be displayed in the basic text editor is described in the section “The command log pane in textual mode” starting on page 449.

A valuable advantage is that even if the documentation editor is used, 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 options for saving the command log (see next section).

More information about the documentation editor is given in section “Documentation” on page 452.

### 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.

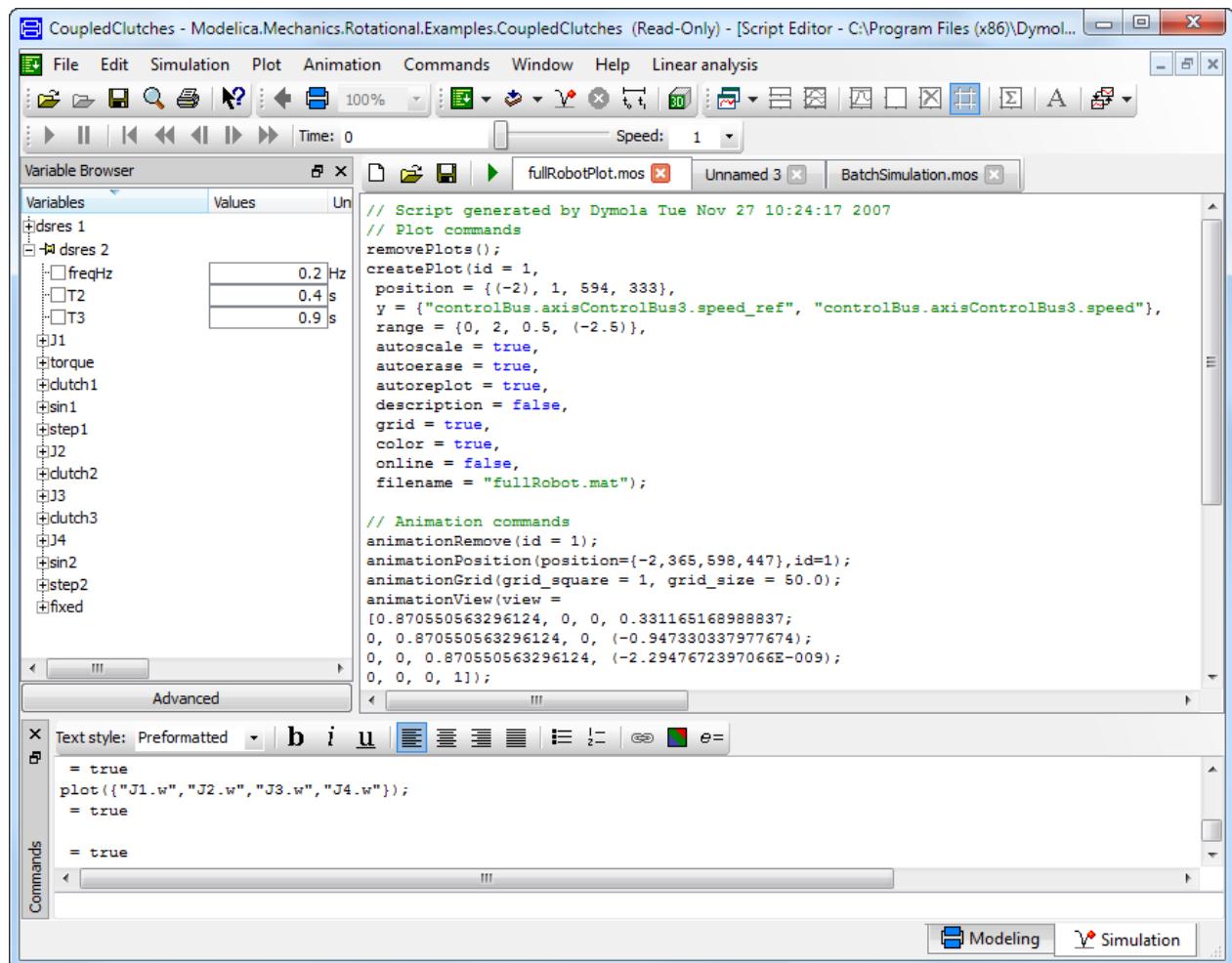
The reason that the saved command log file can have different content is that the content depends on what file format the file is saved in. For more information, see section “The command log file” starting on page 449.

### **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 450.

### **5.1.8      Script editor**

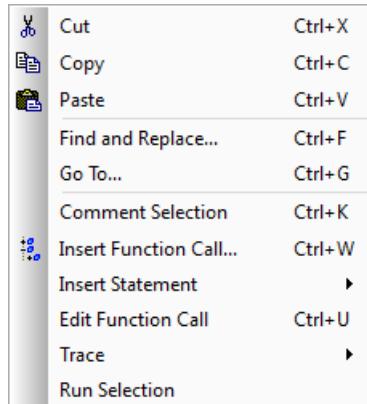
The Dymola script editor is displayed when the user wants to create a new script or open an existing script by the commands **Simulation > Commands > New Script**, or **Simulation > Commands > Open Script...** or the corresponding alternatives to the **Run Script** button.



Several scripts can be handled in the same 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 549 for more information.

By right-clicking, a context menu is available:



Please see section “Context menu: Script editor” on page 516 for more information.

### 5.1.9 Message window

The message window is not displayed by default, it will either pop up if needed (e.g. when errors are detected) or when the user explicitly asks for it (by e.g. the command **Simulation > Show Log**)

The message 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

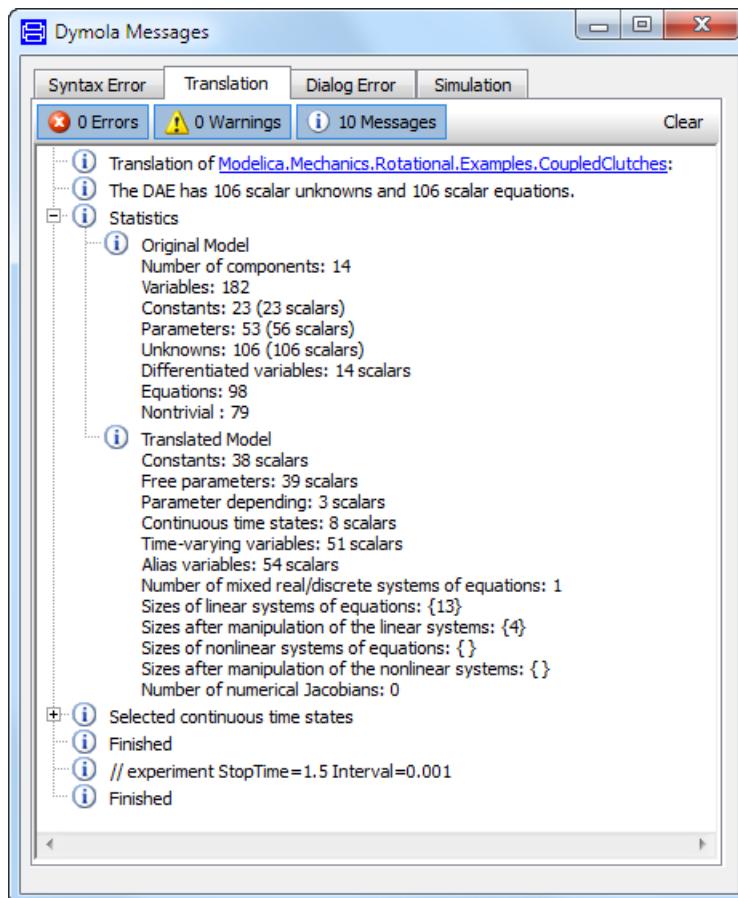
The **Translation** tab shows information and statistics related to translation and symbolic manipulation of a model. The translation tab can be very useful when debugging and improving models, for examples to find iteration variables missing start values and to see if there is any dynamic state selection.



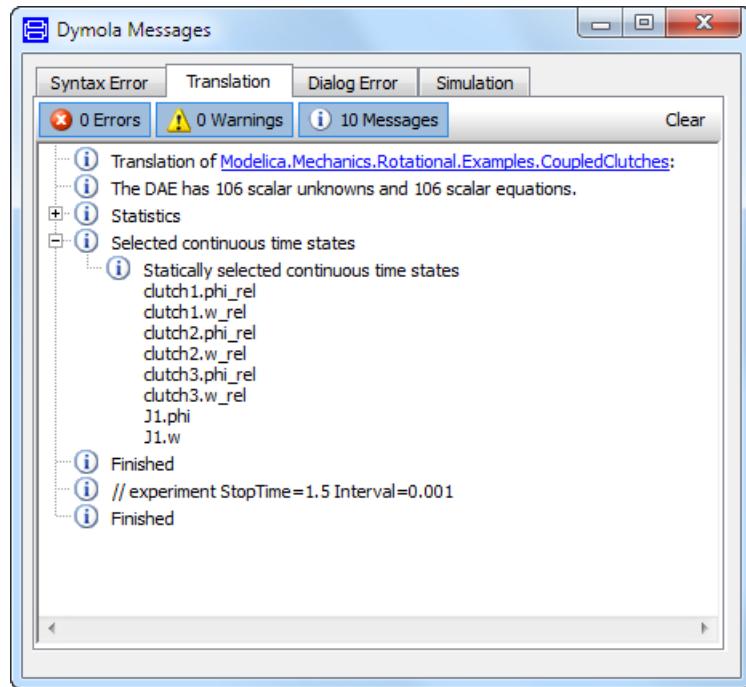
What information is displayed depends on the settings in the **Translation** tab in the simulation setup. The simulation setup is reached by clicking the **Setup** button or by the **Simulation > Setup...** command. Please see the section “Translation tab” on page 473 for more information.

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 439.

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;
```

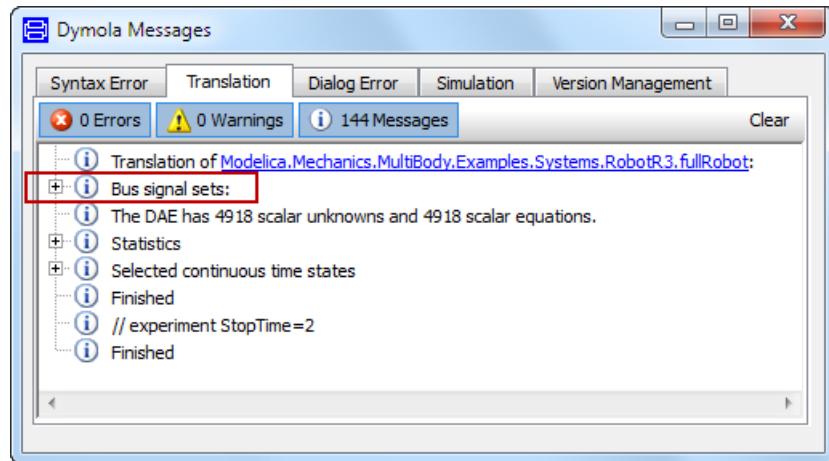
By setting this flag to `false`, vectors and matrices are listed if all their elements are states.

### Bus signal tracing

From the Translation 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:

```
+ [i] Bus-signal: speed_ref
Source: connect\(pathPlanning.pathToAxis1.qd\_axisUsed.y, pathPlanning.pathToAxis1.axisControlBus.speed\_ref\);
Use: connect\(axis1.controller.axisControlBus.speed\_ref, axis1.controller.gain2.u\);
Use: connect\(axis1.axisControlBus.speed\_ref, axis1.initializeFlange.w\_start\);
```

**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).

```
+ [i] Bus-signal: speed_ref
Source: connect\(pathPlanning.pathToAxis1.qd\_axisUsed.y, pathPlanning.pathToAxis1.axisControlBus.speed\_ref\);
Use: connect\(axis1.controller.axisControlBus.speed\_ref, axis1.controller.gain2.u\);
Use: connect\(axis1.axisControlBus.speed\_ref, axis1.initializeFlange.w\_start\);
[i] Connected bus variables:
output pathPlanning.pathToAxis1.axisControlBus.speed\_ref "Reference speed of axis flange"
pathPlanning.controlBus.axisControlBus1.speed\_ref
axis1.motor.axisControlBus.speed\_ref
controlBus.axisControlBus1.speed\_ref
input axis1.controller.axisControlBus.speed\_ref
input axis1.axisControlBus.speed\_ref
```

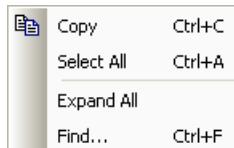
Note:

- **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 is incomplete (except that if 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.

### Context menu

By right-clicking in the tab, a context menu is displayed.



Please see section “Context menu: Message window” on page 519 for more information.

The content in the tab can be erased by clicking **Clear** in the header.

### Dialog Error tab

The dialog error tab shows error messages generated from errors in interactive functions. Consider the following example of model that is syntactically correct but contains another error.

```
function foo
    input Integer a;
    output Real vec[a];
algorithm
    for i in 1:a+1 loop
        vec[i] := 2*a;
    end for;
end foo;
```

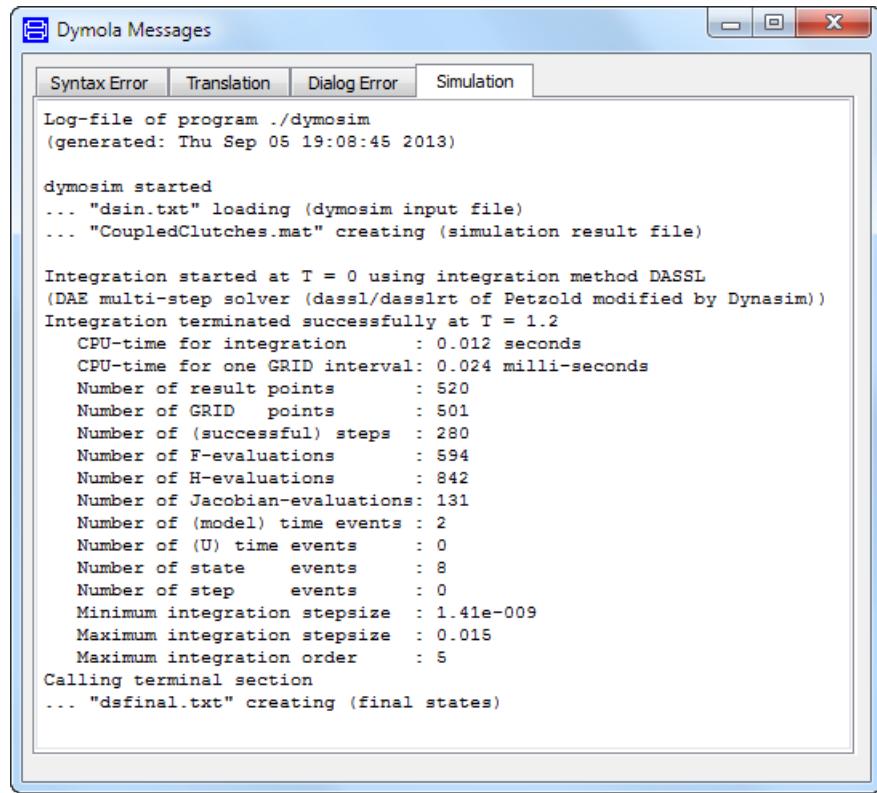
The error in this model is not detected by the **Check** command. However trying to execute the function by right-clicking, selecting **Call Function...** and pressing **Execute**, an error is displayed in the Dialog Error tab (`Assignment failed equation vec[i] = 2*a;`). The error here is that we try to assign the array vec at index `a+1` even though it is defined to only have `a` elements. To correct the function change the declaration of vec to be `output Real vec[a+1];`.

### Simulation tab

The simulation tab displays information regarding the simulation of a model. The layout and available information may differ from the example image below depending on the selected integration routine.

From the simulation tab we can see, for example, start and stop time, CPU time for integration and the number of result points.

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.



The screenshot shows a Windows application window titled "Dymola Messages". The window has a title bar with standard window controls (minimize, maximize, close) and a menu bar with tabs: "Syntax Error", "Translation", "Dialog Error", and "Simulation". The "Simulation" tab is selected. The main area of the window contains a text log of a simulation run:

```
Log-file of program ./dymosim
(generated: Thu Sep 05 19:08:45 2013)

dymosim started
... "dsin.txt" loading (dymosim input file)
... "CoupledClutches.mat" creating (simulation result file)

Integration started at T = 0 using integration method DASSL
(DAE multi-step solver (dassl/dasslrt of Petzold modified by Dynasim))
Integration terminated successfully at T = 1.2
    CPU-time for integration      : 0.012 seconds
    CPU-time for one GRID interval: 0.024 milli-seconds
    Number of result points       : 520
    Number of GRID points         : 501
    Number of (successful) steps  : 280
    Number of F-evaluations       : 594
    Number of H-evaluations       : 842
    Number of Jacobian-evaluations: 131
    Number of (model) time events : 2
    Number of (U) time events    : 0
    Number of state events        : 8
    Number of step events         : 0
    Minimum integration stepsize : 1.41e-009
    Maximum integration stepsize : 0.015
    Maximum integration order    : 5
Calling terminal section
... "dsfinal.txt" creating (final states)
```

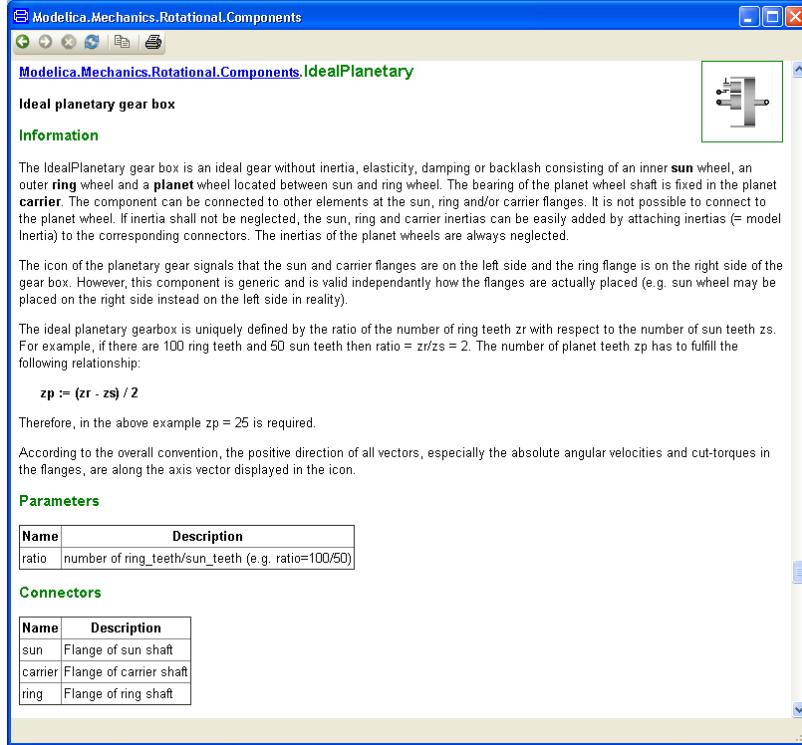
## Version Management tab

The version management 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" in the manual "Dymola User Manual – Volume 2" 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.

## 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 529.

#### Selecting model

The model to be used in a simulation experiment is selected in Modeling mode. The package browser is used to select a model as the active model. The title of the Dymola window is set to the name of the active model. To enter the Simulation mode click on the **Simulation** tab in the bottom right corner of the Dymola window.

#### Translation



To prepare a model for simulation, it needs to be translated. The translation is initiated by pressing the **Translate** button in the toolbar. If the translation fails, a message window is displayed. For more information on this, please see "Errors and warnings when translating" on page 439.

#### 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.**

Variables	Values	Unit	Description
CoupledClutches 1			
freqHz	0.2	Hz	frequency of sine function to invoke clutch1
T2	0.4	s	time when clutch2 is invoked
T3	0.9	s	time when clutch3 is invoked
J1			
J	1	kg.m <sup>2</sup>	Moment of inertia
initType			Type of initialization (defines usage of start values below)
phi_start	0	rad	Initial or guess value of rotor rotation angle phi
w_start	10	rad/s	Initial or guess value of angular velocity w = der(phi)
a_start	0	rad/s <sup>2</sup>	Initial value of angular acceleration a = der(w)
stateSelection			Priority to use phi and w as states
phi		rad	Absolute rotation angle of component (= flange_a.phi = flange_b.phi)
der(phi)		rad/s	der(Absolute rotation angle of component (= flange_a.phi = flange_b.phi))
flange_a			
flange_b			
w		rad/s	Absolute angular velocity of component
der(w)		rad/s/s	der(Absolute angular velocity of component)
a		rad/s <sup>2</sup>	Absolute angular acceleration of component
torque			
clutch1			
sin1			
Advanced			

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 in Simulation mode can be saved to the model.

For more information about the variable browser, see “Variable browser interaction” starting on page 392.

## 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 460.)

## Perform simulation



To run the simulation, click on the **Simulate** button.

## 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 397.

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.

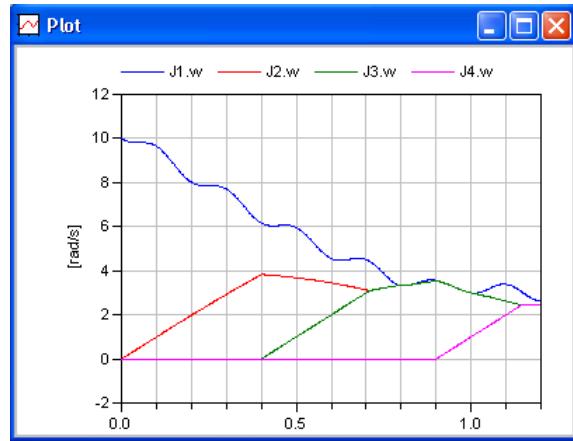
Variables to be plotted are selected by clicking in the variable browser. When a variable is selected the square in front of the name is ticked. The variable browser above has J1.w and J2.w selected for plotting.

The user can interact in the plot window using zooming, tooltips etc. Please see section “Plot window interaction” starting on page 405 for more information.

For an overview of where to find information in the manual concerning plot functionality, please see “Plotting” starting on page 404.

Please note that the diagram layer of the edit window also can be shown in Simulation mode. That diagram can be used to decide what variables should be shown in the variable browser. Please see section “Using the diagram layer to select what should be visible in the variable browser” on page 392 for more information.

**A plot window.**



### Exporting results

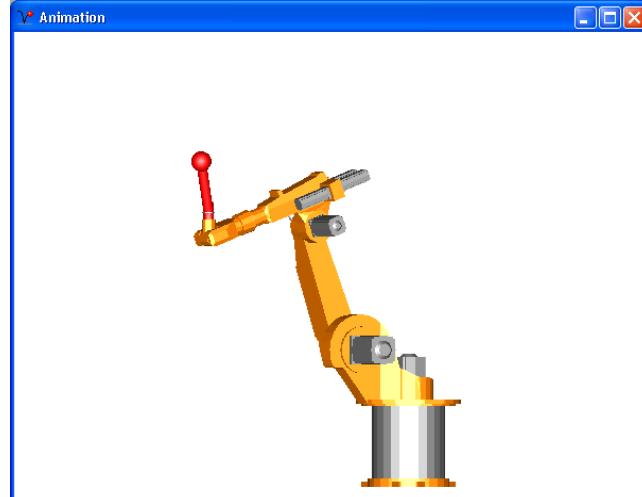
It is possible to export plotted variables in files of different formats (CSV, Matlab and text format). Please see section “Context menu: Variable browser – nodes” on page 504 for more information.

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 “Defining Graphical Objects for the animation window” on page 442 and section “Animation window interaction” on page 444.

**An animation window**



## Documenting the simulation

The simulation can be documented using the command window. A documentation editor is available in the command log pane of the command window by default. 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 “Documentation” starting on page 452 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 529.

## 5.2.2 Variable browser interaction

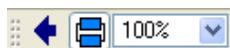
### Displaying plot variables

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 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 475 for more information about this.

### Using the diagram layer to select what should be visible in the variable browser

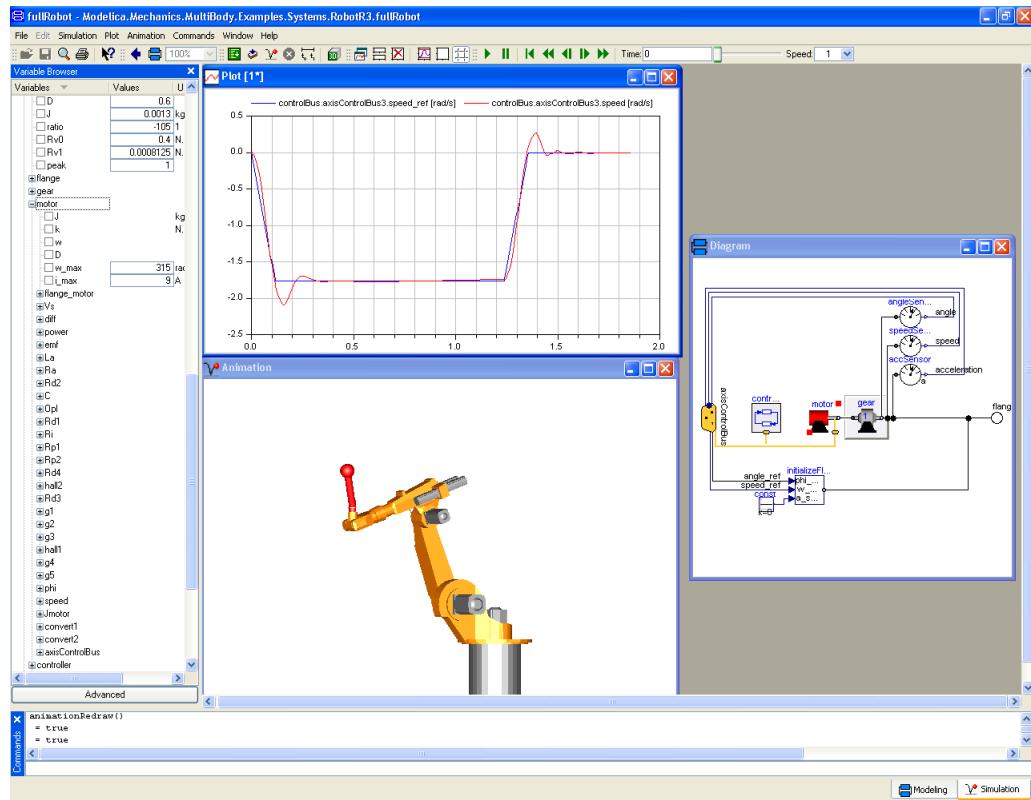
The diagram layer representing the model is available in Simulation mode to enable the user to follow a simulation by displaying variables and to control it by setting parameters. The user can descend into any level of the model in order to plot or display variables.



Push the **Diagram layer** button in the diagram layer toolbar to show the diagram. (The diagram toolbar also contains **Previous** button to go back to previous selection and a zooming tool. For more information on this toolbar, please see section “Main window: Window menu” on page 504.)

To display the sub-components in a component, select it, right-click to display the context menu and select **Show Component**. Use the **Previous** button in the diagram layer toolbar to get back.

The context menu for a component contains the choice **Show Variables**, which will highlight and open the selected component instance in the variable browser.



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.

More information on how to use the diagram layer in simulation mode is presented in section “Using the diagram layer” starting on page 401.

### 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.

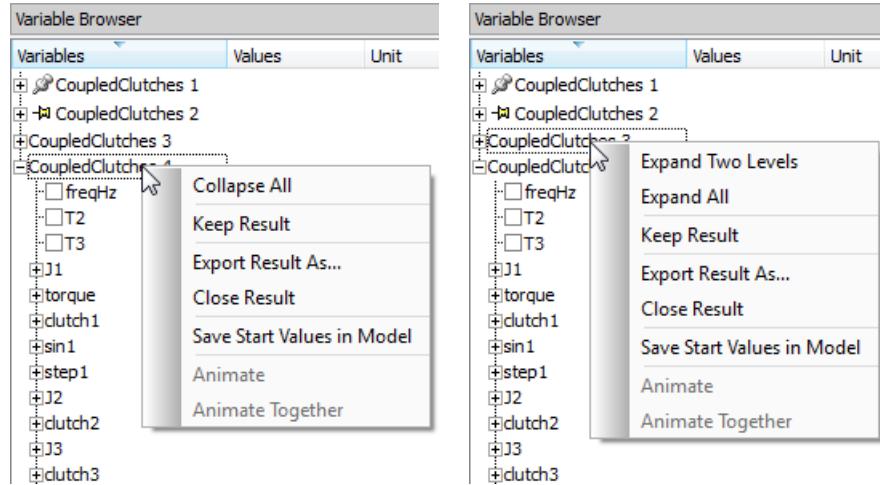
Please note that it is possible to use the diagram layer to decide what variables should be shown in the variable browser. Please see section “Using the diagram layer to select what should be visible in the variable browser” on page 392 for more information.

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 397.

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 504. Please note that if the cursor is beside/resting on a *variable*, another context menu is shown; please see below.

### Opening/Closing of the tree structure

- Clicking on a + in front of a 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.
- 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 – in front of a node will close it.
- 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.
- **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.



The **Erase Content** button erases all curves, and text objects, if any (see “Plot > Erase Content” on page 487).

## 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 in simulation mode. For conditions and interactive parameter tuning, see section “Changing parameter values using the diagram layer” on page 402.

## Save start values in model

Modified initial conditions and parameter values entered in the variable browser in Simulation mode 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 (see images above).

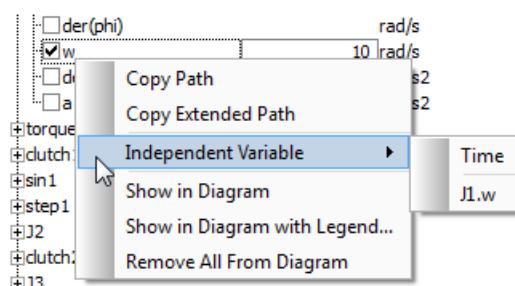
There are some limitations; initial values of the following parameters cannot be saved this way:

- Evaluated parameters.
- Final parameters.
- Protected parameters.
- Parameters in array of components.
- Parameters being part of bounded record components.

## 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, 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.

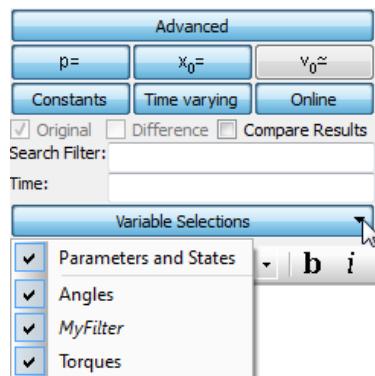


Time is always an alternative to allow going back to plotting trend curves.

## Advanced mode

Pressing the **Advanced** button in the variable browser 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.

### Advanced variable browser mode.



**p=** Set parameters of translated model. You must first translate the model.

**x<sub>0</sub>=** Set initial conditions of translated model. You must first translate the model.

**v<sub>0</sub>~** 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.

**Original** Plot the selected signals.

**Difference** Plot the difference between selected signals. When plotting the difference between two Boolean signals, exclusive OR (XOR) is used to display when the signals differ.

**Compare Results** Use last results as master result file. When selecting a plot variable it also plots the corresponding variables from the other results.

**Search Filter** 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.

**Time** The value column of the variable browser will show the values of variables for this time. Enter time and press return.

**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”.

### 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

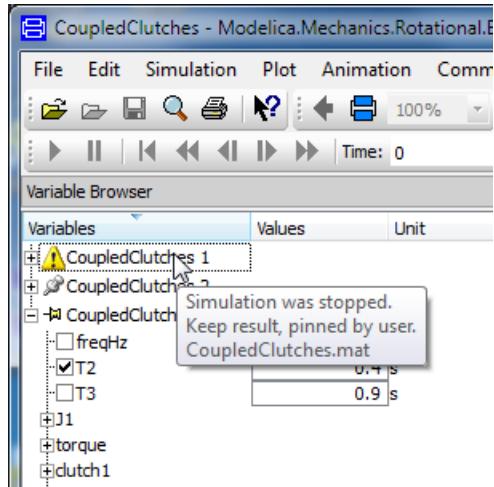
#### Default handling

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 **Options** tab reached by e.g. the command **Plot > Setup**. Please see section “Options tab” on page 496 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, 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 **Plot > Open Result...**, or by dragging or double-clicking such files. Such manually loaded files are not included in the **Number of results to keep setting** above. Please see section “**Plot > Open Result...**” on page 487 for more information about this.

### Closing result files

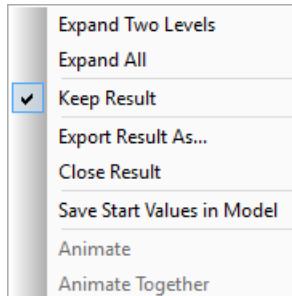
Result files can be closed using the context command **Close Result** in the context menu available for the result files.

### 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 automatically be kept when a new simulation is performed, unless the variable is deselected before the simulation.



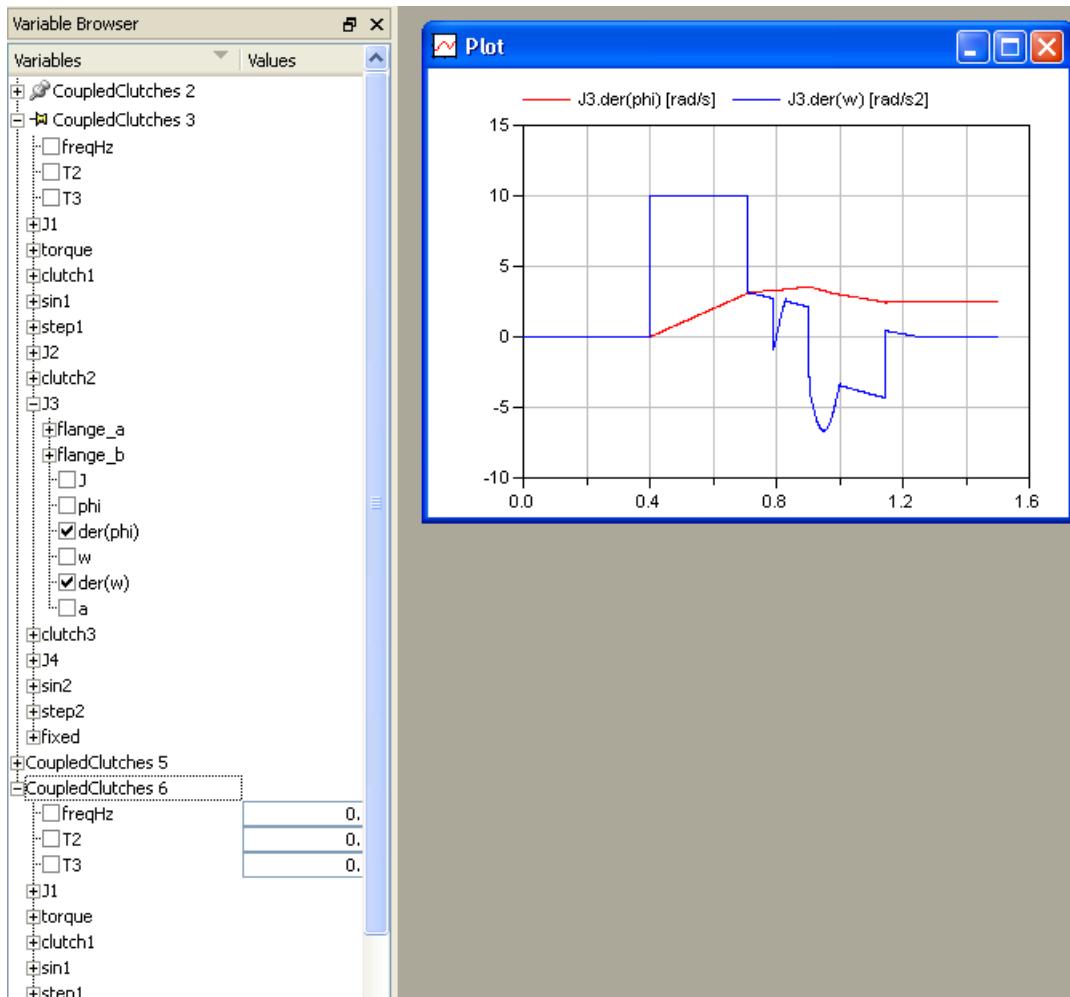
Such result files will be indicated by a specific pin symbol in front of the result file in the variable browser – see also the example below, and compare with the pin symbol for user-defined selection of result files to keep.

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.

#### **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, Matlab and txt) using the command **Export Result As...** in the context menu of the variable browser.

Note that it is possible to export all variables in a result file, but also a selection (plotted variables). What to export is selected when selecting the file format of the export.

Please see section “Context menu: Variable browser” on page 504 for more information.

## 5.2.3 Using the diagram layer

### Moving and zooming in the diagram layer

#### Moving

To move the view, press and 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.

From Dymola 2015 FD01, infinite diagram layer is supported; moving works independently of zoom factor, and scroll bars are not shown.

Previously the zoom factor had to be larger than 100 % for moving to work (otherwise the total diagram layer was shown anyway). When the zoom factor was larger than 100 % also scroll bars appeared that could be used to move the view. This old behavior can be retained setting the flag

```
Advanced.InfiniteDiagramLayer = false
```

The flag is **true** by default.

#### Zooming

To zoom, there are four different options:

- Press and hold the **Alt** key and span a rectangle<sup>3</sup>. 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.

---

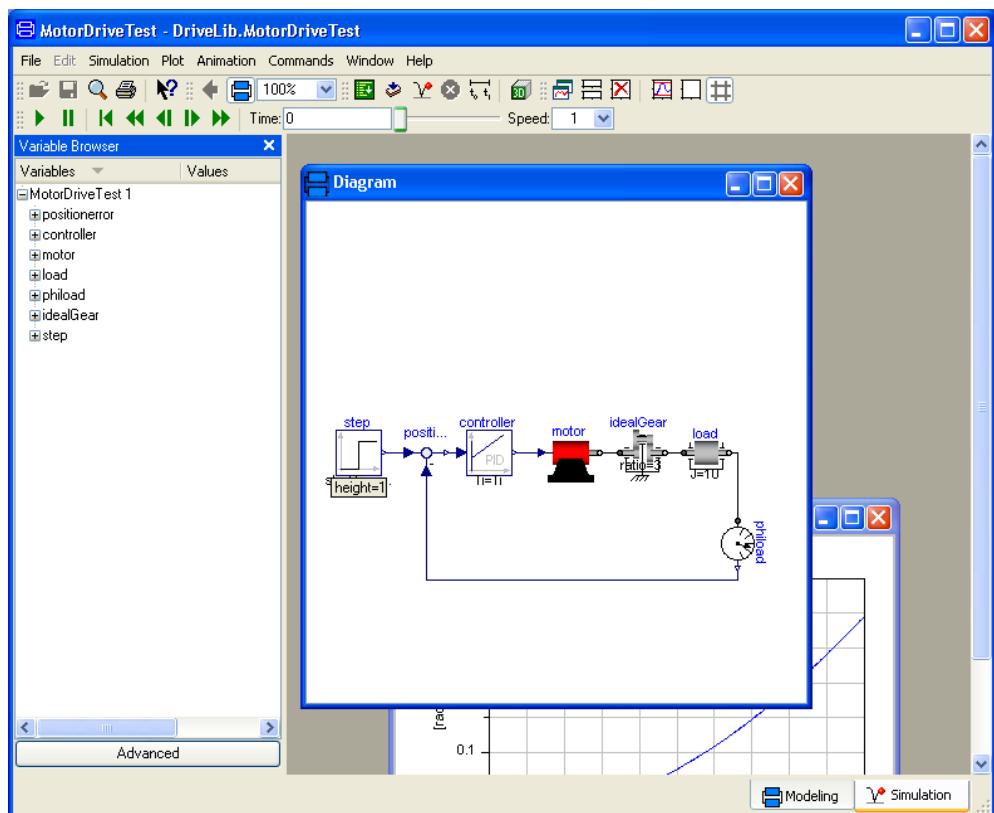
<sup>3</sup> On Linux, **Ctrl+Shift** have to be used instead of **Alt**.

A convenient way of returning to 100% of zooming is to right-click an empty area, having nothing selected, and select **Fit to Window** from the context menu.

## 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.

### Presenting values in the diagram layer.

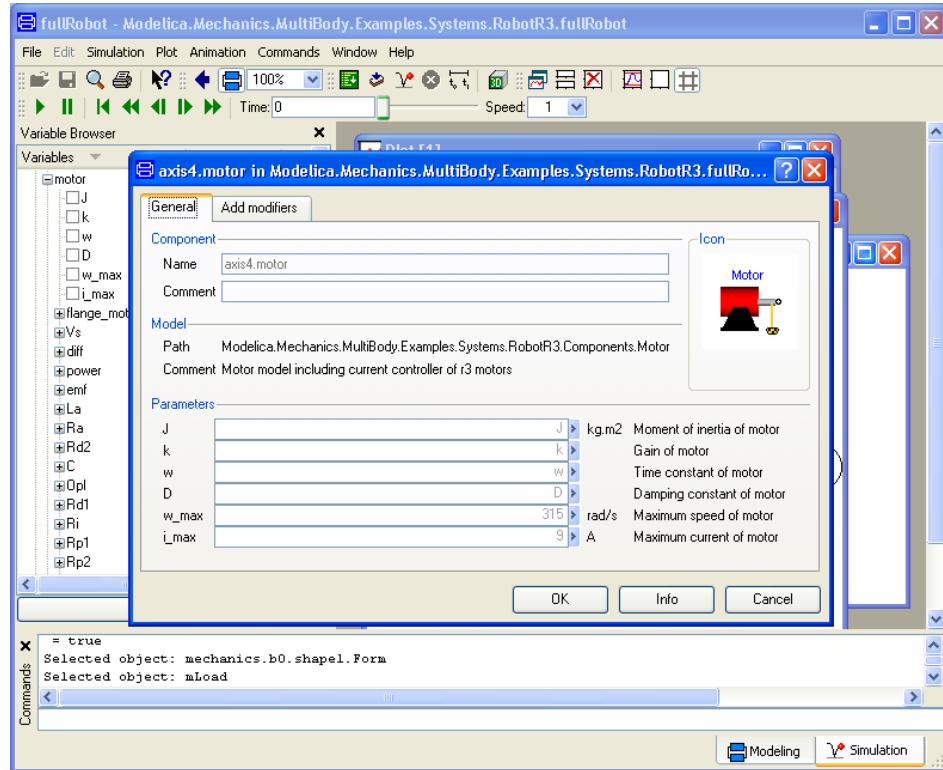


The context menu entry **Show in Diagram with Legend...** makes it possible to change the representation of the value display. For more information, please see section “Context menu: Variable browser – signals” on page 506.

## Changing parameter values using the diagram layer

By double-clicking on a component or selecting **Parameters...** in the context menu the parameter dialog is displayed. 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 from the parameter dialog of the diagram view in Simulation mode. (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.

## 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 390 and in the section “Plot window” on page 374.

### Selecting signals

The variable browser can be used for selecting what signals to plot. Please see section “Variable browser interaction” starting on page 392 for options here. Please note the use of the Advanced mode to be able to plot differences between selected signals etc.

### Plotting from large result files

For performance reasons, there is a limit how many points that can be plotted in a curve. This limit is specified by the flag `Advanced.Plot.PerformanceLimit`. The default value is 100 000. If more points are to be plotted, this value can be changed. Note that this will increase the time for plotting.

### Plot window interaction

This is described in the next section “Plot window interaction” starting on page 405. For an overview, please see the first sub-section of that section.

### Saving a png image of the active plot window

The command **File > Export... > Image...** can be used to save a png or svg image of the active plot window. Please see section “Main window: File menu” on page 462. 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, Matlab and text format). Please see section “Context menu: Variable browser – nodes” on page 504 for more information.

### Storing global plot settings between sessions

It is possible to store global plot settings, as well as individual plot settings, between sessions by ticking **Plot layout and setup** in the **Save settings** tab that is available using

the command **Edit > Options....**. For more information about this command, please see the corresponding description in previous chapter.

### Saving the settings of a plot window

The settings of a plot window can be saved using the command **File > Generate Script...**, ticking **Plot setup**. Please see section “File > Generate Script...” on page 464.

### Plot scripting

A number of built-in functions are available for scripting. For an overview of the functions, please see section “Plot” on page 570 and “Trajectories” on page 581.

### 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 438. They can also be inserted into the command log automatically. Please see section “Plots automatically inserted in command log” on page 455 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 in a number of ways:

- Display several diagrams and work with them in the same plot window.
- Display Boolean and enumeration signals.
- 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.
- Change time unit on x-axis.
- Change the display unit of a signal.
- Display signal operators.
- Plot general expressions.
- Select multiple curves in a diagram (for e.g. copying the curve values to Excel).
- Display a table instead of curves.
- Create tables from curves.
- Plot parametric curves.

- Change color, line/marker style and thickness of a signal.
- 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.
- Display a time line in plots for an animation.
- 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 > New Plot Window** will display a new plot window.

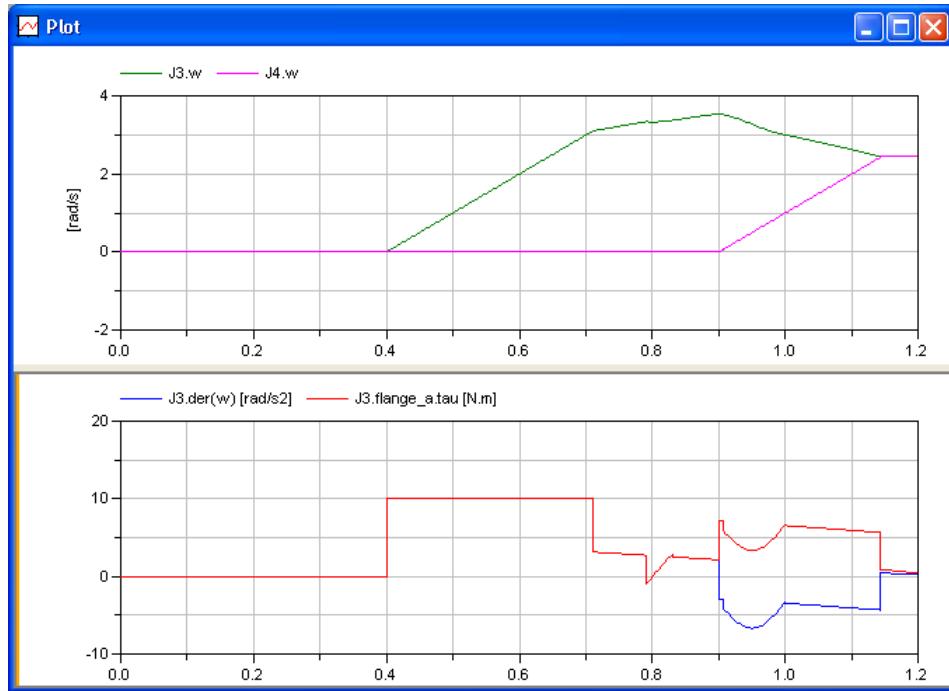
### **Plot setup menu**

In a number of interactions below, the plot setup menu is used. It can be reached in two ways; as a command **Plot > Setup...** and by right-clicking in the plot window and selecting **Setup...** in the context menu. We will below refer to it as the “plot setup menu”. (More information about it can be found in the section “Plot > Setup...” starting on page 489.)

### **Several diagrams in a plot window**

By default, a new plot window contains only one diagram. It is easy to add more diagrams and work with them.

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 in the plot toolbar and also as plot command.

**New Diagram** will create a new empty diagram in the active plot window.

**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 > Reset Window Layout**.

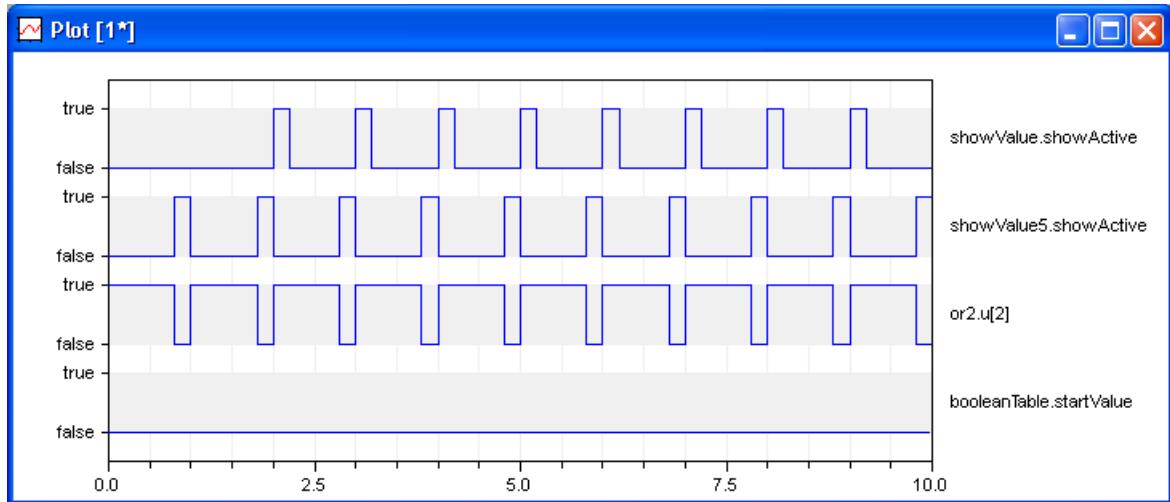
**Rescale All** will rescale all diagrams (reset them to initial zooming) in the active plot window.

**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.

## Display of Boolean and enumeration signals



The display of Boolean and enumeration 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.

### Duplicate diagram to new plot window

A plot diagram, with content, can be copied to a new plot window by the command **Show In > Duplicate Diagram** in the context menu of the diagram. (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.)

### Dynamic tooltips for signals

Resting the cursor over a plotted signal displays a tooltip with information about the closest data point. That data point is also highlighted. The closest local min and max value is also shown, 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.**

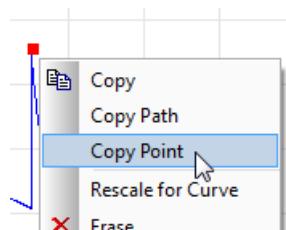


**Tooltip for multiple signals and a common independent variable; with and without additional information.**



The amount of information in the tooltip can be selected. By unchecking **Show extra information in variable tooltip** in the **Options** tab, using the command **Plot > Setup...**, local min, local max and slope will not be displayed in the tooltip.

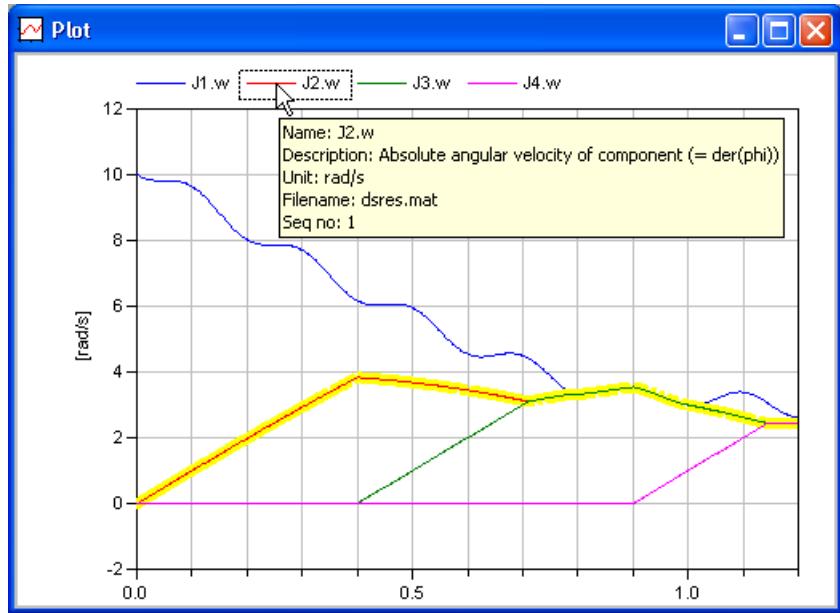
The entire text of the tooltip can be copied to clipboard by right-clicking and selecting **Copy Point** in the context menu.



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, and sequence number.



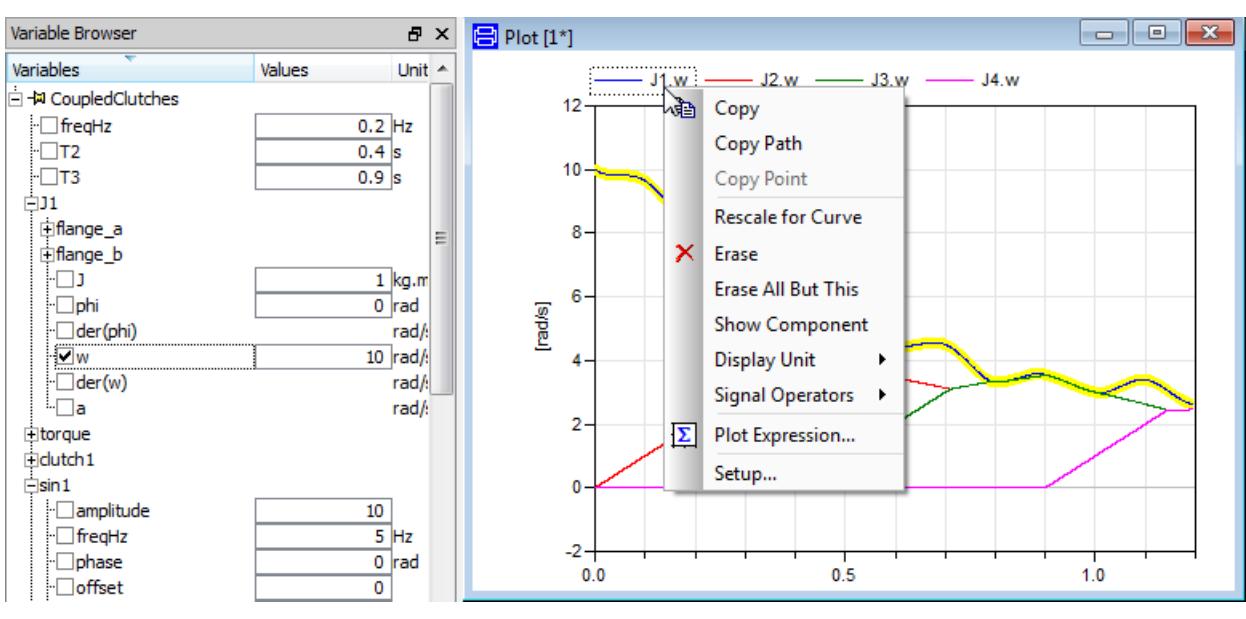
Note that the corresponding curve is highlighted in the plot window. It will stay highlighted as long as the legend symbol is selected.

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 > 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 variable browser.



### 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.

### 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**

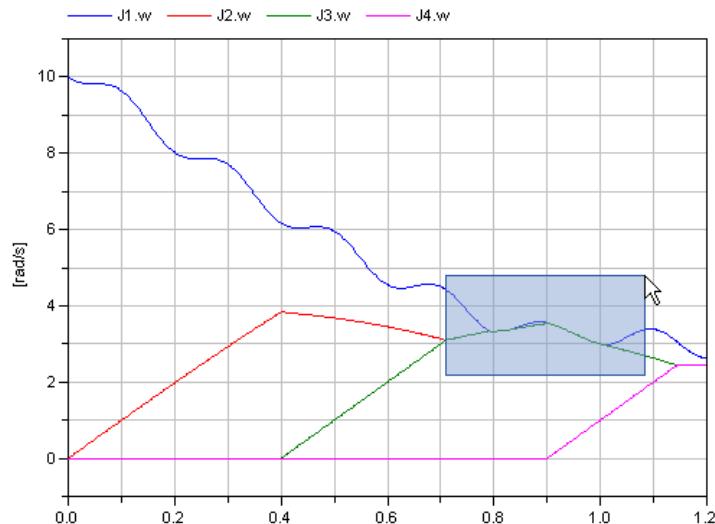
By pressing the **Ctrl** key while working in the plot window, zooming is possible using the mouse, it two ways:

- 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 418 and “Rescaling the plot window using commands/buttons” on page 418.

### **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<sup>4</sup>. Zooms in on the drawn rectangle, which is scaled to fit the diagram.



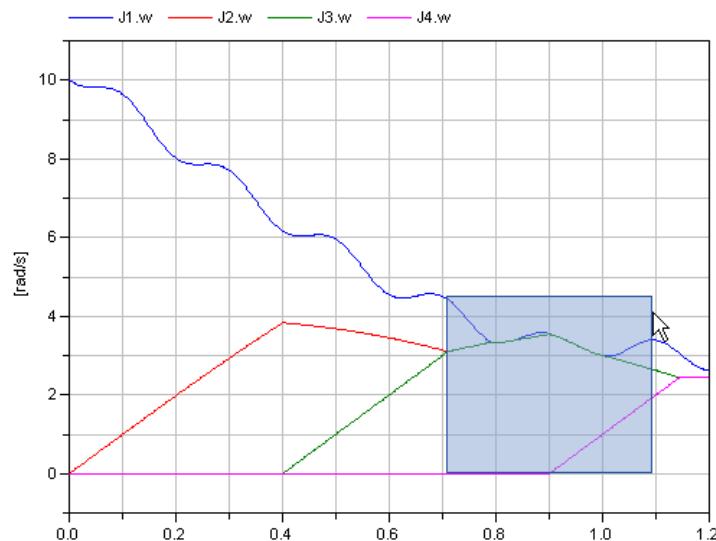

---

<sup>4</sup> 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 418 and “Rescaling the plot window using commands/buttons” on page 418.

### Zooming with maximum scaling in y direction

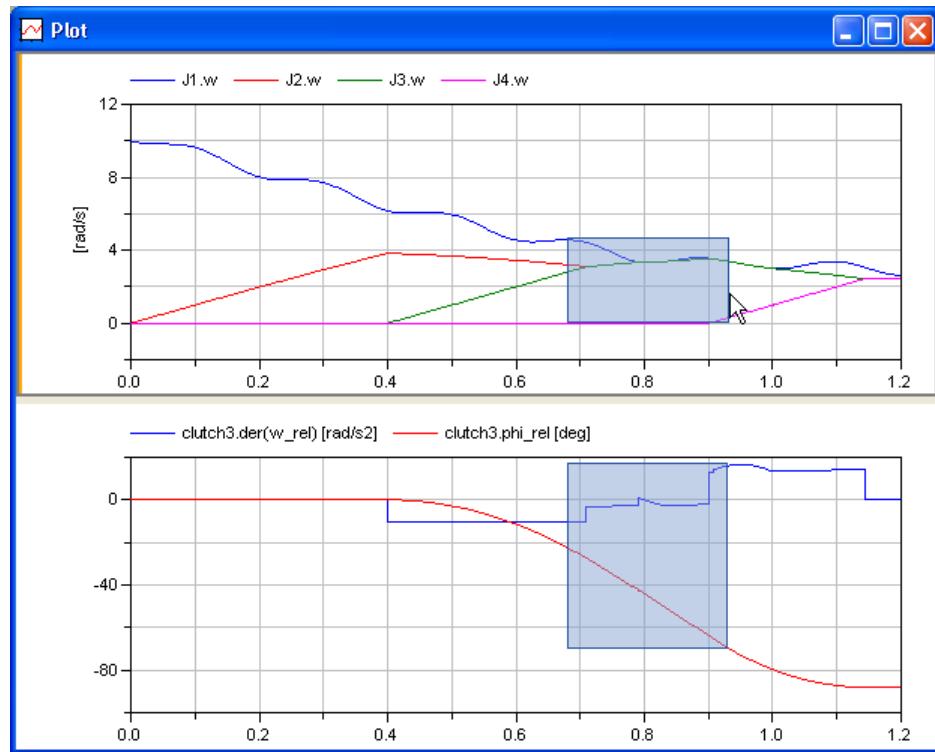
By pressing **Alt** while pressing the left mouse button and spanning the rectangle to be zoomed in<sup>5</sup>, zooming with maximum scaling in y direction is obtained. This includes all curves that are in the spanned rectangle.



Zooming with maximum scaling supports concurrent zooming; the rectangle to be zoomed will be spanned in all diagrams in the active plot window:

---

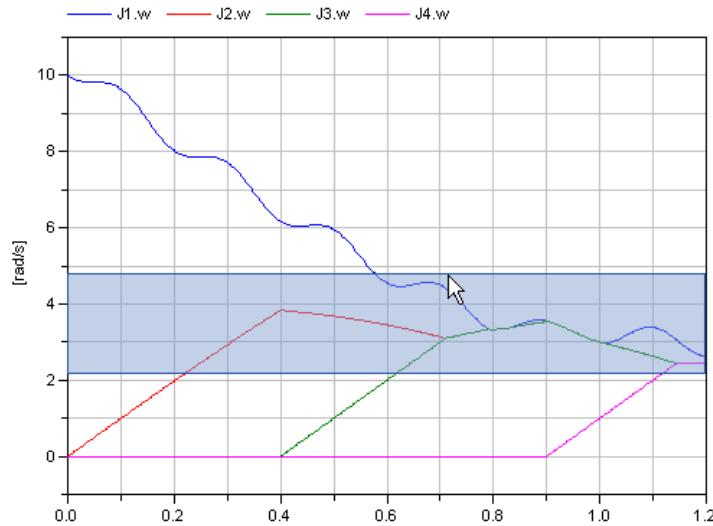
<sup>5</sup> On Linux, **Ctrl+Shift** have to be used instead of **Alt**.



Zooming out/rescaling is performed using the context menu or buttons, please see the sections “Zooming out/rescaling using context menus” on page 418 and “Rescaling the plot window using commands/buttons” on page 418.

### Vertical zooming

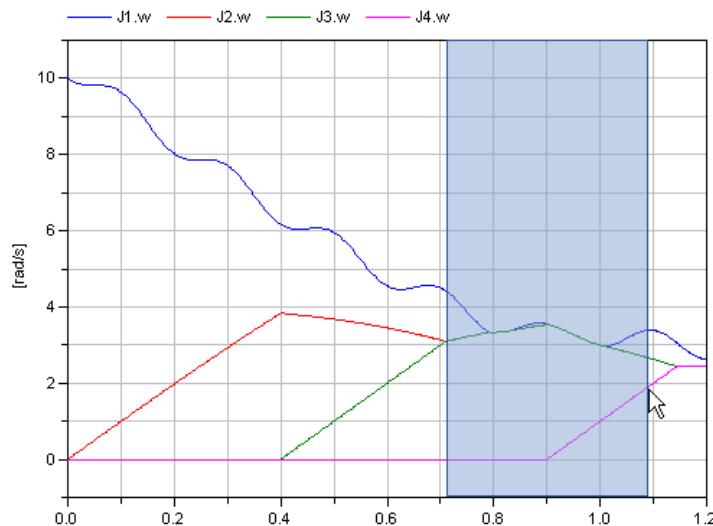
By pressing **Shift** while dragging the left mouse button up or down, a pure vertical zooming is obtained:



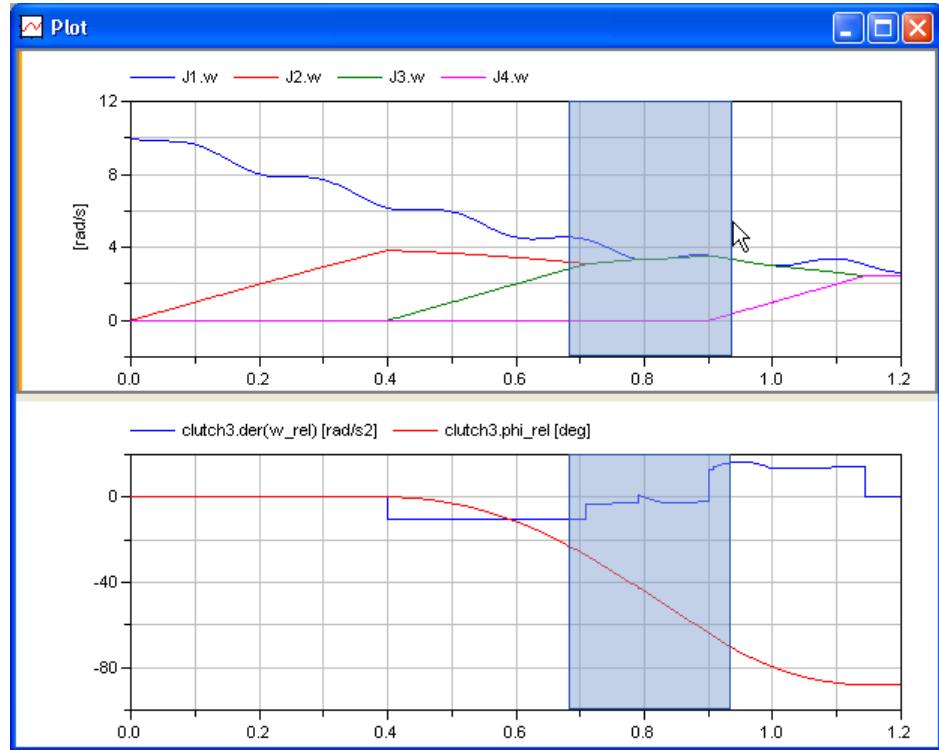
Zooming out/rescaling is performed using the context menu or buttons, please see the sections “Zooming out/rescaling using context menus” on page 418 and “Rescaling the plot window using commands/buttons” on page 418.

### 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:



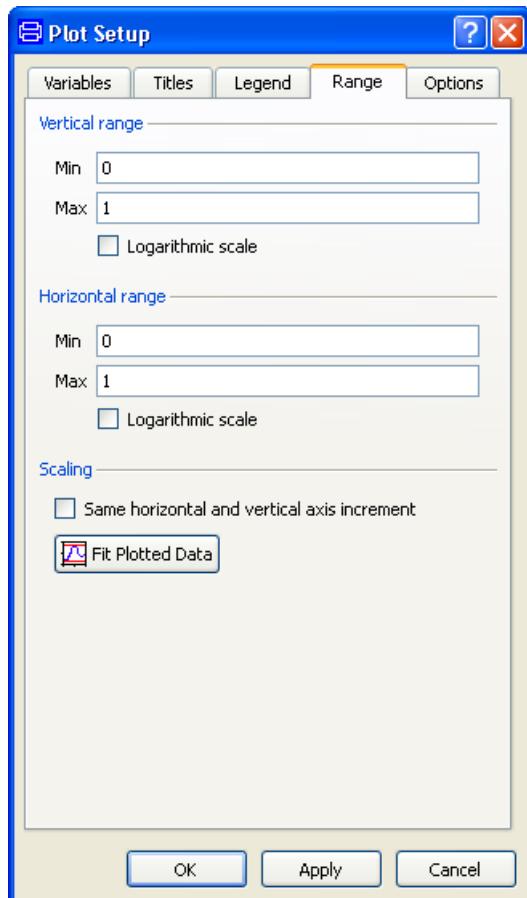
Horizontal (time axis) zooming supports concurrent zooming; the time to be zoomed will be used for all diagrams in the active plow window.



Zooming out/rescaling is performed using the context menu or buttons, please see the sections “Zooming out/rescaling using context menus” on page 418 and “Rescaling the plot window using commands/buttons” on page 418.

### Zooming in/out using the Range tab in the plot setup menu

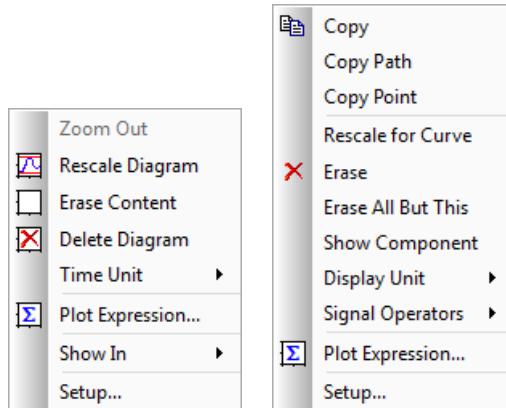
The tab looks the following:



For more information, please see the section “Plot > Setup...” starting on page 489, 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 507 and “Context menu: Plot window – curve and legend” on page 508.)

## 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.

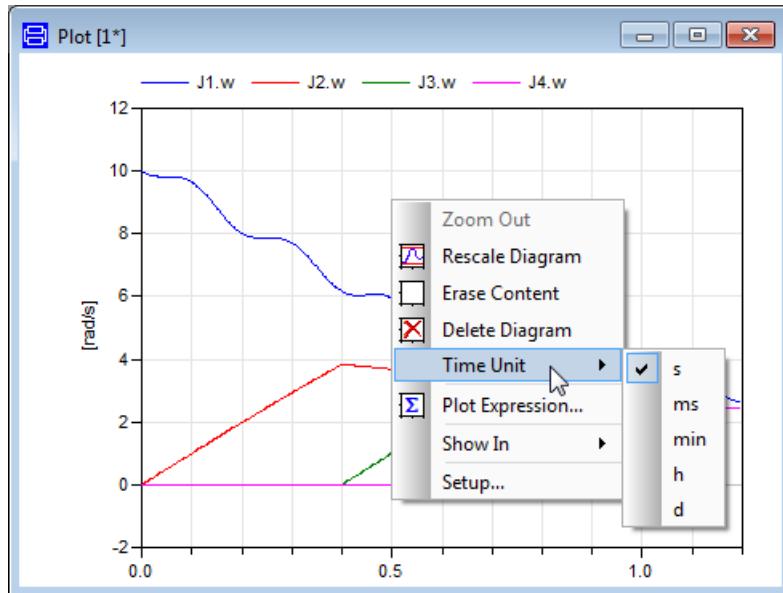


The command **Plot > Rescale All** will rescale all diagrams (reset them to initial zooming) in the active plot window. This command is not available in the plot window context menu.

The command **Plot > Rescale Diagram** 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.

## Changing time unit on x-axis

The time unit of the x-axis 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 s (seconds), if changing to another time unit 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.

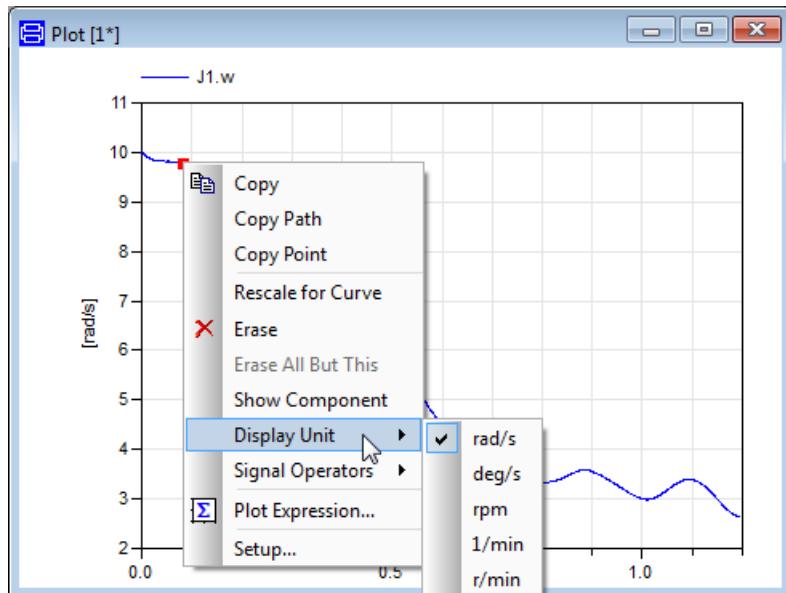
## Changing the displayed unit of signals

Changing of the displayed unit of signals can be done in two ways, either directly using the context of the plot window, or using the settings available in the plot window.

## Selecting the displayed unit directly in the context menu

Taking up the context menu in a plot window and selecting **Display unit** might display the following:

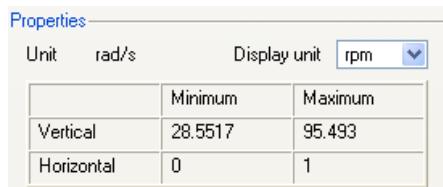
### Selecting display unit.



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 > Setup** or from the context menu). 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 (x86)\Dymola 2015 FD01\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.

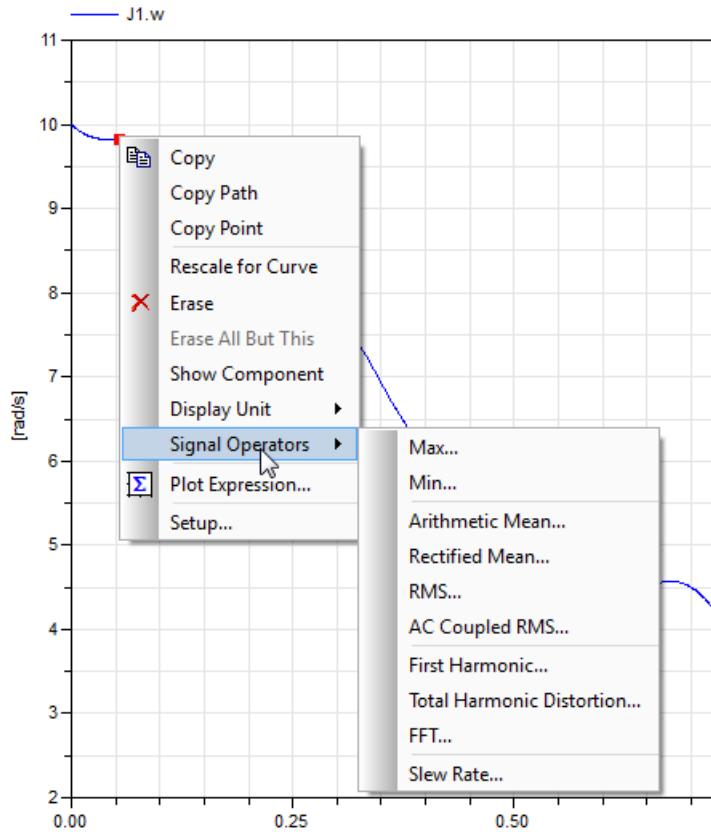
## Displaying signal operators

### 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:

- Minimum and maximum value
- Arithmetic mean
- Rectified mean
- Root-mean-square (RMS)
- AC-coupled RMS
- First harmonic
- Total harmonic distortion (THD)
- FFT (Fast Fourier Transform)
- Slew rate

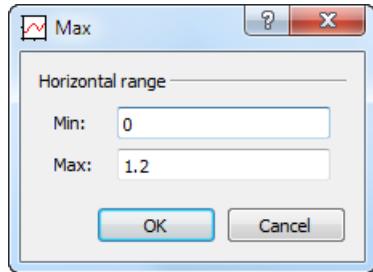
The **Signal Operator** selection in the context menu is shown below:



The signal operator functionality is also supported by scripting, using the built-in function `plotSignalOperator` (except **First Harmonic**, **Total Harmonic Distortion**, and **FFT**). For more information, please see section “`plotSignalOperator`” on page 577. Concerning scripting support of **First Harmonic** and **Total Harmonic Distortion**, please see section “`plotSignalOperatorHarmonic`” on page 578.

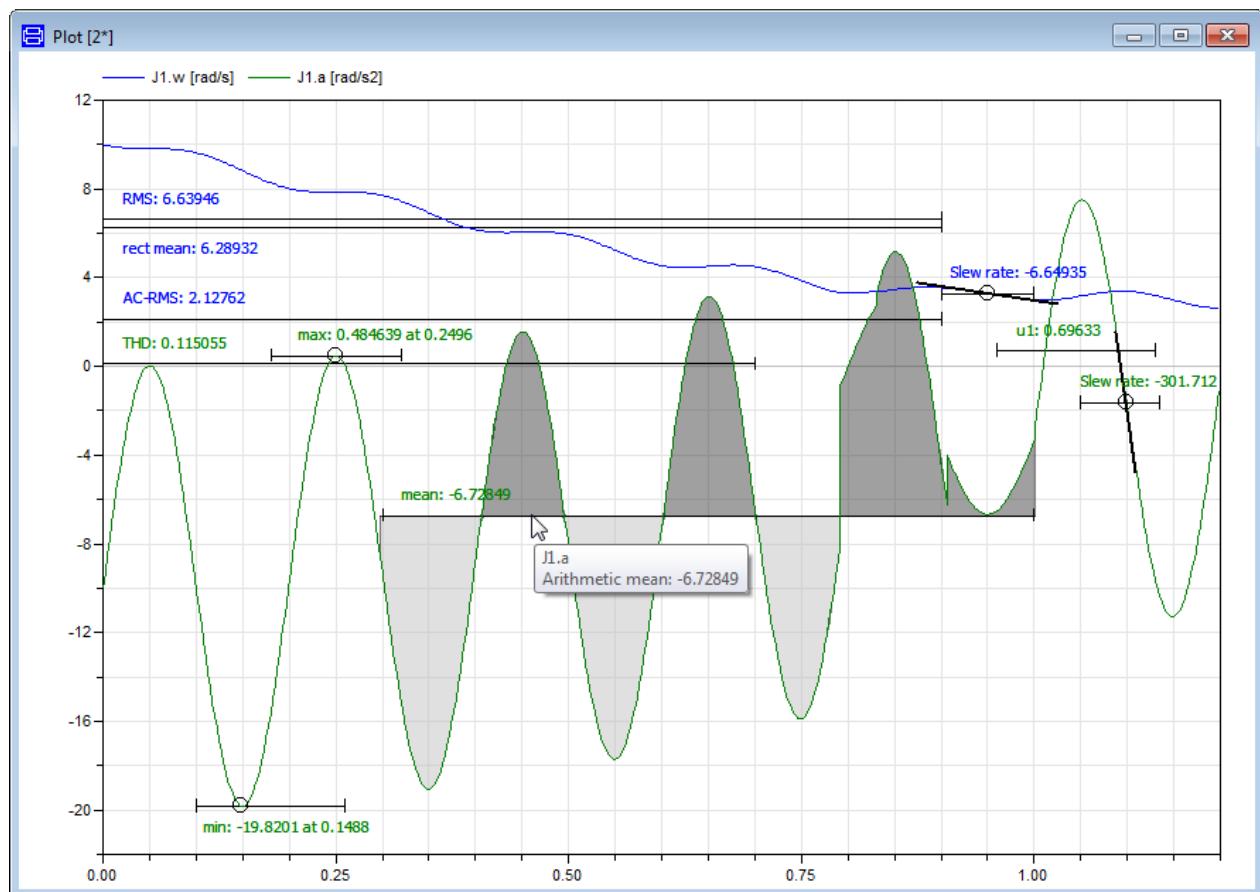
### Inserting signal operators

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.



An error message is raised if the specific time range is out of bounds or if the minimum is larger than the maximum.

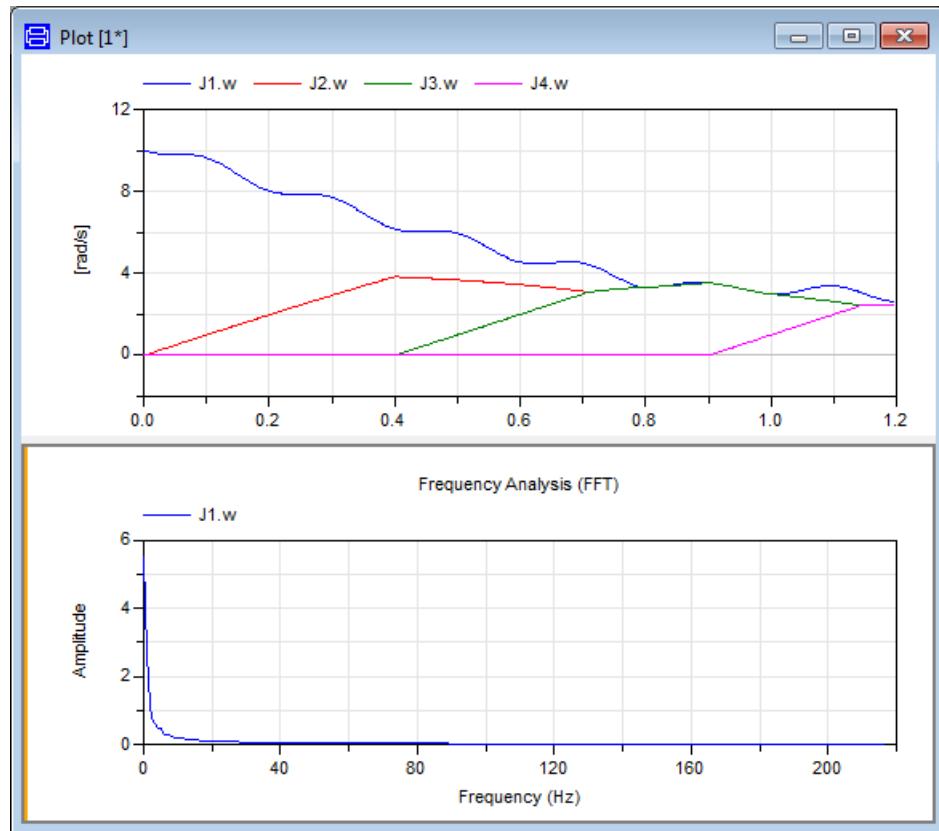
The result is then visualized in the plot window in different ways depending on which operator that was selected. The time range and the associated text are using black color by default. If the plot contains more than one curve, the text is written in the same color as the plotted signal. The picture below shows the appearance of most presently available operators (for FFT, see below):



Common for all cases, except 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.

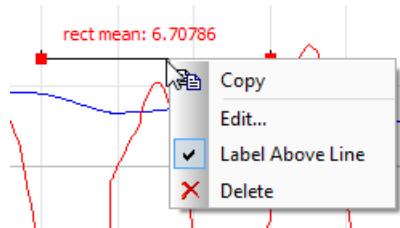
For FFT an extra diagram is created when using the operator:



### Editing signal operators

It is possible to interactively change the range of signal operators 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.

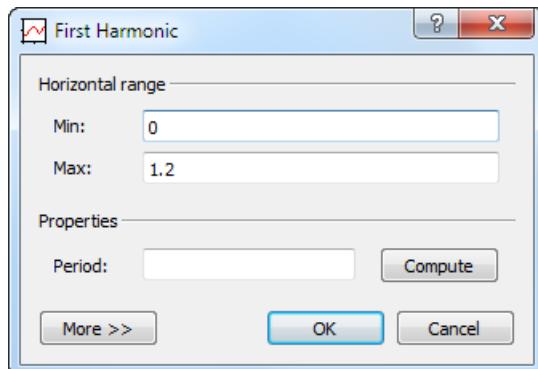
Right-clicking the signal operator displays a context menu for editing:



See “Context menu: Plot window – signal operators” on page 509 for more information about this menu.

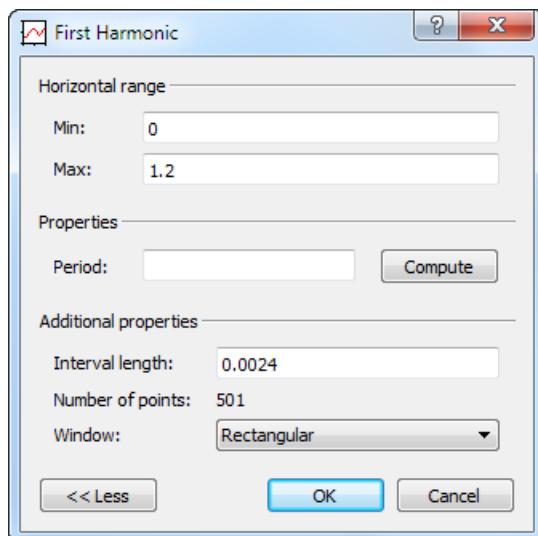
### Additional selections for First Harmonic

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**.

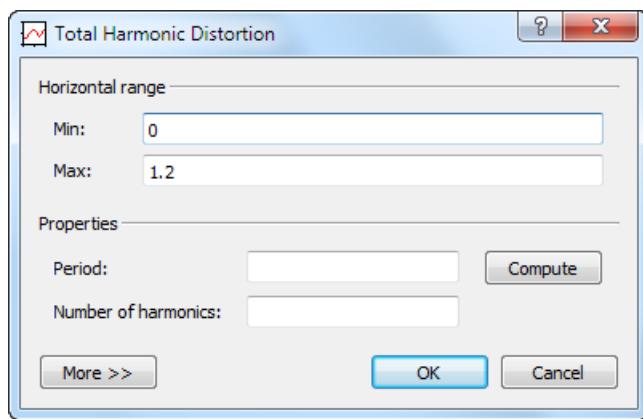
**Window** can be selected:



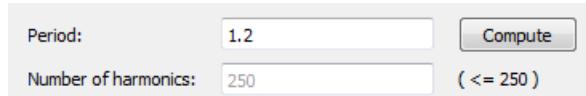
Default is **Rectangular** which gives equal weight to all points.

#### **Additional selections for Total Harmonic Distortion**

Selecting **Signal Operators > Total Harmonic Distortion...** in the context menu of a curve will display:



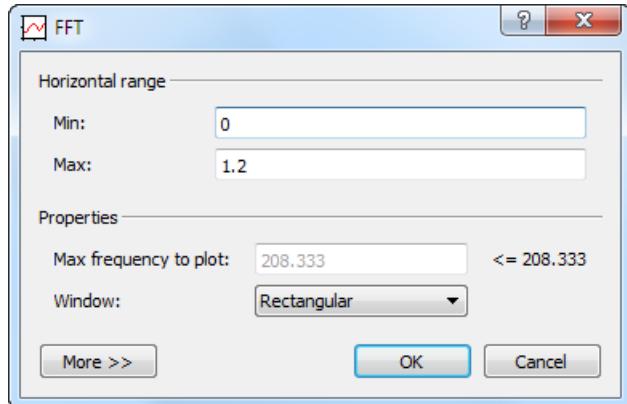
Comparing with the above command **First Harmonics**, the only difference is that for **Total Harmonic Distortion** also **Number of harmonics** can be selected. Entering or computing the period length will display limit for **Number of harmonics**:



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):

In the below min and max are omitted.

#### Arithmetic mean

This operator computes the arithmetic mean of the curve on the selected range. It is computed with the following formula (trapezoid approximation):

$$\text{Mean}(a) := \frac{\int_{\min}^{\max} a(x) dx}{\int_{\min}^{\max} dx} = \frac{\sum_{i=1}^{n-1} \left( \frac{a_i + a_{i+1}}{2} \right) * (x_{i+1} - x_i)}{\max - \min}$$

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:

$$\text{RMean}(a) := \frac{\int_{\min}^{\max} |a(x)| dx}{\int_{\min}^{\max} dx} = \frac{\sum_{i=1}^{n-1} f(a_i, a_{i+1}) * (x_{i+1} - x_i)}{\max - \min}$$

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  $\{x_1, x_2, x_3, \dots, x_n\}$  representing the values of the point of the curve, the RMS (Root-mean-square) value is given by this formula:

$$\text{RMS}(a) := \sqrt{\frac{\int_{\min}^{\max} a^2(x) dx}{\int_{\min}^{\max} dx}} = \sqrt{\frac{\sum_{i=1}^{n-1} \left(\frac{a_i^2 + a_{i+1}^2}{2}\right) * (x_{i+1} - x_i)}{\max - \min}}$$

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):

$$\text{ACRMS}(a) = \text{RMS}(a - \text{Mean}(a)) = \sqrt{\text{RMS}(a)^2 - \text{Mean}(a)^2}$$

### 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):

$$x(t) \cdot \text{window}(t) = u_0 + \sum_{k=1} u_{s,k} \cdot \sin(\omega k t) + u_{c,k} \cdot \cos(\omega k t); \quad u_k = \sqrt{u_{s,k}^2 + u_{c,k}^2}$$

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.

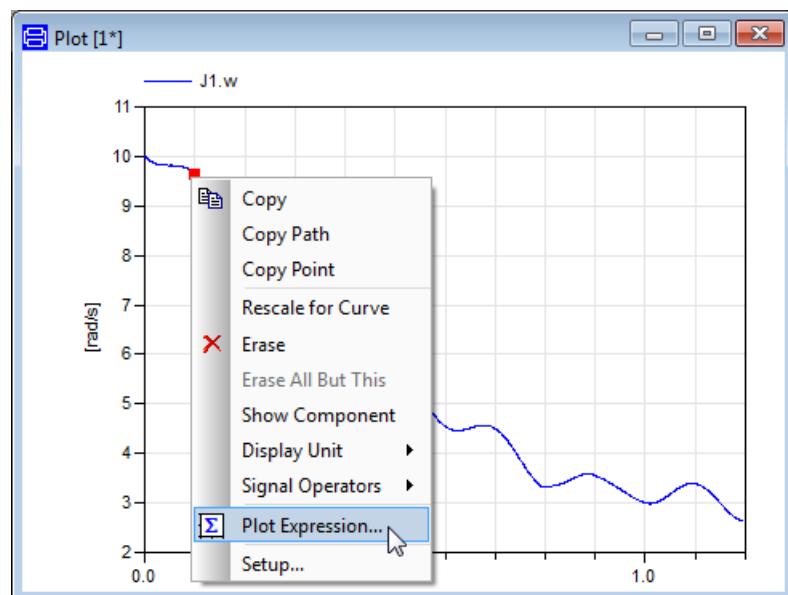
## Plotting general expressions

### Displaying the Plot Expression dialog



The functionality to create and plot a general expression can be invoked in a number of different ways:

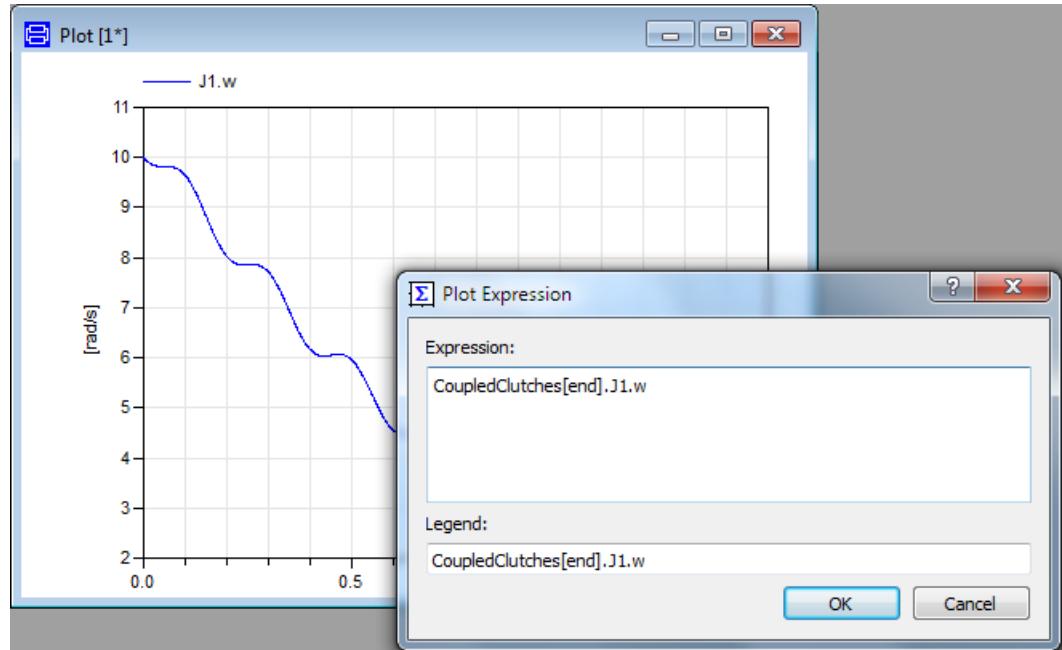
- Through the context menu of a selected signal in the plot window as shown below:



- Through the general context menu of the plot window (no signal selected).
- Through the command **Plot > Plot Expression....**
- Through the button **Plot Expression** in the **Plot** toolbar.

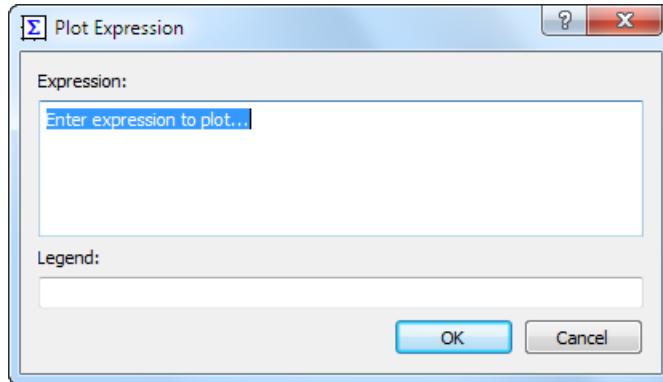


If the first alternative above has been used, the dialog for creating a plot expression will be pre-populated with the selected signal:



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.

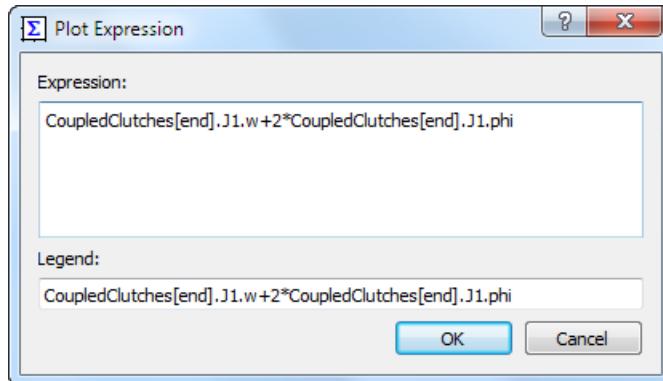
Using any of the other alternatives above, the Plot expression dialog will not contain any pre-populated signal:



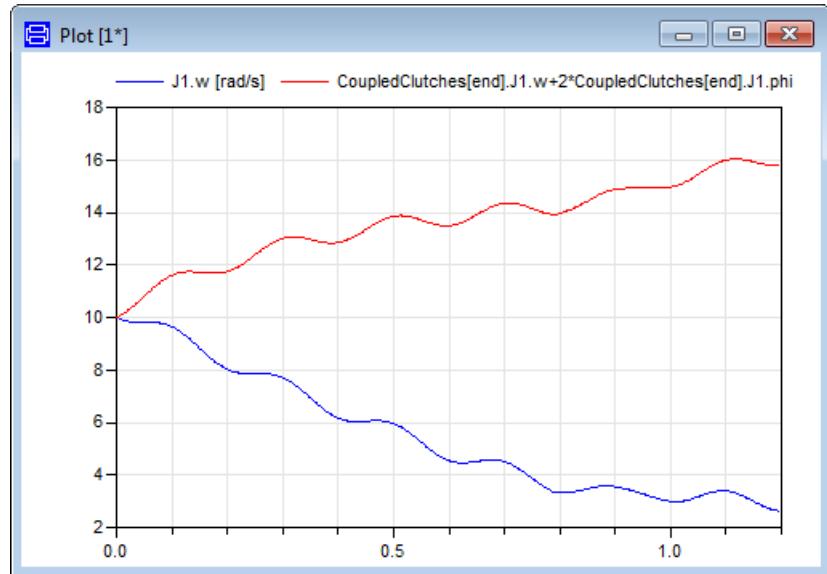
The functionality is also supported by scripting, using the built-in function `plotExpression`. Please see section “`plotExpression`” on page 576 for more information.

### Editing plot expressions

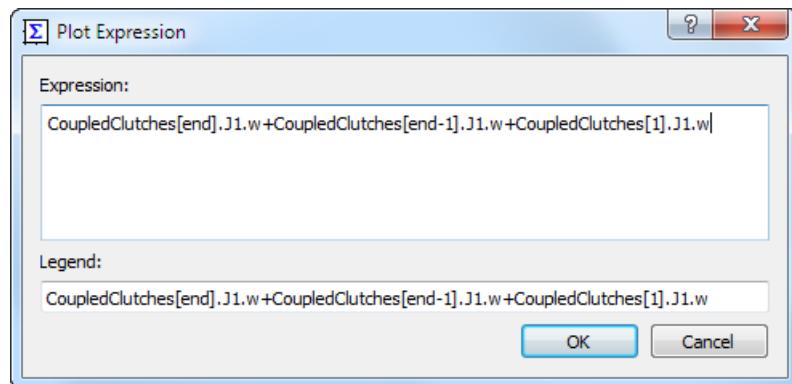
The expression in the dialog can be created by a combination of manual editing and by selecting signals from the variable browser. When a signal is selected in the variable browser, the name of the signal is automatically generated in the Plot expression dialog at the current position of the cursor. 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.



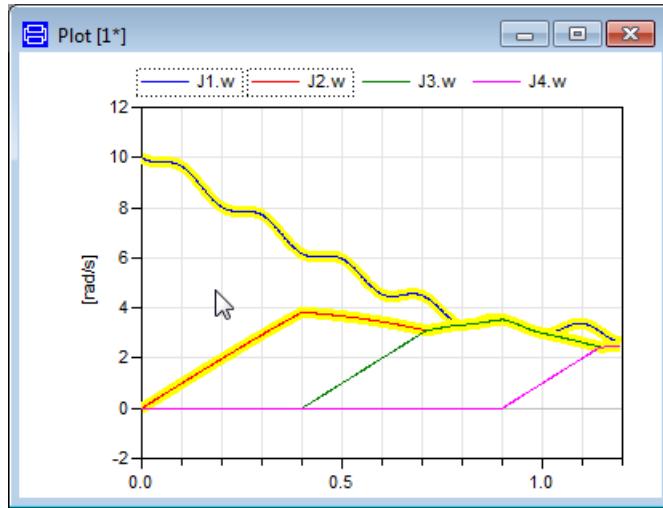
It is also possible to combine signals from different result files when building the expression. As shown already above, the notation used to indicate the result file is `resultFile[resultID]`, see the example below:



### 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 clear the selection, you can click the background.

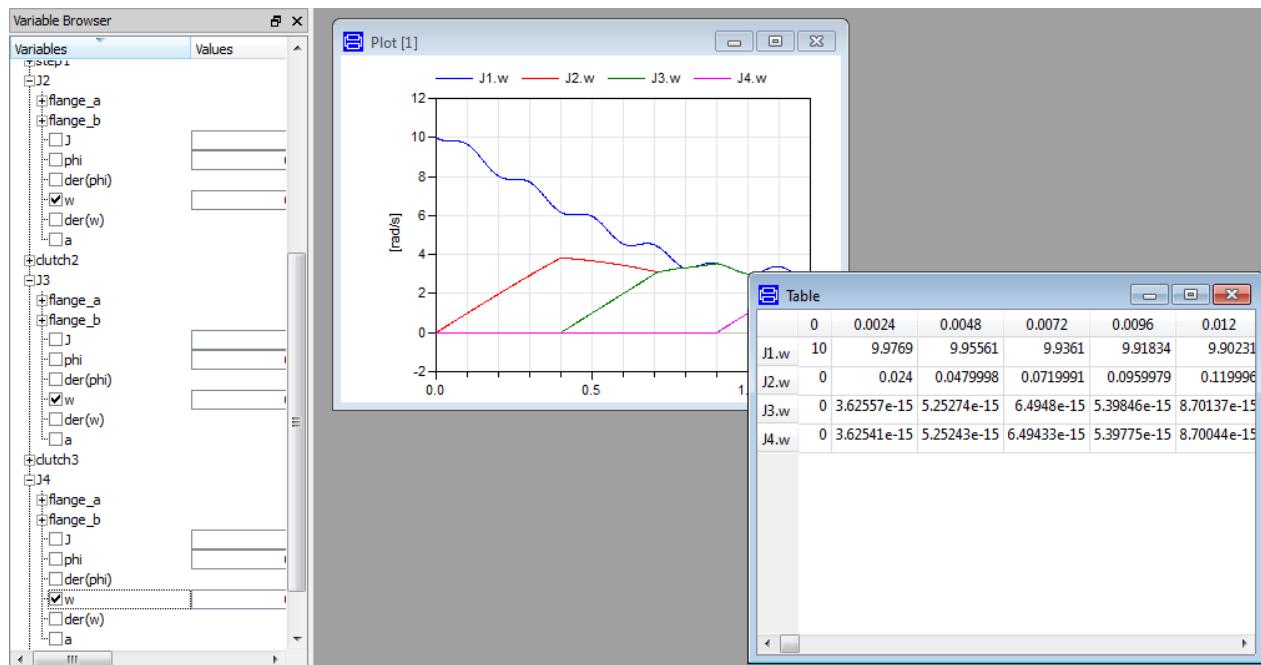
If right-clicking a curve and selecting **Copy** from the context menu, all selected curves will be copied to a tab-separated list with one column per curve. The first column contains the x-value, normally time.

	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	

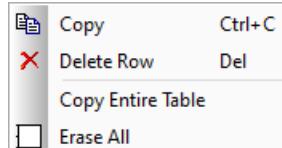
### Displaying a table instead of curves

A plot can be displayed as a table instead of a curve. To create a plot table, select the command **Plot > New Window > New Table Window**. 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 will display a context menu:



For more information about the alternative, see section “Context menu: Table window” on page 511.

### Creating tables from curves

The curves in a plot diagram can be copied to a new table window by the command **Show In > Table Window** in the context menu of a diagram. (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.)

Please also compare the previous command.

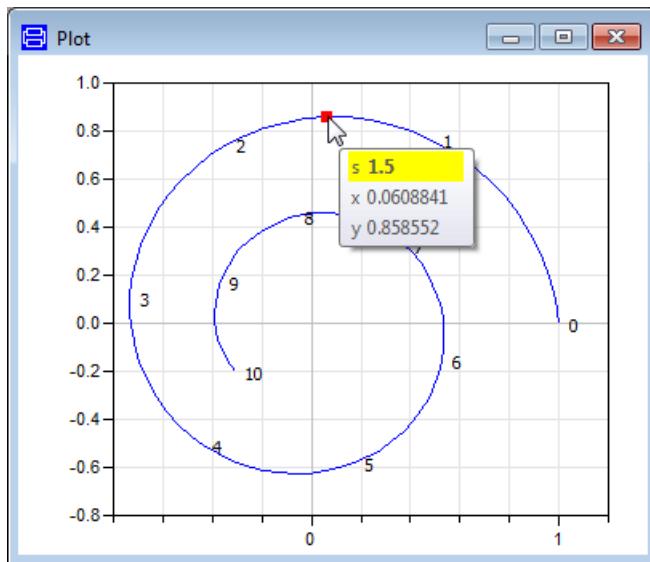
### Displaying parametric curves

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 577 and section “`plotParametricCurves`” on page 577.

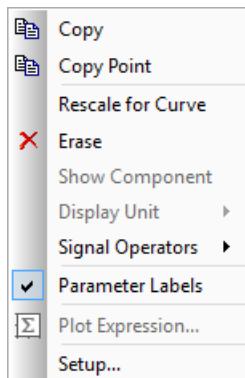
As an example of the first function, giving the following commands in the command input line of the commands window (or as a .mos script):

```
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 508.

## 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 > Setup...” on page 489, 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 570.

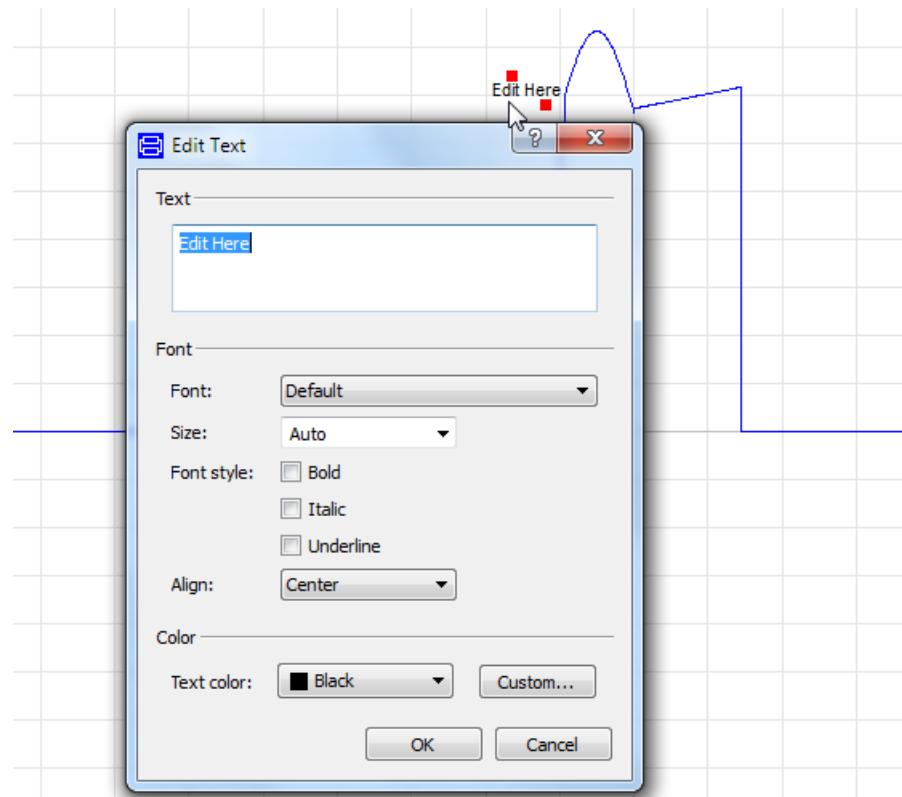
## Set diagram heading and axis titles and ranges

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 > Setup...” starting on page 489, 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 576.

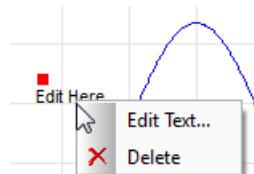
## Insert and edit text objects in the plot



Text objects can be inserted in the active diagram in a plot window using the command **Plot > Draw > Text** or the button **Text** in the plot toolbar.

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 510.

Inserting text objects in plots are also supported by scripting. Please see section “plotText” on page 579 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 > Setup...” starting on page 489, the **Legend** tab.

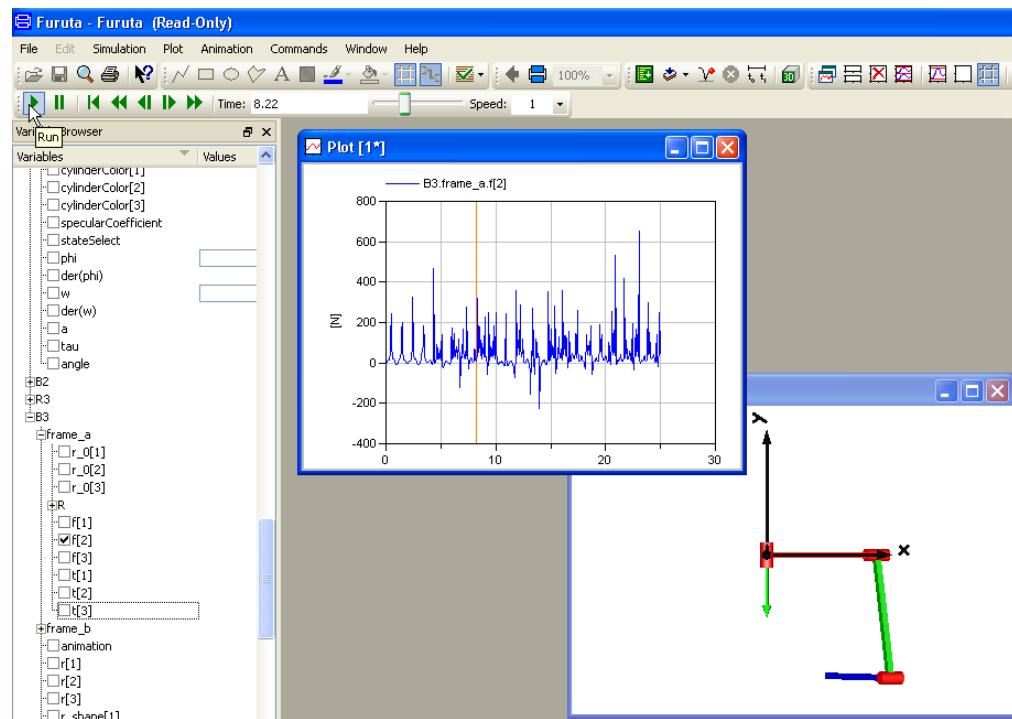
### **Displaying the component where the selected signal comes from**

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.

### **Displaying a time line**

If a plot is displaying signals of an animation, a dynamically updated time line will automatically be displayed, indicating the current time of the animation.

### Time line at $t = 8.22$ .



The time line is invisible at  $t = 0$  and at  $t = \text{max time}$ . The time line is updated in all present plot windows.

### Going back to a previously displayed plot window



The button **Recent Windows** in the Plot toolbar will toggle between the two last sub-windows. Clicking the arrow between 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.

### Insert plot and its corresponding command in command window

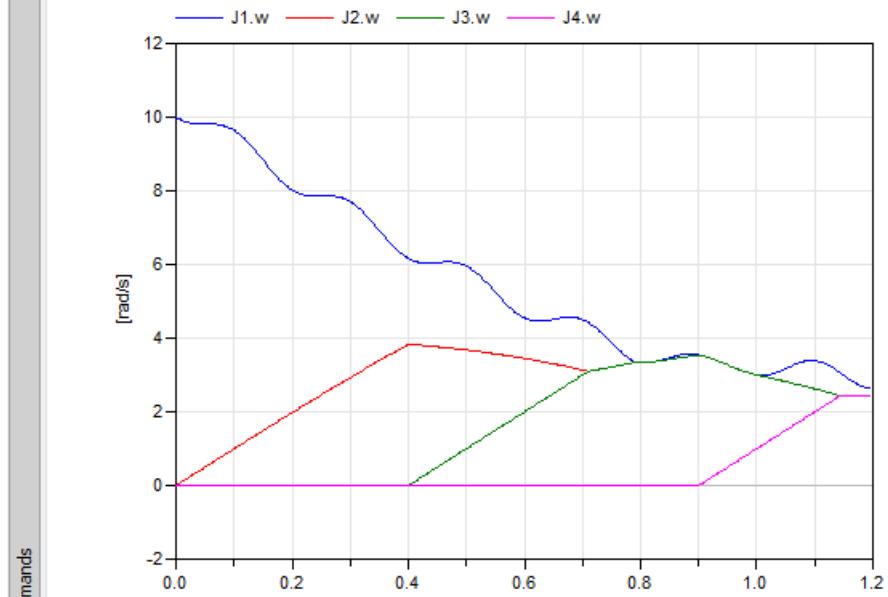
The plot and corresponding command can be inserted in the command window by the command **Show In > Command Window** in the context menu of a plot diagram. (Note that no curve etc. can be selected, neither must the cursor be over a curve or a legend if this command is to be shown in the context menu.)

An example of the result of this command:

```

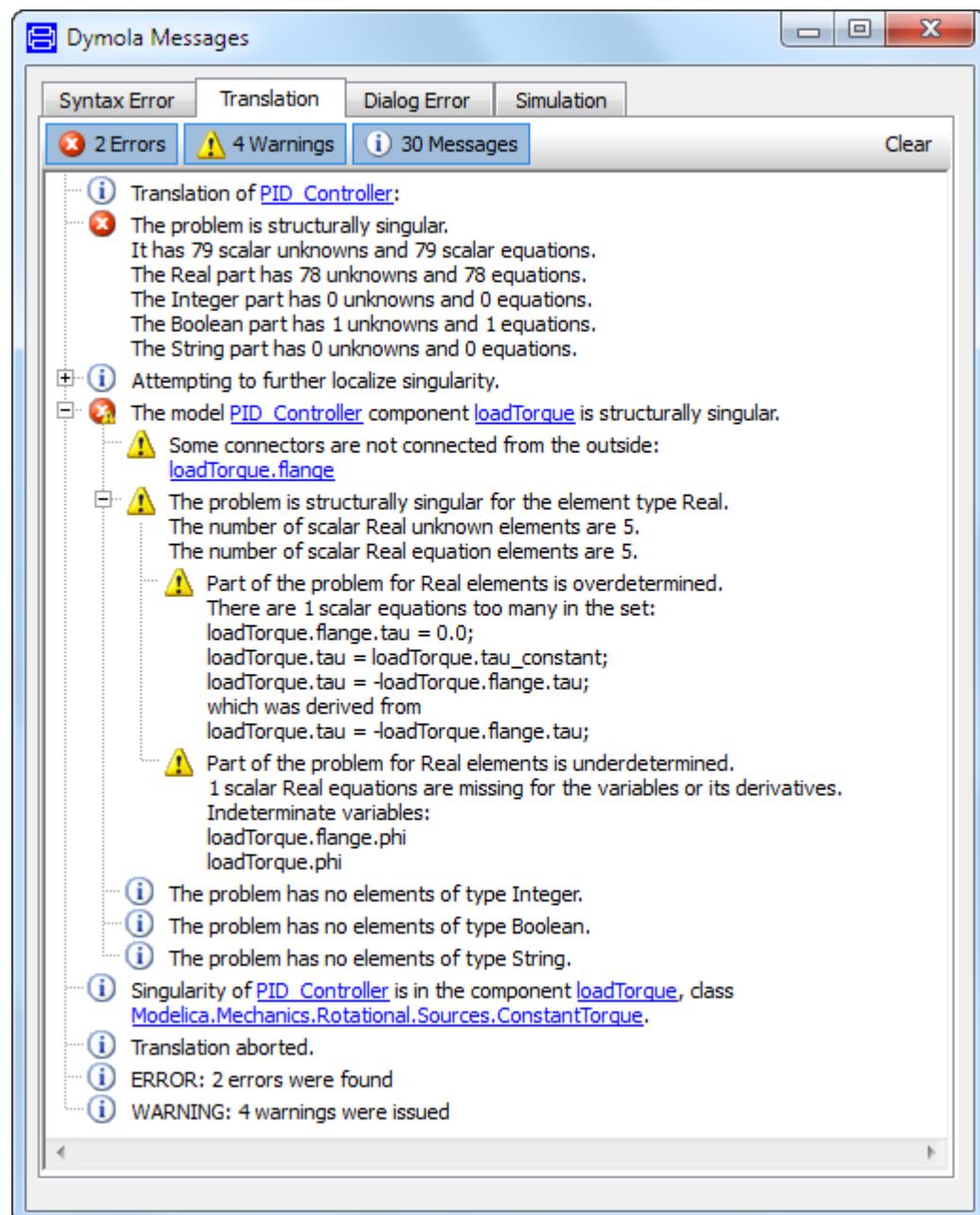
createPlot(id = 2,
position = {698, 93, 536, 381},
y = {"J1.w", "J2.w", "J3.w", "J4.w"},
range = {0.0, 1.2000000000000002, -2.0, 12.0},
autoscale = true,
autoerase = true,
autoreplot = true,
description = false,
grid = true,
color = true,
online = false,
filename = "dsres.mat",
leftTitleType = 1,
bottomTitleType = 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 message window that is automatically displayed in such case. 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 message window and the contained tabs please see section “Message window” on page 382.

## 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 message window.

If parent icons contain messages of other type, subicons are presented in the parent icon, e.g.

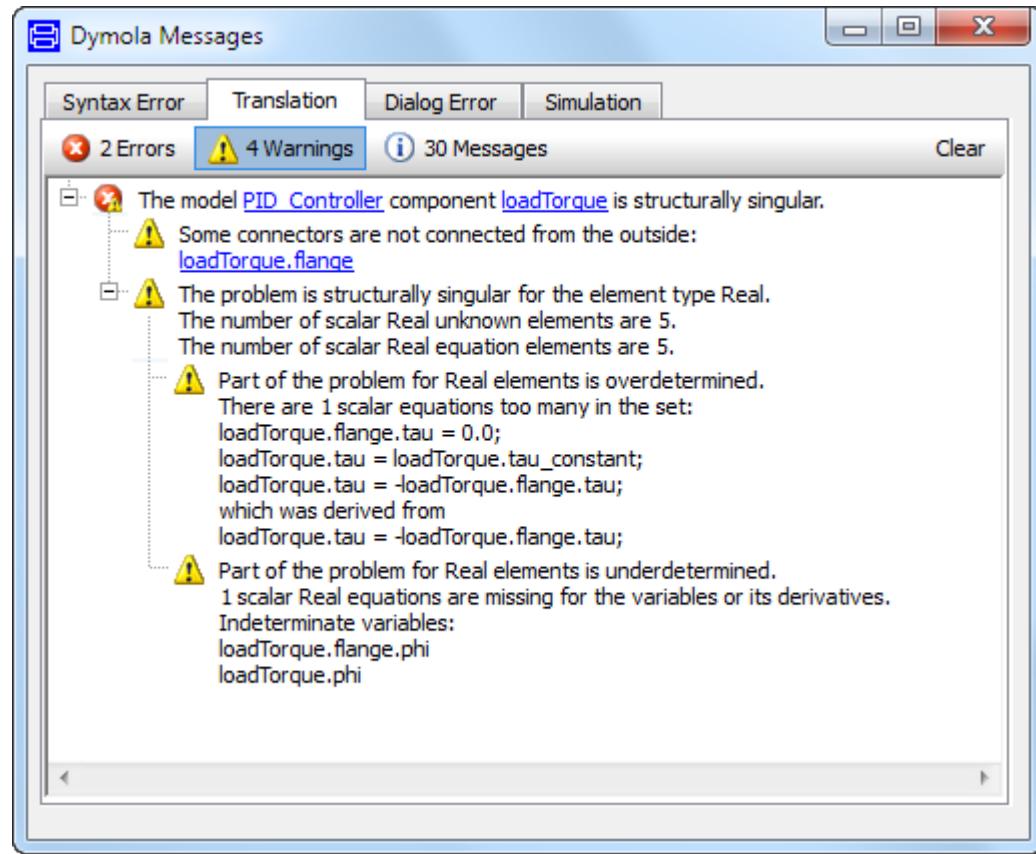
- error with warning submessages.
- information messages with warning submessages.

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. All message groups can be expanded by right-clicking and selecting **Expand All** in the context menu.

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 previous 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 group 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.

## 5.2.7 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".

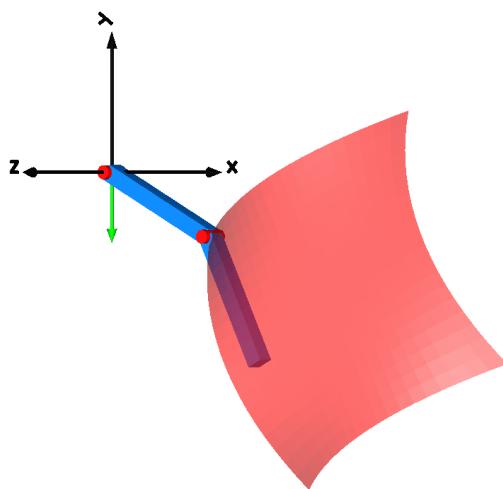
Since the DXF files contain color and dimensions for the individual faces the corresponding information in the model is currently ignored, but the specular coefficient is utilized.

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 (x86)\Dymola 2015 FD01\Modelica`.

**Example of surface-object with transparency.**



## **Animation of several result files together**

By using the variable browser context command **Animate Together** several (selected) result files can be animated together in the same window. For more information about this possibility, please see section “Context menu: Variable browser” on page 504.

### **File menu**

In the **File** menu the commands **Export... > Image...** and **Export... > Animation...** provides export of the animation window in several file formats, see “File > Export... > Image...” on page 463 and “File > Export... > Animation...” on page 463 respectively.

The animation window settings can be saved using the command **File > Generate Script...**, ticking **Animation setup**. See section “File > Generate Script...” on page 464.

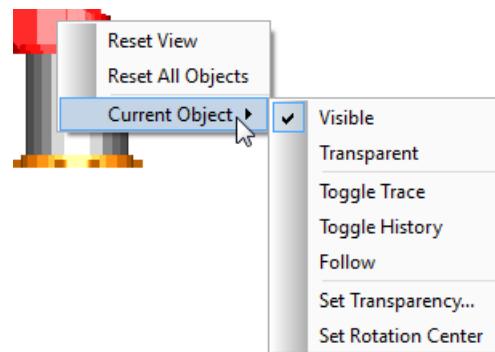
## **5.2.8 Animation window interaction**

### **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 511.

#### **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:

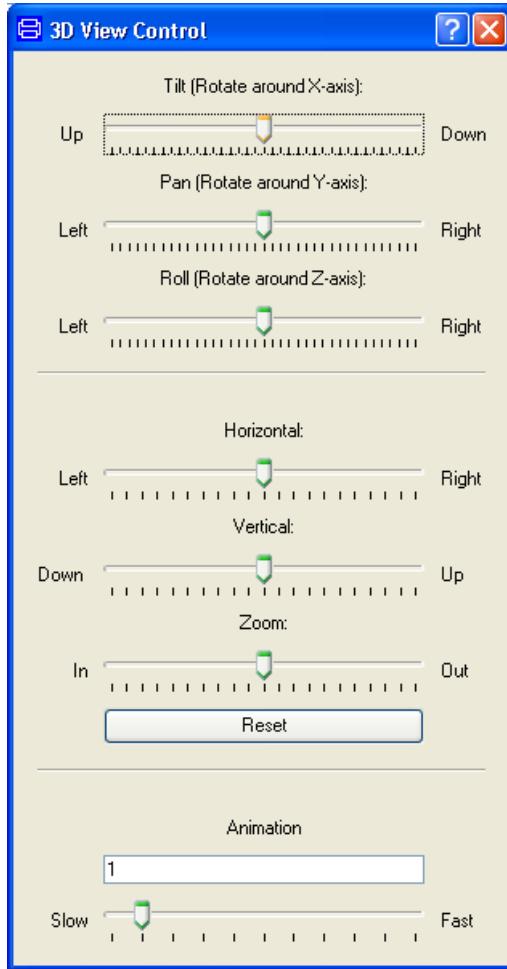
Operation	Meta key	Mouse move (dragging)	Arrow keys
Moving up/down/left/right	none	Up/Down/Left/Right	Up/Down/Left/Right
Tilt (Rotate around x-axis)	Ctrl	Up/Down	Up/Down
Pan (Rotate around y-axis)	Ctrl	Left/Right	Left/Right
Roll (rotate around z-axis)	Ctrl+Shift	Clockwise/Counter-clockwise	Left/Right
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 503.

## Running the simulation

Using the Animation toolbar (or the commands in Animation) the model can be animated.



For more information about this menu, please see section “Main window: Animation menu” on page 497.

## Going back to a previously displayed animation window



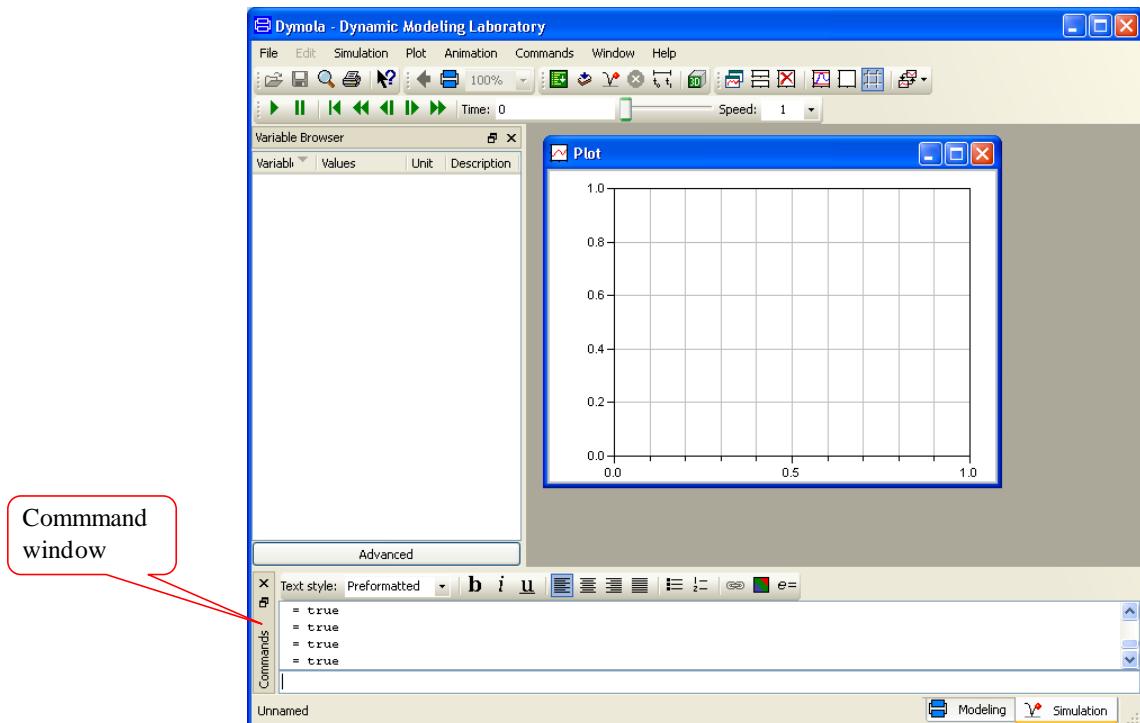
The button **Recent Windows** will toggle between the two last sub-windows. Note that this button is only available in the Plot toolbar. Clicking the arrow between the button displays a menu with all sub-windows available (plot, animation, diagram layer, vizualiser).

This button makes it easy to go back to any previously displayed sub-window.

## 5.2.9 Using the command window

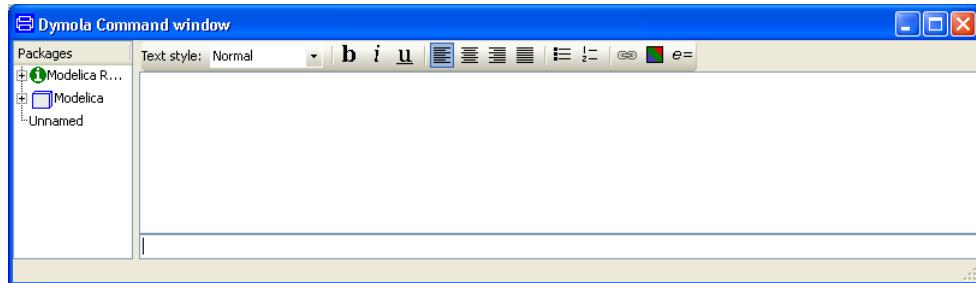
### Displaying the command window

When entering **Simulation** mode the Command window is shown by default:



The Command window is also possible to display as a stand-alone window in both **Modeling** and **Simulation** mode by using the command **Window > New Command Window**. In that case it will look like:

## A stand-alone command window.



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 sub-window of Dymola can be undocked 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.

### The command log pane

The command log pane displays the command log, containing commands given and the result of these commands, plus additional content added by the user. (The commands can be activated by command buttons, scripts or by typing in command in the command input line.)

Basically the user can work with the command log pane in two modes, the default mode of a documentation editor or a simpler text editor activated by a flag.

#### The command log pane in documentation mode

The documentation editor (and options that can be added in this mode) 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 even if the documentation editor is used, 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 in documentation mode is described in a separate section; please see section “Documentation” starting on page 452.

### **The command log pane in textual mode**

The command log pane in textual mode displays the command log in mainly textual format. This reflects looking at the command log primarily as a list of commands and the result of these.

The textual mode is activated by setting the flag

```
Advanced . ScriptingWindow.UseNewFeatures=false
```

Please note that when changing the mode (in either way) the previous part of the command log will not be displayed anymore; it will disappear once any new command is executed.

Changing this flag in either way (e.g. by typing in the command above in the command input line of the command window) will have the following consequences:

When the next command is executed, the content of the command window will be erased and the new command given (and following commands and outputs) will be displayed in the new mode.

This will also have implications concerning saving the content of the command window:

Once a new command is given in the “new mode”, the content displayed in previous mode cannot be saved as a .txt or .HTML file (using e.g. the commands **File > Save Log** or **File > Generate Script > Command log**). Only the content of the “new mode” will be saved.

However, it is still possible to save the total content as a .mos file (using any of the commands mentioned), since that file is created in another way.

The command log pane can be used to edit the text in the command log, and it is also possible to include images and html outputs. This is described in section “Basic features available in the command log pane” on page 452.

The command **File > Save Log...** can be used to save the log in various formats. Please see next section. The command **File > Clear Log** will erase the log in the command log pane.

A context menu is available for text editing. Please see section “Context menu” on page 460 for more information.

### **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 529.

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 **File > Generate Script... > Command log** can be used.

The default location for saving the log is in the working directory of Dymola. The working directory can be displayed using the command **File > Change Directory...** or 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.

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 456, information about the command button **Create link**.
- Concerning images inserted, please see section “Details of the documentation editor” starting on page 456, information about the command button **Insert image**.
- Concerning “math” objects created by e.g. the command **Insert equation or expression**, see the section “Storage of “math” objects referred by a stored command log” on page 453.

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!

Please note the possibility to 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.)

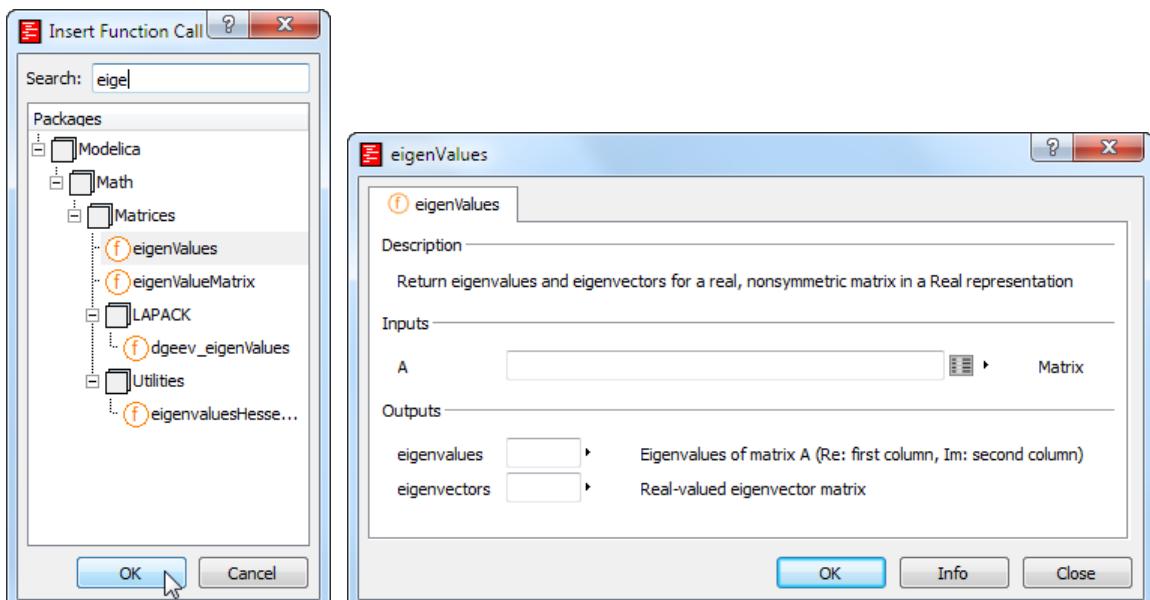
The command input line (and the command log) is important when creating scripts. Please see section “Scripting” starting on page 529 for more information.

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 515.

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 557)

## 5.2.10 Documentation

### General

This section describes the command log pane in documentation mode; some parts are however also applicable to the older textual mode of the command log pane, if so specified.

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

This section is applicable also when using the command log pane as a more basic text editor.

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 **File > 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 please see section “File > Export... > Image...” on page 463. (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 455 and “Animation window content automatically inserted in command log” on page 456.)

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 in the documentation mode

A number of additional features are available in the command log pane when in documentation mode compared to being the older text mode:

- Documentation editor

- Math rendering of formulas and equations etc, including Greek characters and index rendering.
- 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 input from the command input line in the command window is analyzed by Dymola. If the input is evaluated as an expression, the input is presented in a formatted way, “math formatting”. If the input cannot be evaluated as an expression (e.g. if semicolon “;” concludes the input) then the input will 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 button Insert equation or expression**

Equations/expressions inserted using the button **Insert equation or expression** will always be treated the same way as “math” objects when it comes to rendering. 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 **Insert equation or expression** 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. E.g. `alpha` is rendered  $\alpha$ , and `Gamma` is rendered  $\Gamma$ . This also applies when the Greek character is trivially embedded, as for e.g. `alpha_2`, rendered  $\alpha_2$ . Trailing numbers following Greek characters are rendered as subscripts; e.g. `alpha12` is rendered  $\alpha_{12}$ .

Greek characters can of course also be used as indexes, e.g. `x_beta` will be rendered  $x_\beta$ .

The rendering of Greek characters is by default active, to disable it, set the flag

```
Advanced.RenderGreekLetters=false;
```

## Index rendering

Index within brackets [ ] are rendered as subscripts, e.g. `x[2]` is rendered as  $x_2$  and `alpha12[3]` is rendered as  $\alpha_{123}$ .

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  $v_{123}$ .

## Some examples of rendering

Modelica expression	Rendering
division <code>(a*(1+b))/(1+c)</code>	$\frac{a(1+b)}{1+c}$
square root <code>sqrt(1/(1+a))</code>	$\sqrt{\frac{1}{1+\alpha}}$
power <code>(1+a)^(1+sigma)</code>	$(1+\alpha)^{1+\sigma}$
absolute values <code>abs(1/(a-3))</code>	$\left  \frac{1}{\alpha - 3} \right $
sum <code>sum({1,2,3})</code>	$\Sigma\{1, 2, 3\}$

simple matrix (declaration) <code>A=[1,2;3,4]</code>	$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$
derivative of x <code>der(x)</code>	$\frac{dx}{dt}$
array construct <code>{i+j for i in 1:3, j in 4:7}</code>	$\{i+j \text{ for } i \in 1:3, j \in 4:7\}$
if-then-else-expressions <code>y = if x&lt;0 then -x else if x==0 then 0 else if x==1 then 1 else x</code>	$y = \begin{cases} -x & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ 1 & \text{if } x = 1 \\ x & \text{else} \end{cases}$
logical expressions <code>p and not (p or q)</code>	$p \wedge \neg(p \vee q)$

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:

$$= \text{Modelica\_LinearSystems.StateSpace}\left(A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, B = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, C = [1 \ 2], D = [0]\right)$$

### 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 context command **Show In > Command Window**. See section “Insert plot and its corresponding command in command window” on page 438.

### **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:



**Text style** contains presently four selections, **Normal** being default. **Custom** cannot be selected, but will be the result if selecting any text that has been changed in any way (e.g. bold, italics) compared to the other styles. **Preformatted** is the text style of text generated from Dymola.



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 **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: **b**



The **Italic** button will change the text to italics; the **Underline** button will change the text to underline. The **Center align** button will change the text to centered. They work the same way as the bold button when it comes to activation etc.



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.

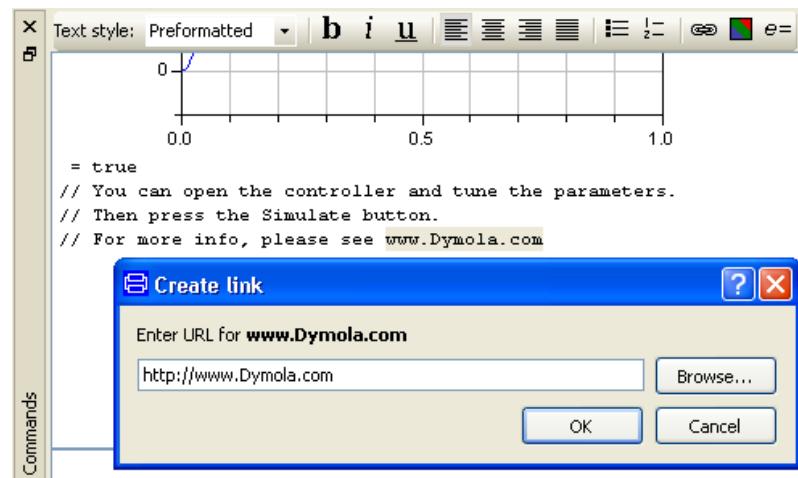


**Bullet list** will create a bullet list; **Numbered list** will create a numbered list. Presently only one level is available. They work the same as the bold button when it comes to activation etc.



The **Create link** button will open a menu for entering of an URL according to usual html rules. If a link is already present, that link will be shown.

#### Example of link creation.



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. There is no syntax check of the URL. The links cannot be activated from the command window, the log has to be 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 **Create 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 **Insert image** button, please see below.



The **Insert 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.



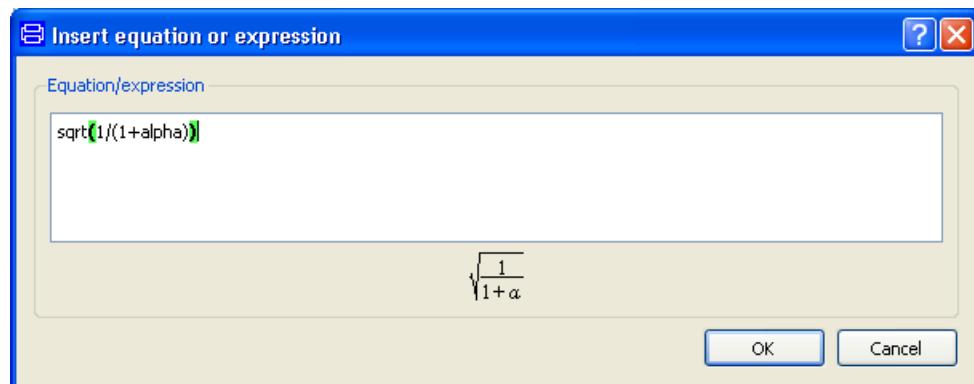
If the link is invalid (image not found) in the shown html log it will be indicated with a symbol.

Please note that the command **File > 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 section “File > Export... > Image...” on page 463 for more information about this command.



Clicking the button **Insert equation or expression** 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 **Insert equation or expression** button in the first place.

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. Taking up the context menu by right-clicking and selecting **Edit equation** 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 **Insert equation or expression** 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 453.

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 513.

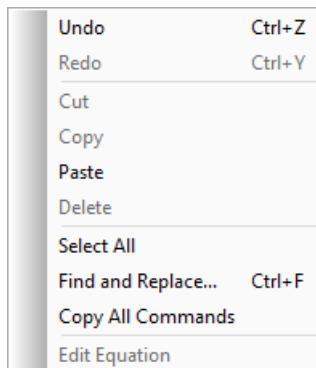
## Commands

The command **File > Save Log...** can be used to save the log in various formats. Please see section “The command log file” on page 449. The command **File > Clear Log** will erase the log in the command log pane.

## Context menu

A context menu 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 513.

## 5.2.11 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...” on page 470 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 **File > Generate Script....** Please see previous chapter, section “Editor command reference”, sub-section “Main window: File menu”, command “File > Generate Script...”. 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 529.

### 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. Please see section “Translation tab” on page 473 for more information. (This 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.)

- Specify that a certain parameter should be evaluated<sup>6</sup>. 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 setting above. Please see chapter “Developing a model”, section “Advanced model editing”, sub-section “Parameters, variables and constants” for more information about these annotations.

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”.

### **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 475 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.)

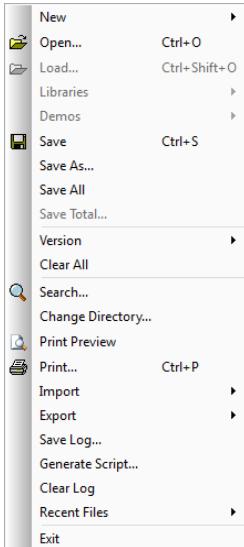
Please note the close relation with hidden variables. A variable can be declared as hidden by using the annotation `annotation(HideResult=true)`. Such a variable will not be stored. Please see previous chapter, section “Advanced model editing”, sub-section “Parameters, variables and constants” for more information concerning this.

---

<sup>6</sup> Note that if a parameter having annotation `Evaluate=true` is bound to a parameter expression the parameter expression will also be evaluated, unless it involves parameters with annotation `Evaluate=false`.

## 5.3 Editor command reference – Simulation mode

### 5.3.1 Main window: File menu



The selection of commands that can be selected is different from what can be selected in Modeling mode. For information about commands not listed here, please see the previous chapter, section “Editor command reference – Modeling”. Only some commands are further commented below.

#### File > Open...

In Simulation mode, this command allows opening Modelica script files (.mos files). The script is opened in the Dymola script editor.

#### File > Save

In Simulation mode, this command is available when the script editor is open. The command saves the active script of the script editor. Note that the save command is also available in the script editor.

#### File > Save As...

In Simulation mode, this command is available when the script editor is open. The command saves the active script of the script editor with another name. Note that a corresponding command is *not* available in the script editor.

#### File > Save All

Saves all modified model classes with a File > Save operation. This operation can also be made in Modeling mode. The reason that this command is also available in Simulation mode is to have an opportunity to save the complete model during a simulation without having to return to the Modeling mode.

#### File > Print...

Prints the contents of the window on the current printer.

In Windows (not in Linux) there are some further options: If the active window is a plot window, please note that it is possible to set the resolution for high resolution printing (or export) to clipboard. The variable is called Advanced.ClipboardResolution; default is 600 dpi. It is also possible to export low resolution plots (96 dpi) in Dymola by setting Advanced.PrintLowRes = true; this will print (and copy) the plot window using 96 dpi, which is compatible with earlier versions of Dymola.

#### File > Import > FMU

This command corresponds to the built-in function `importFMU` of the selected FMU, with import settings defined in the FMI tab of the simulation setup, reached by the command **Simulation > Setup....** For more information about this command, please see section “FMI tab” on page 484.

Note that dragging an .fmu file into Dymola main window will perform this command on that file.

### **File > 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 are pasted. We suggest using **Edit > Paste Special** and selecting Enhanced Metafile Format. (Newer programs, e.g. Microsoft Word 2003 do not have this problem.)

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; default is 600 dpi. It is also possible to export low resolution plots (96 dpi) in Dymola by setting Advanced.PrintLowRes = true; this will copy (and print) the plot window using 96 dpi, which is compatible with earlier versions of Dymola.

This command is only available in the Windows version of Dymola.

### **File > Export... > Image...**

Saves a PNG or SVG image of the contents of the active window (without window borders and plot bar, if any). The user is prompted for a file name.

The image is identical to the image shown in the window, so the size and representation can be changed by first resizing the window. The relevant active window in Simulation mode should usually be a plot window in this case.

Exported images are included in the command log.

For more information about this command, please see the corresponding section in the chapter “Developing a model”.

### **File > 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.

For more information about this command, please see the corresponding section in the chapter “Developing a model”.

### **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 log, without any images but including command output.
- As script file, containing all commands given but without the output from the commands.

For more information about the command log and saving it in different ways, please see section “The command log file” starting on page 449.

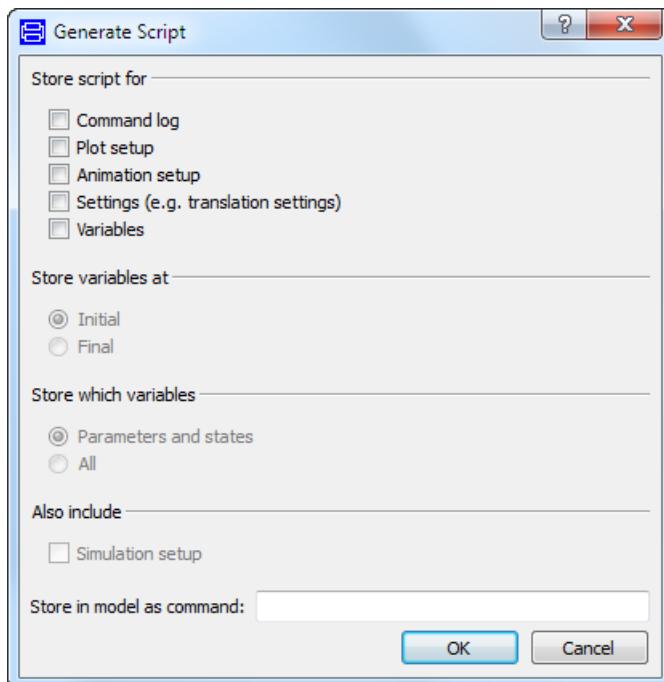
### **File > Generate Script...**

Saves various settings in a script file. 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 541.

The command corresponds to the function `saveSettings` (except storing the command log and store the script as a command in the model). Please see section “`saveSettings`” on page 569 for more information (and section “`savelog`” on page 569 for storing of the command log).

**Generate script file alternatives.**



The **Store script** for group specifies what should be stored in the script. The command log can be saved. Alternatively switch settings, plot and animation setup, and variables can be saved. Concerning the **Plot Setup** alternative, note that besides being possible to obtain this functionality also using the built-in function `saveSettings` (see above), this setup can also be saved (however except what variables are shown) using the command **Edit > Options...**, the **Save Settings** tab, the **Plot layout and setup** alternative. Please see this command in previous chapter for more information.

The **Store variables at** group decide whether specified initial values or final values after simulation should be stored.

The **Store which variables** group decides whether all variables or only parameters and initial values of states should be saved.

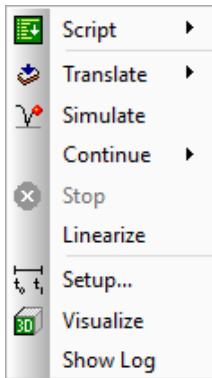
The **Also include** group gives possibility to also include simulation setup, that is, start time, stop time and integrator.

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.

### **File > Clear Log**

Erases the contents of the command log in the command log pane of the command window.

### 5.3.2 Main window: Simulation menu



The simulation menu includes commands to setup and run simulations.

#### Toolbar

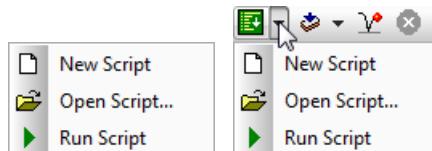
The most common simulation commands are also available in the toolbar:



#### Simulation > Script > 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 selected by the simulation menu or by the simulation toolbar.



**Run Script...** is the default command when clicking the **Run Script** button in the simulation toolbar.

The scripting facilities are described in more detail in “Scripting” starting on page 529.

#### Simulation > Script > 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 selected by the simulation menu or by the simulation toolbar, see images above.

The scripting facilities are described in more detail in “Scripting” starting on page 529

#### Simulation > Script > 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 selected by the simulation menu or by the simulation toolbar, see images above.

The scripting facilities are described in more detail in “Scripting” starting on page 529

### Simulation > Translate > Normal

Translates the active model to simulation code. Error messages and warnings are shown in the message window. After successful translation, the class is ready for simulation.

The command can be selected by the simulation menu or by the simulation toolbar.



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 470.

The “normal” translation is default when clicking the **Translate** button in the simulation toolbar.

### Simulation > Translate > Export

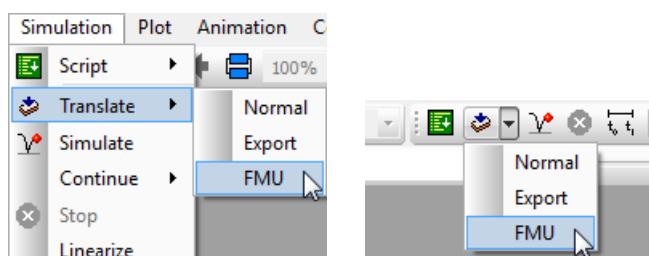
Translates the active model to code executable on any platform without a Dymola license at the target system. The command corresponds to the built-in function `translateModelExport`.

The “export” simulation must be selected by using the drop-down menu when using the **Translate** button in the simulation toolbar. See image above.

This command demands license. For more information, please see the manual “Dymola User Manual Volume 2”, chapter 6 “Other Simulation Environments”, section “Code and Model Export”.

### Simulation > Translate > FMU

**Translate to FMU  
using command or  
button.**



Translates the active model to an FMU, with the export settings specified in the FMI tab of the simulation setup, reached by the command **Simulation > Setup....** See section “FMI tab” on page 484 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.

## **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....**

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 559.

## **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 previous 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.

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 602). 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.

## Simulation > Linearize

The command **Simulate > 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 page 561, 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 562.

## Simulation > Setup...

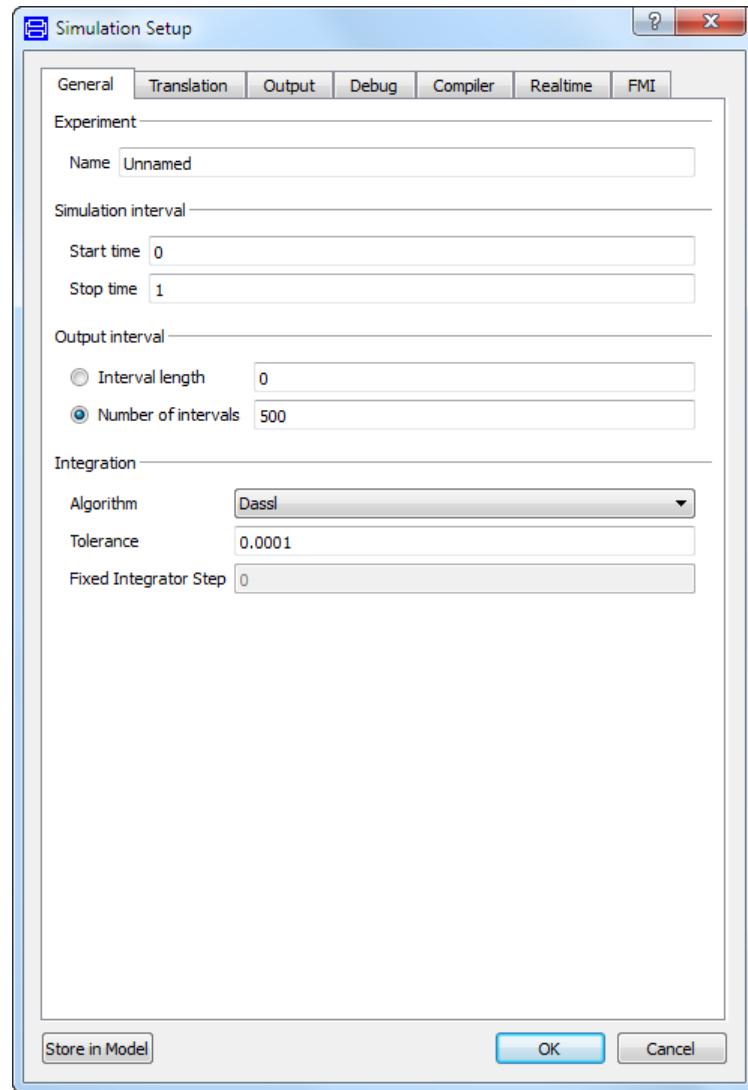


Setup opens a dialogue for specifying experiment name, simulation interval, integration algorithm etc.

The information from this tab can also be stored in the current model by clicking the button **Store in model**. This only applies to the **General** and **Output** tabs. When the model is later selected, these settings appear in the simulation setup dialog and can be changed before simulation.

## General tab

General tab of Simulation > Setup...



The **General** tab includes:

An **Experiment** group where **Name** specifies the name of the experiment. It is used to name the result file.

A **Simulation interval** group where **Start time** and **Stop time** for the simulation can be specified.

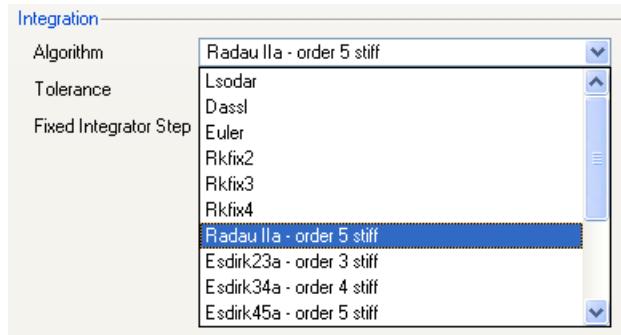
An **Output interval** group to specify how often results shall be stored. It can be specified in terms of **Interval length** or **Number of Intervals** for the simulation. By default the results are also stored at discrete events.

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.

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**).

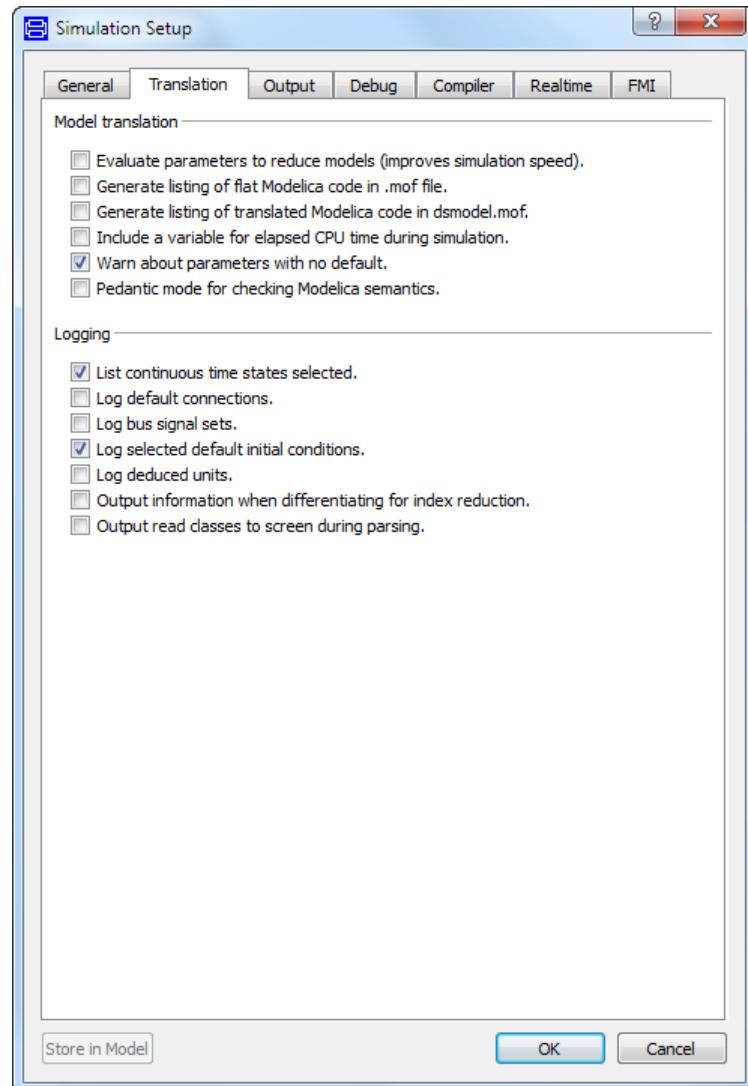
More information about the algorithms is given in the section “Selecting the integration algorithm” starting on page 522.

**Store in model**

The information from this tab can also be stored in the current model by clicking the button **Store in model**. When the model is later selected, these settings appear in the simulation setup dialog and can be changed before simulation.

## 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).

For more information about evaluation of parameters (e.g. the possibility to set evaluation selectively for certain parameters) see section “Parameter evaluation” on page 460.

**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 589 for more information about this.

**Include a variable for elapsed CPU time during simulation** introduces two new 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.

**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.

**List continuous time states selected** lists the continuous time states selected.

**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 384.

**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”.

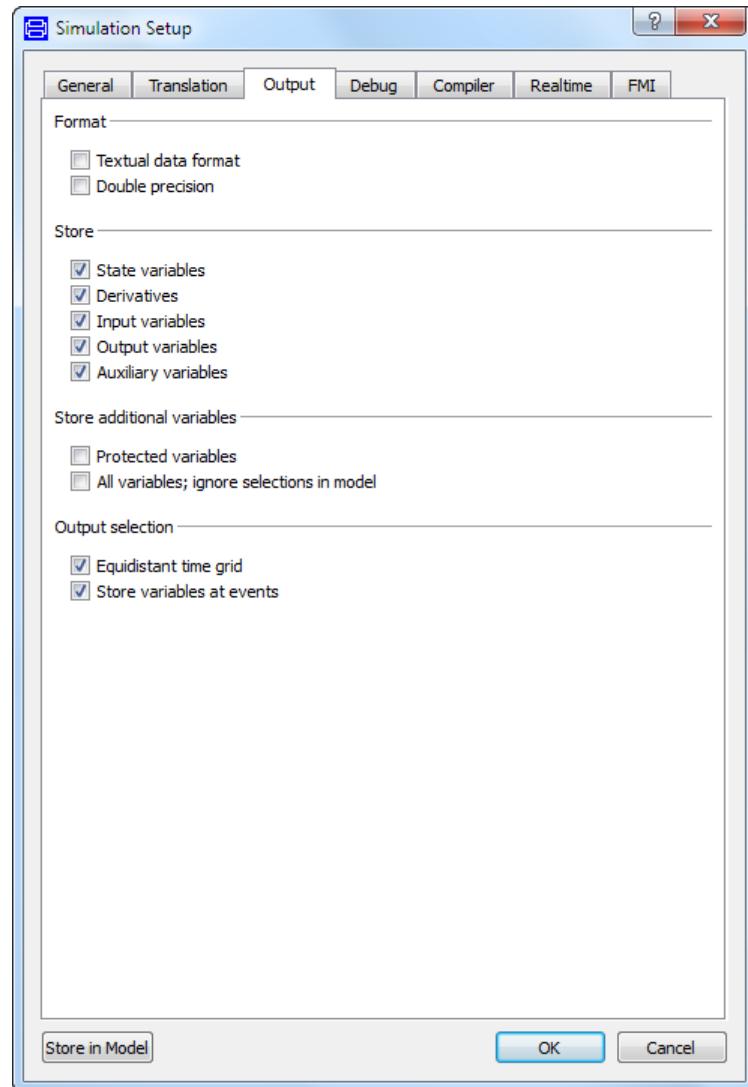
**Output information when differentiating for index reduction** reports about which equations that are differentiated.

**Output read classes to screen during parsing** reports on the read classes during parsing.

(The **Store in model** button only applies to the tabs **General** and **Output**.)

### 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.

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 461. 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.

**Equidistant time grid** stores result equidistantly as given by simulation setup.

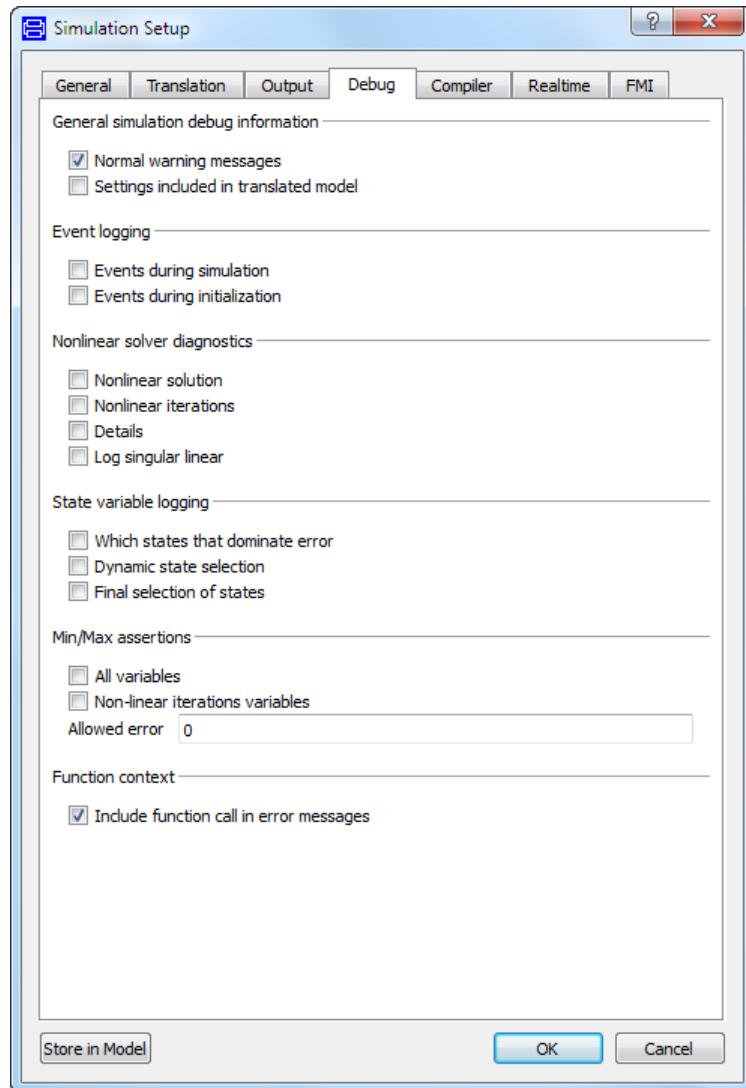
**Store variables at events** stores variables before and after events.

**Store in model**

The information from this tab can also be stored in the current model by clicking the button **Store in model**. When the model is later selected, these settings appear in the simulation setup dialog and can be changed before simulation.

## Debug tab

### 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 583.

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**

**Normal warning messages** If you want to disable all non-fatal warnings and errors.

**Settings included in translated model** Useful if you want to debug the model when not running inside Dymola.

## **Event logging group**

**Events during simulation** Log events during the simulation. This feature is useful for finding errors in the event logic including chattering.

**Events during initialization** Also log events during the initial phase.

## **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 manual “Dymola User Manual Volume 2”, chapter “Advanced Modelica Support”, section “Using noEvent”).

**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**

**Which states that dominate error** - if the simulation is slow due to tolerance requirements for some state variable this can help in which variables are the critical ones.

**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 manual “Dymola User Manual Volume 2”, 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, please see the section “Bound checking for variables” on page 600.

**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.

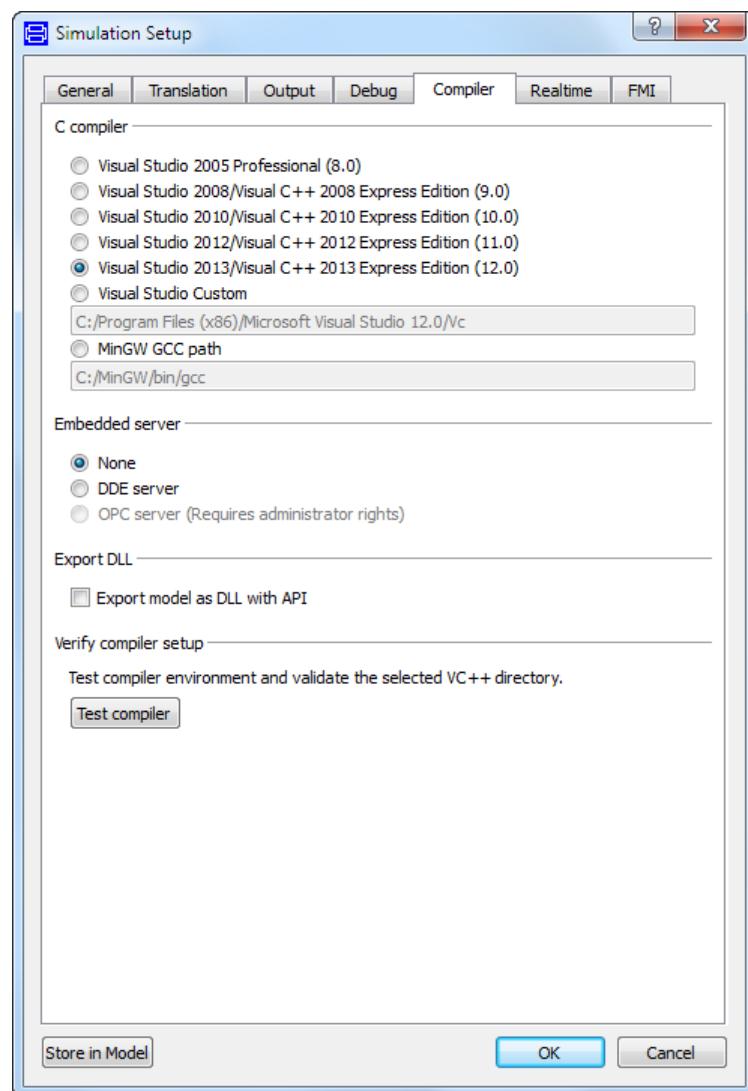
## **Function context group**

**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.

(The **Store in model** button only applies to the tabs **General** and **Output**.)

**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.

## 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”. If support for a compiler has not been selected when installing Dymola, the corresponding choice is disabled, see also chapter “Appendix – Installation”, section “Installation on Windows”, sub-section “Installing a C compiler”.

Note that although the Visual Studio 2013 and 2012 compilers are fully supported, these compilers by default generate a bit less efficient code than previous versions of the compiler, with the selected optimization settings. As a temporary work-around you can set the flag

```
Advanced.Define.GlobalOptimizations = 2;
```

before generating code, to activate global optimization in the compiler. (The default value of the flag is 0.)

This flag works the same for all Visual Studio compilers, but the effect on compilers of previous versions is small. For the Visual Studio 2013 and 2012 compiler, however, the simulation performance is restored, but the compilation of the code might take substantially longer for large models. The setting corresponds to the compiler command /Og.

Concerning consequences of not writeable directories etc please see 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 chapter “Appendix – Installation”, section “Installation on Linux”.

## Embedded server

If real-time simulation should be used under Windows, **DDE server** or **OPC Server** must be checked. Checking this alternative means that Dymosim will be compiled to provide a DDE server or OPC server. **Note** that if real-time *synchronization* should be used, **DDE server** must be selected.

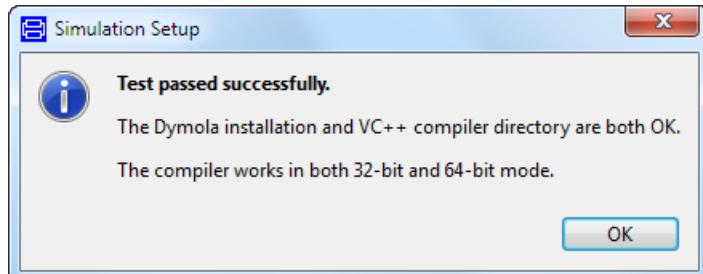
For more information about these servers, please see the manual “Dymola User Manual Volume 2”, chapter 6 “Other Simulation Environments”.

## 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 manual “Dymola User Manual Volume 2”, chapter 6 “Other Simulation Environments”. Note that this feature will in some later release be replaced by the use of FMI.

## Verify compiler setup group

Several different kinds of problems may occasionally arise when compiling the generated C code produced by Dymola. Pressing the **Test 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 window will pop:

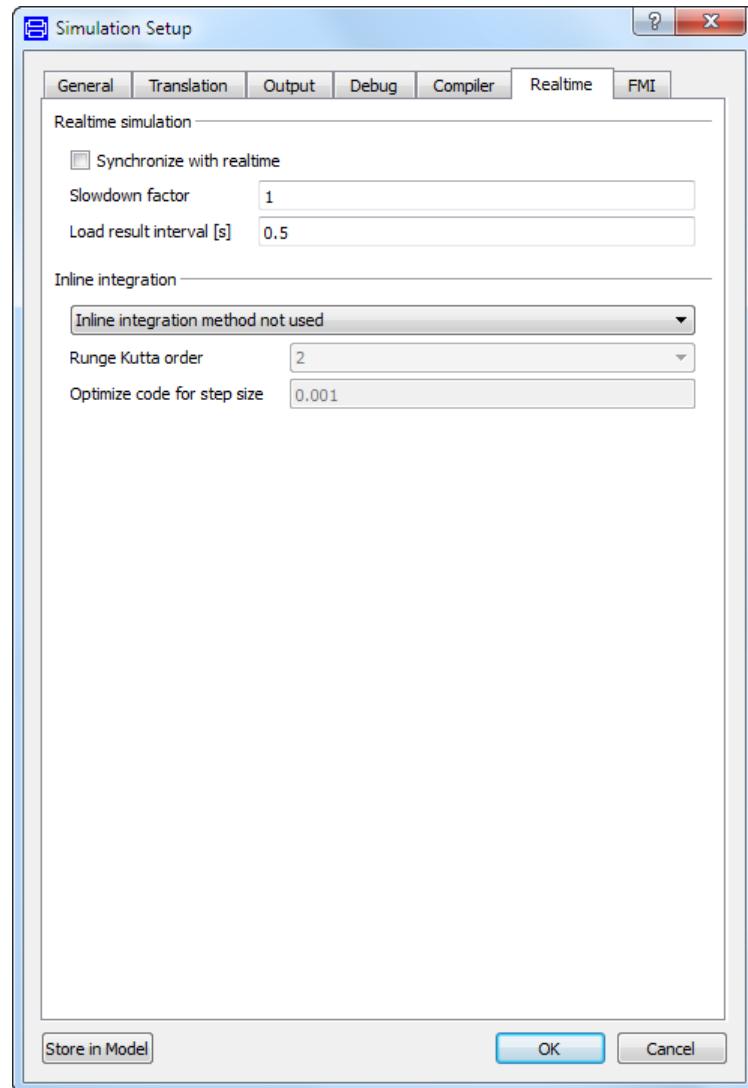


See chapter “Appendix – Installation”, section “Troubleshooting” for more information.

(The **Store in model** button only applies to the tabs **General** and **Output**.)

## 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.)

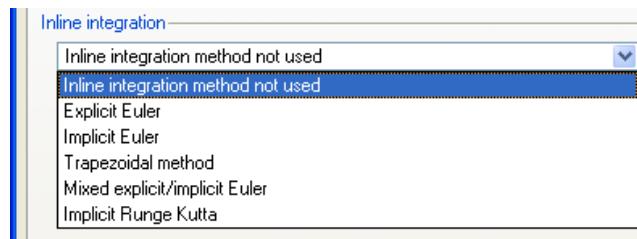
For hardware-in-the-loop simulation see the manual “Dymola User Manual Volume 2”, chapter “Other Simulation Environments”, section “Real-time simulation”.

**Slowdown factor** makes it possible to simulate in scaled real-time.

**Load result interval [s]** - how often the result file should be loaded. Online plots are updated this often.

**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.



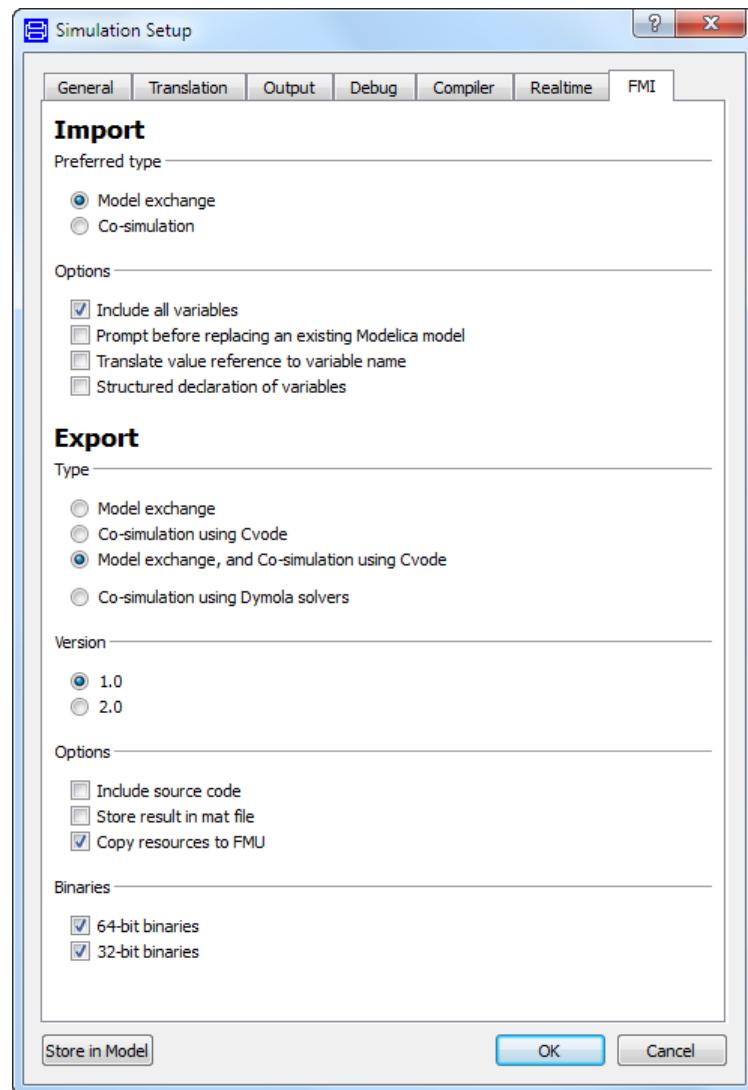
The order of the Implicit Runge-Kutta algorithm can be set between 2 and 4.

More information about this can be obtained in the section “Inline integration” starting at page 614.

(The **Store in model** button only applies to the tabs **General** and **Output**.)

## FMI tab

FMI tab of Simulation  
> Setup...



For more information about FMI in general, and the settings below, please see the manual “Dymola User Manual Volume 2”, chapter 6 “Other Simulation Environments”, section “FMI Support in Dymola”.

### Import part of tab

The settings in the import part of tab correspond to parameters in the built-in function `importFMU` (except the two last options). For more information about this function, see section “`importFMU`” on page 560. Note that this built-in function gives the possibility to select to which package the FMU should be imported.

For the 64-bit version of Dymola, it is required to set the flag Advanced.CompileWith64=2 when compiling applications (dymosim.exe) that contain imported 64-bit FMUs.

The preferred type of FMU to import can be set to **Model exchange** or **Co-simulation**. This setting is only relevant if the FMU to import supports both types. Otherwise this setting is silently ignored. (These settings correspond to integrate in importFMU, the default setting **Model exchange** corresponds to integrate=true.)

Four options are available:

**Include all variables** – corresponds to the parameter includeAllVariables in importFMU. If ticked (includeAllVariables=true) all variables are included, if unticked only inputs, outputs and parameters from the model description are included when generating the Modelica model.

**Prompt before replacing an existing Modelica model** – corresponds to the parameter promptReplacement in importFMU. If ticked (promptReplacement=true) prompting is done if Modelica models from previous import exists when importing.

**Translate value reference to variable name** – this option is not present in importFMU. If ticked, the imported FMU will contain a file with variable names. This is useful for debugging, however lowers the performance. With this option selected, value references will be mapped to variable names in the FMU log messages.

**Structured declaration of variables** – this option is not present in importFMU. If ticked, 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. Note. 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".

### Export part of tab

The settings in the export part of the tab correspond to parameters in the built-in function translateModelFMU. For more information about this function (including note on Linux), see section “translateModelFMU” on page 566.

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. Note – Co-simulation using Dymola solvers requires Binary Model Export license, and is currently only available on Windows.

Version of exported FMU can be selected as **1.0** or **2.0**. This setting corresponds to the input string fmiVersion in importFMU.

Three general options exist; note that the two first ones cannot be ticked simultaneously.

**Include source code** – corresponds to the parameter storeResult in translateModelFMU. If ticked (storeResult=true) source code is included, if unticked the source code is not included.

**Store result in mat file** - corresponds to the parameter `storeResult` in `translateModelFMU`. If ticked (`storeResult=true`) source code is included, if unticked the source code is not included.

**Copy resources to FMU** - external resources using the functions `ModelicaServices.ExternalReferences.loadResource` or `Modelica.Utilities.Files.loadResource` are by default copied to the FMU. 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.

Finally, the user can select whether 32- and/or 64-bit FMU should be generated. This option is not available in `translateModelFMU`. Note that even if the option 64-bit binaries are selected, no such binaries are created unless 64-bit compilation is enabled. In a 32-bit version of Dymola, this can be enabled by setting the flag `Advanced.CompileWith64=2`.

(The **Store in model** button only applies to the tabs **General** and **Output**.)

### Simulation > Visualize

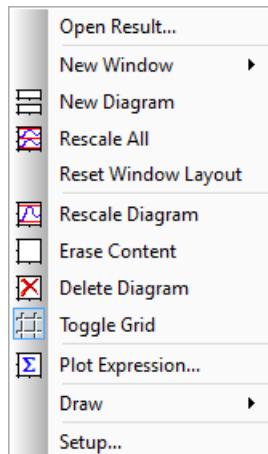


### Simulation > Show Log

**Show Log** opens the message window and shows a log for the last simulate command. More about this window can be read in the section “Message window” on page 382.

## 5.3.3 Main window: Plot menu

The plot menu includes commands to load results, create and edit plots and diagrams.



## Toolbar

The most common plot commands are available in the tool bar (the last one is only available in the toolbar; please see below, just before Plot > Setup...):



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 > Open Result...

Reads the result from a simulation made with Dymola to allow variables to be plotted. Note that the result of a simulation is automatically opened.

Note that you will obtain the same result by dragging a result/Matlab file into Dymola, or double-clicking it. To use double-clicking, .mat files must be associated to Dymola.

### Plot > New Window > New Plot Window



Creates a new active plot window which is initially empty.

### Plot > New Window > New Table Window



Creates a new active table window which is initially empty.

### Plot > New Diagram



Creates a new active diagram in the active plot window.

### Plot > Rescale All



Rescales all diagrams (reset them to initial zooming) in the active plot window

### Plot > Reset Window Layout

Rescales all diagrams in the active plot window and distribute them evenly in the window.

### Plot > Rescale Diagram



Rescales the current 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 > 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 > Delete Diagram**



Deletes the currently active diagram.

## **Plot > Toggle Grid**

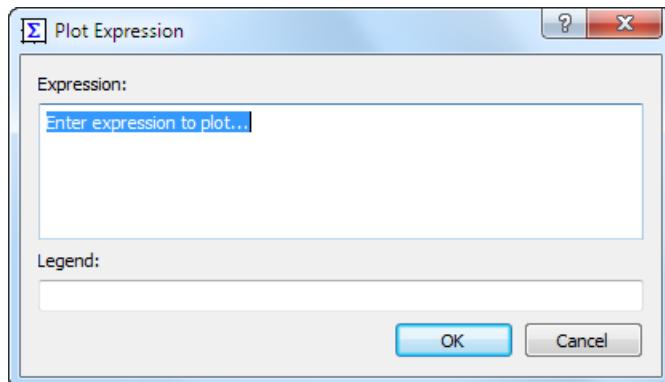


Enables grid lines in the diagram if they are not visible, otherwise disables grid lines.

## **Plot > Plot Expression...**



Displays a dialog for creating an expression that will be plotted if the expression is valid.



Please see section “Plotting general expressions” starting on page 429 for more information.

## **Plot > Draw > 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 436 for more information.

## **Command button Recent Windows**



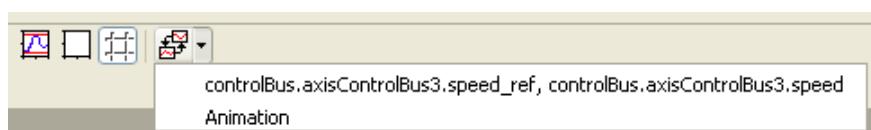
The command button **Recent Windows** has no corresponding command in the plot menu; it is only available in the Plot toolbar.

Clicking the button will toggle between the two last shown sub-windows.

Clicking the arrow displays a menu with all sub-windows available (plot, animation, diagram layer, visualizer). The menu 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 menu will bring it to the top.

This feature is valuable when working e.g. with a number of plot windows.

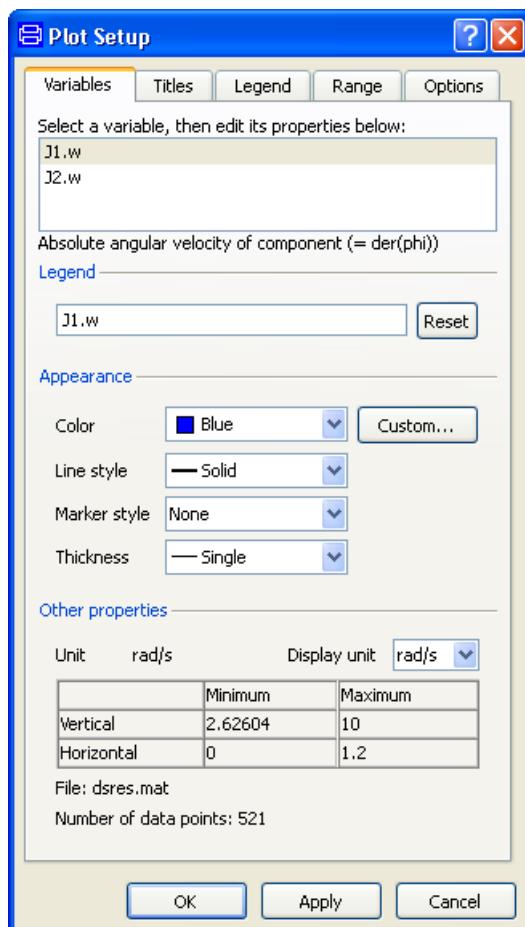


## Plot > Setup...

The settings of a plot window can be saved using the command **File > Generate Script...**, ticking **Plot setup**. See section “File > Generate Script...” on page 464.

### 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.

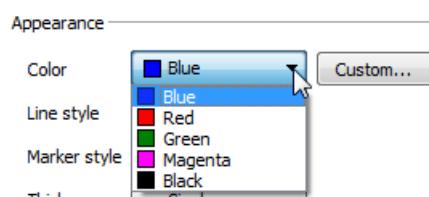
The **Legend** group contains the current description of the selected variable. The legend may be written using extended character set (UNICODE). Edit the text and press **Apply** to change the legend. If the text is empty, no legend is shown for this variable. Pressing the **Reset** button fills the edit field with the default legend. User-defined legends are shown in section “Plot window” on page 374.

The **Appearance** group enables changing the color, line/marker style and thickness of the selected variable.

### Color

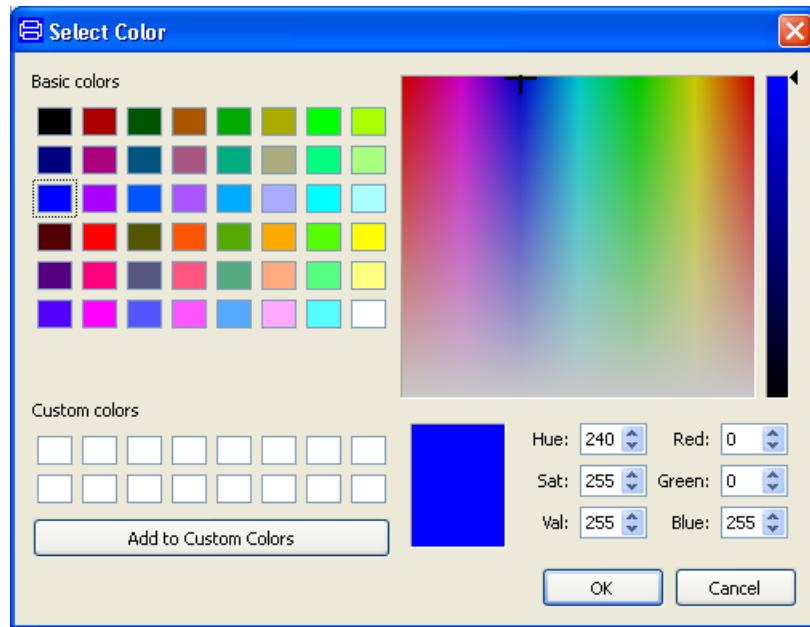
The default drop-down alternatives are:

#### Color of a variable.



Custom colors can be set by clicking the **Custom...** button. A color palette is displayed:

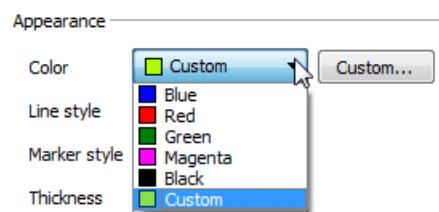
#### The color palette.



The handling of the palette is described in the chapter “Developing a model”, section “Graphical object menus: Line and fill style”.

Selecting a color in this palette and then clicking **OK** will display this selection as a custom color last in the drop-down list, e.g.:

**Custom color of a variable.**

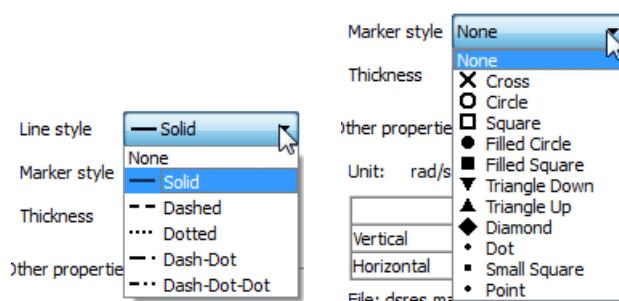


Only one custom color per variable is displayed. Note that any color selection in the palette will define this color as custom color for the variable; it does not have to correspond to a custom color in the palette.

### 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.**

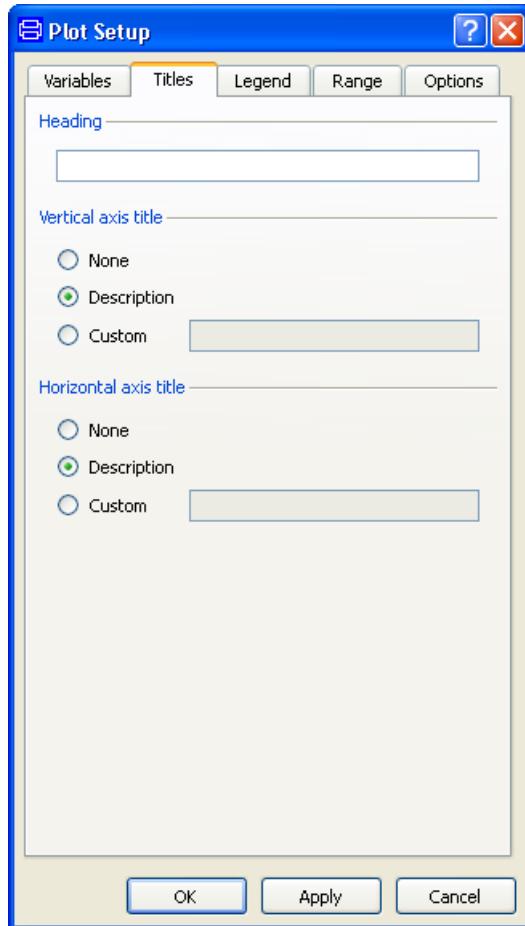


The thickness alternatives are the same as for lines when drawing primitives.

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” on page 408.

## **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.

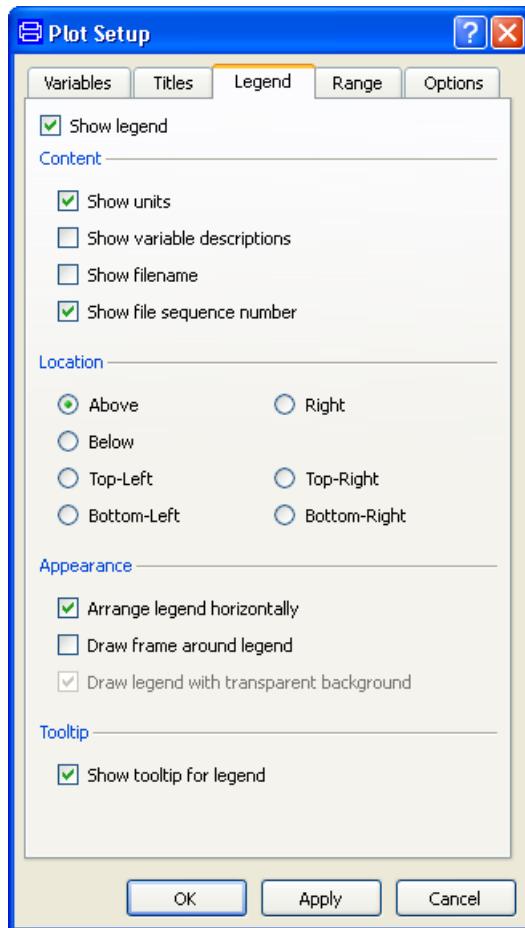
**Horizontal axis title** specifies the title for the horizontal axis of the diagram.

None                No title is shown by the horizontal axis.

Description	The title is extracted from the independent variable, if other than time.
Custom	The title is specified by the user in the input field.

### 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 appearance of the variables is set as follows:

- **Show 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).
- **Show variables 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.

- **Show filename** adds the file name (without extension .mat) in the legend, as // file name, if the plot contains variables from more than one file.
- **Show file sequence number** adds the file sequence number in the legend of the variable, as // sequence number, if the plot contains variables from more than one simulation.

The layout of the legend is controlled by these settings:

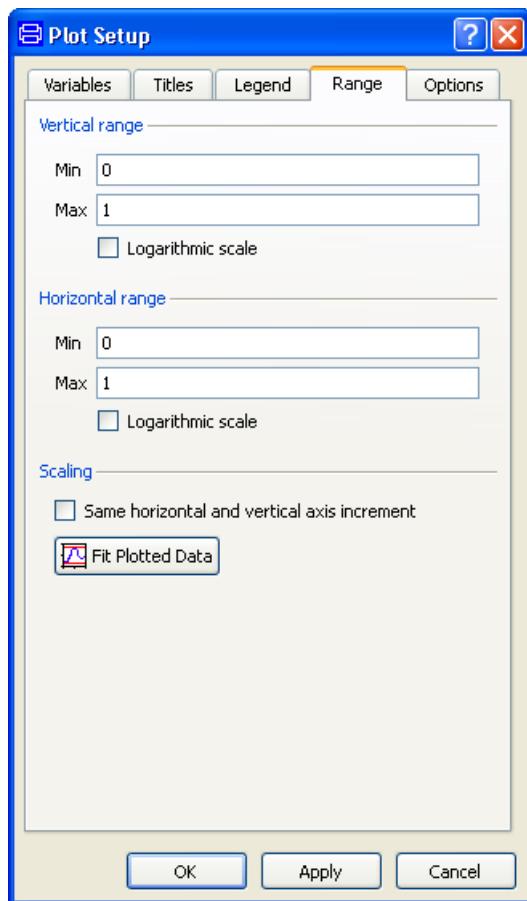
- **Location** of the legend is either outside of the diagram (above, below or to the right of the diagram), or close to one of the corners inside the diagram. See section “Plot window” on page 374 for examples of diagrams with different locations of the legend.
- **Arrange legend horizontally** lists 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 is still present.

## Range tab

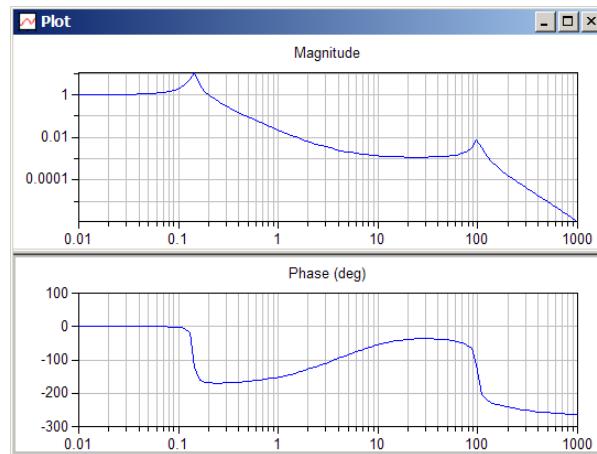
Range tab of the plot window setup.



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:

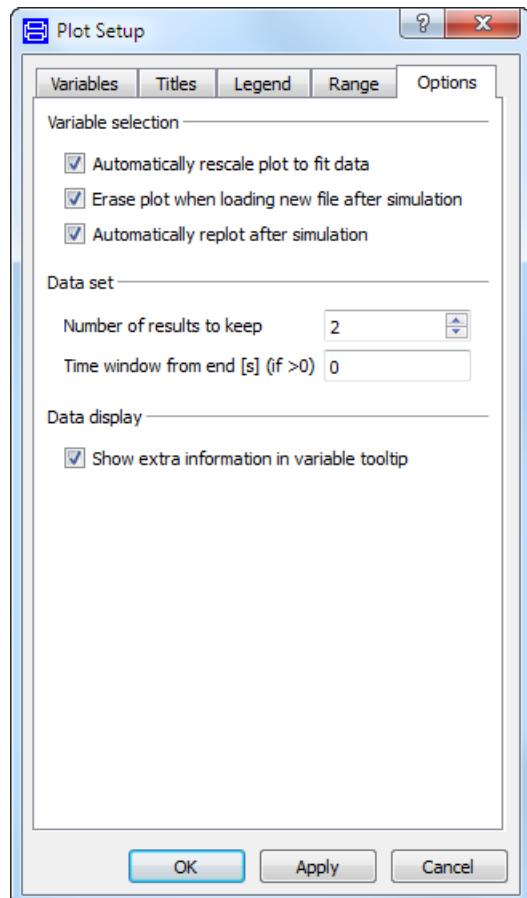
- **Same horizontal and vertical axis increment** may be used when the relative scale of vertical and horizontal axes is important.
- **Fit Plotted Data** rescales the diagram to fit plotted variables. This is same as pressing the Rescale button and then reads the default range.

**Logarithmic scale used in Bode plot.**



### Options tab

**Options tab of the plot window setup.**

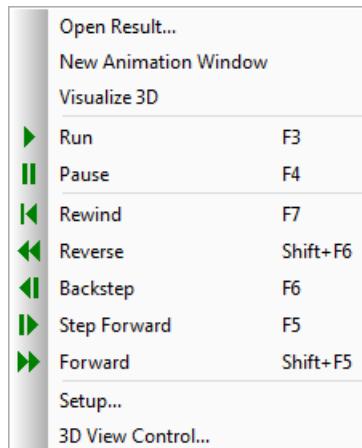


The Options tab sets up options controlling plotted variables:

- **Automatically 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.
- **Number of results to keep when simulating**. Manually loaded results are not included, nor 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 398 for more information about this.
- **Time window from end** defines the size of the time window during online plotting.
- **Show extra information in variable tooltip**. Unticking this setting, the tooltip will only contain the variable name, value and time; local min, local max and slope are not displayed.

### 5.3.4 Main window: Animation menu

The animation menu contains commands for animation of models.



#### Toolbar

The toolbar can be divided in two parts, one part that corresponds to the “player” commands in the Animation menu and one part that does not correspond to the menu.



### Time



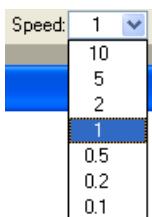
The time shows the progress of the simulation in time.

### Progress bar



The progress bar shows the progress of the simulation.

### Speed



The speed enables setting different speed. A higher value means that the time runs faster in the animation.

### **Animation > Open Result...**

Reads the result file from a simulation made with Dymola. The 3D-view is replaced by the rendering of the new result file. The result file is by default automatically opened after simulation and if it contains animation data the animation window is opened automatically.

### **Animation > New Animation Window**

Creates a new animation window.

### **Animation > Visualize 3D**

Creates a new visualizer window.

### **Animation > Run (function key F3)**



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 > Setup...” below) the animation automatically restarts with the first frame.

### **Animation > Pause (function key F4)**



Stops a running animation. This command has no effect if no animation is running.

### **Animation > Rewind (function key F7)**



Rewinds the animation to the first frame. A running animation is stopped.

### **Animation > Reverse (function key Shift+F6)**



Moves the animation backward at high speed.

### **Animation > Backstep (function key F6)**



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 (function key F5)**



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 > Forward (function key Shift+F5)**



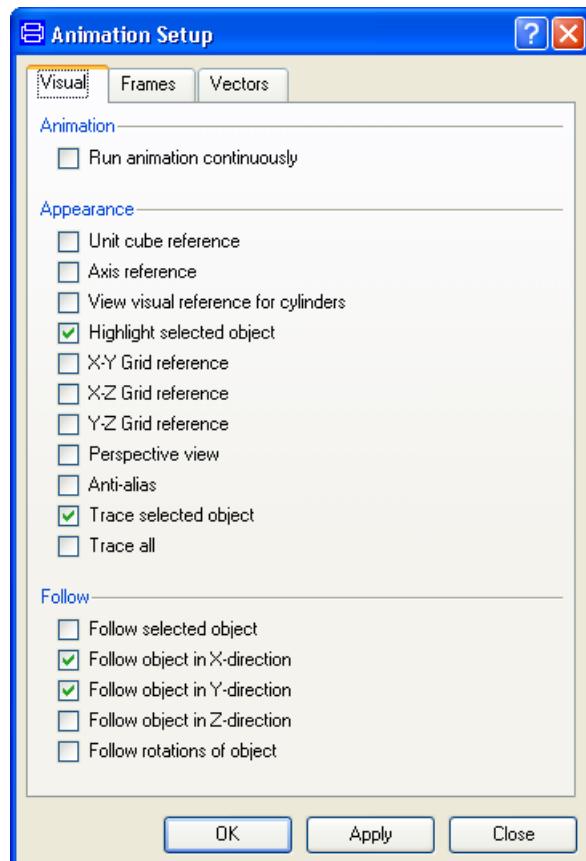
Moves the animation forward at high speed.

## **Animation > Setup...**

Changes several options that control animation and viewing of animated objects.

### **Visual Tab**

**The animation setup dialog.**



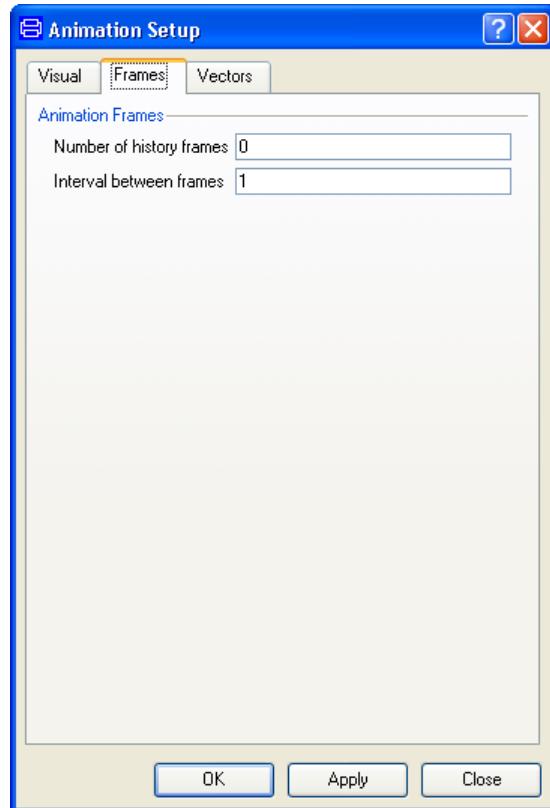
The Visual tab contains:

- **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

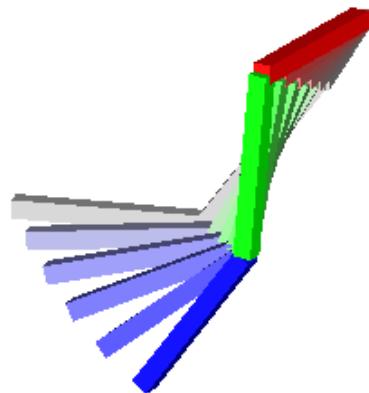
### Animation frames tab.



The Frames tab sets a number of attributes related to history frames. History frames gradually fade to the grey color of the background.

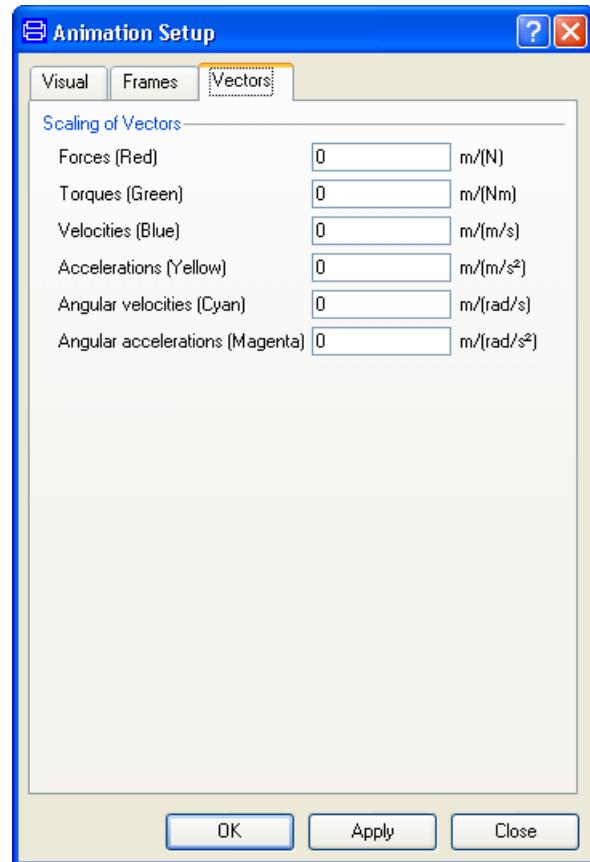
- **Number of history frames** introduces motion blur showing several frames. Zero will turn off all history frames. A large number of history frames may slow down the animation considerably. If the number of history frames times the interval between history frames is greater than the total numbers of frames, all frames are displayed and there is no performance penalty.
- **Interval between history frames.** With the default frame interval (1), frames  $1, 2, \dots, n$  older than the leading edge frame are shown. With a frame interval of  $m$ , frames  $m, 2m, \dots, 2mn$  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.**

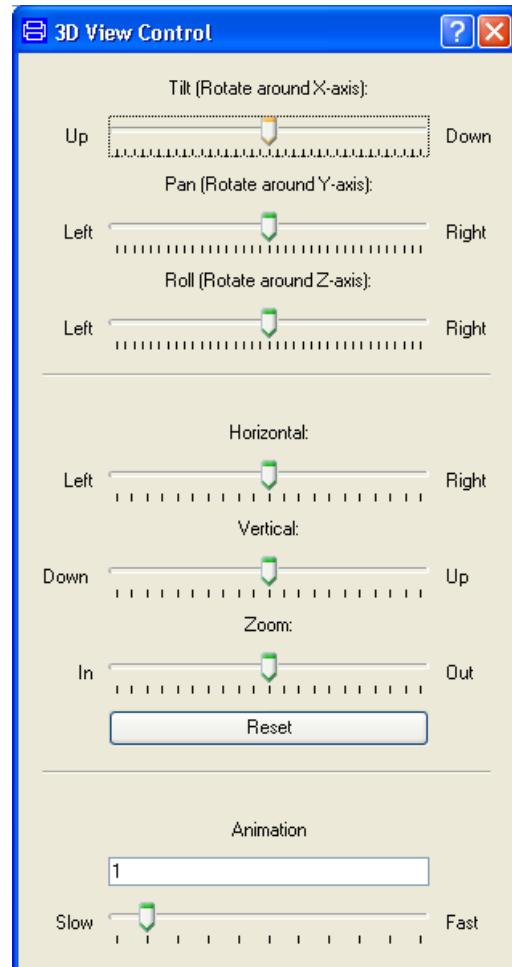


**Vector tab** is for controlling the scaling of vector visualizers (found in Program Files (x86)\Dymola 2015 FD01\Modelica\VisualVector.mo). The units indicate that if you for example set the scale factor for forces to 1e-3 a force of 100N will be represented by an arrow 0.1m (=1e-3 m/N\*100N) long in scale of the diagram.

### Animation > 3D View Control...

This command creates a window with controls for 3D viewing.

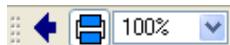
#### 3D view control.



### 5.3.5 Main window: Commands menu

Please see corresponding section in the chapter “Developing a model”.

### 5.3.6 Main window: Window menu



Please see corresponding section in the chapter “Developing a model”. Please however note that not all possibilities in **Window > View** exist in Simulation mode, only **Window > View > Previous** and **Window > View > Diagram**. The toolbar in Simulation mode is shown to the left.

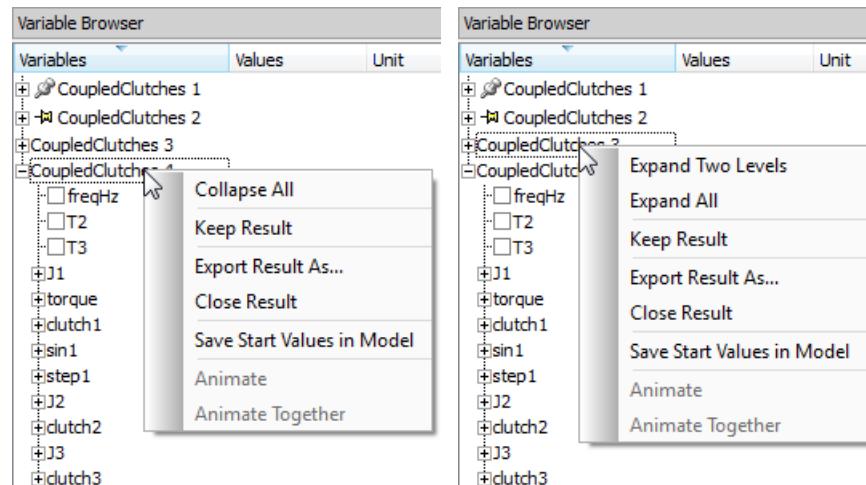
### 5.3.7 Main window: Help menu

Please see corresponding section in the chapter “Developing a model”.

### 5.3.8 Context menu: Variable browser – nodes

The variable browser is introduced in “Variable browser” on page 372. See also “Variable browser interaction” starting on page 392.

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.

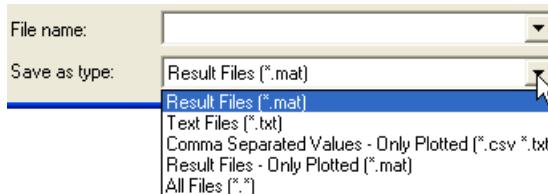
- Clicking on a + opens one more level of the tree. The **Expand Two Levels** operation opens two levels, which makes the model structure clearer without creating a huge browse tree.
- The **Expand All** operation opens every sub-node in the browse tree. Large sub-trees may become very large and hard to navigate.

- Clicking on – 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 five 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 398.
- The **Export Result As...** operation allows the user to export the result file in several different file formats. It is possible to either export the whole result file, or just those signals which are currently plotted.

The **Export Result As...** operation can export data in several different formats.



First, data can be exported as Dymola result files (.mat format), as text, or as comma separated values (suitable for Microsoft Excel and other applications). 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: For use in e.g. Excel. **Note:** some versions of Microsoft Excel uses the Regional Setting of List Separator when reading CSV files, please set this to ','

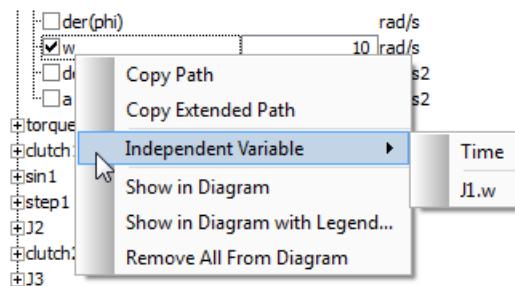
The file types marked with “Only Plotted” will only store *only* the variables which are currently plotted.

- The **Close Result** operation will delete the result file from Dymola and free the occupied storage space.
- The **Save Start Values in Model** will save modified initial conditions and parameter values entered in the variable browser to the model. See section “Save start values in model” on page 395 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.
- 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.

### 5.3.9 Context menu: Variable browser – signals

The variable browser is introduced in “Variable browser” on page 372. See also “Variable browser interaction” starting on page 392.

If the context menu is activated when the cursor is beside a signal (parameter or variable) the context menu will look the following.



**Copy Path** will copy the path of the signal to the clipboard.

**Copy Extended Path** will copy the extended path of the signal to the clipboard. The extended path is the signal path with result file name and sequence name (in brackets) included.

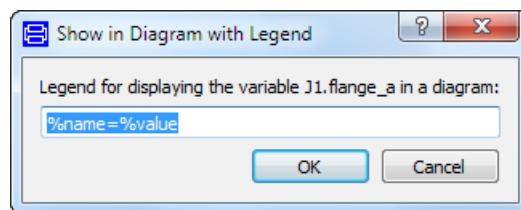
**Independent Variable** enables selection of what should be the value of the horizontal axis in the plot window. Default is usually Time.

**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 402 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:



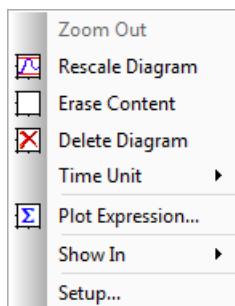
The string %name is expanded as the variable's name; the string %value is expanded as the variable's value.

**Remove All From Diagram** will erase all displayed units.

### 5.3.10 Context menu: Plot window

The plot window is introduced in “Plot window” on page 374. Please also see “Plot window interaction” starting on page 405.

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.



**Rescale Diagram** will reset the zooming to the initial value for all signals in the diagram.

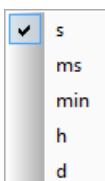


**Erase Content** will erase the content (curves and text objects, if any) in the diagram.



**Delete Diagram** will delete the currently active diagram.

**Time Unit** is used to select the time unit on the horizontal axis. The selections are the ones available as display units for time. An example is:

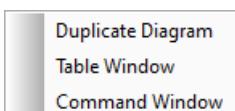


The default is always s (seconds). If another time unit is selected, this will be displayed in the legend of the horizontal axis.



**Plot Expression...** will have the same effect as the command **Plot > Plot Expression...**. Please see this command in section “Plot > Plot Expression...” on page 488.

**Show In** has the following options:



**Duplicate Diagram** will copy the selected diagram, with content, to a new plot window.

**Table Window** will copy the curves in the selected diagram to a new table window.

**Command Window** will insert the plot and corresponding command in the command window. Please see section “Insert plot and its corresponding command in command window” on page 438 for more information.

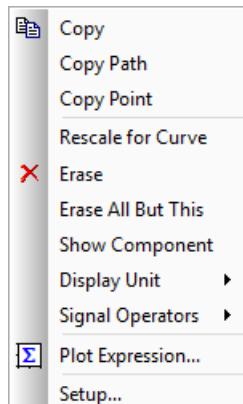
**Setup...** will have the same effect as the command **Plot > Setup...** Please see this command in section “Plot > Setup...” starting on page 489.

**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 405 for the full picture.

### 5.3.11 Context menu: Plot window – curve and legend

The plot window is introduced in “Plot window” on page 374. Please also see “Plot window interaction” starting on page 405.

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 current 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.

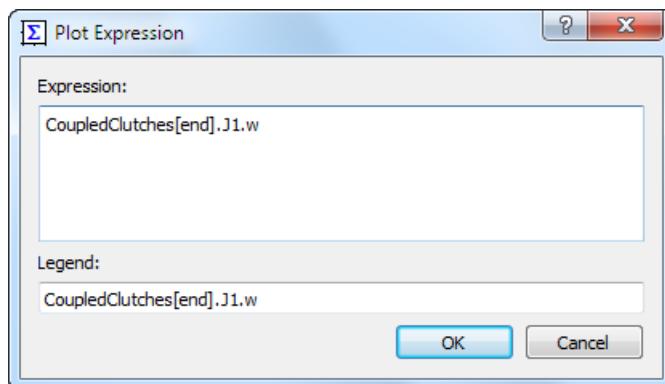
**Show Component** will display the component that contains the variable corresponding to the selected curve. The component will be shown selected in a window displaying the diagram layer, with relevant zooming. If a window containing the diagram layer is not open, such a window will be opened automatically.

**Display Unit** makes it possible to change the display unit of signals.

**Signal Operators** makes it possible to add signal operators to the curve. Please see section “Displaying signal operators” starting on page 421 for information about this.

**Parameter Labels** (only available for parametric curves) will display parameter labels along the curves.

**Plot Expression...** will display a dialog for creating a plot expression, prepopulated with the selected curve; e.g.

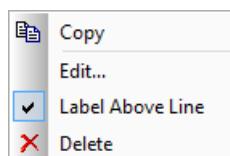


Please see section “Plotting general expressions” starting on page 429 for more information.

**Setup...** will have the same effect as the command **Plot > Setup...**. Please see this command in section “Plot > Setup...” starting on page 489.

### 5.3.12 Context menu: Plot window – signal operators

Signals operators for curves are introduced in “Displaying signal operators” starting on page 421. 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.

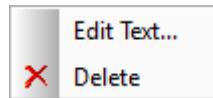
**Edit...** will display the dialog for the selected signal operator; that makes it possible to change it.

**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.

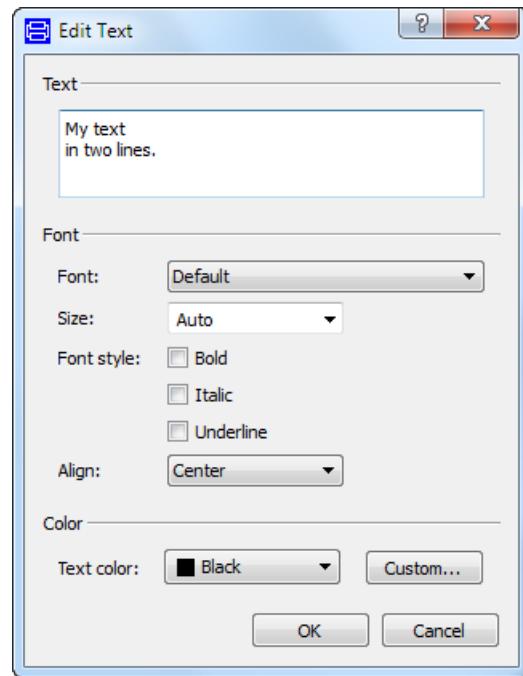
**Delete** will delete the signal operator.

### 5.3.13 Context menu: Plot window – text objects

Texts inserted in plots are introduced in “Insert and edit text objects in the plot” on page 436. 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:

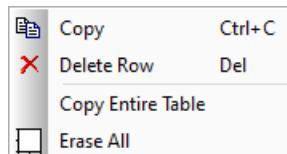


**Delete** will remove the text object from the plot.

For more information about this menu, please see “Texts” in previous chapter, section “Basic model editing”, sub-section “Creating graphical objects”.

### 5.3.14 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 433. Right-clicking in such a table displays a context menu:



The following alternatives can be available:

**Copy** will copy the selected content to the clipboard.

**Delete Row** will delete selected row.

**Copy Entire Table** will copy the entire table to clipboard.

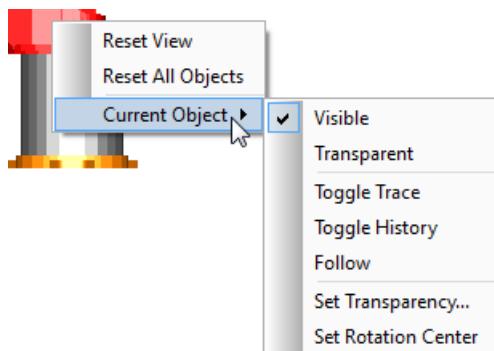
**Erase All** will erase the table.

### 5.3.15 Context menu: Animation window

The animation window is introduced in “Animation window” on page 376. Please also see “Animation window interaction” starting on page 444.

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.



**Reset View** resets all manipulation of the view (panning, rotation and zooming), however not any selection in **Current Object** in this menu.

**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**.

**Current Object** gives the following subentries:

**Visible** is checked by default. Unchecking it makes the object invisible and impossible to select.

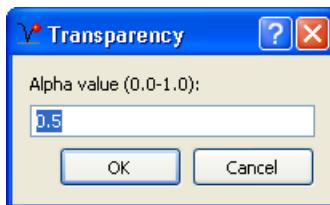
**Transparent** makes the object transparent.

**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 501.

**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 499.

**Set Transparency...** will display the following menu for setting the transparency:



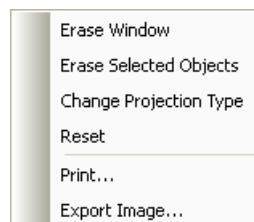
An alpha value of 1.0 means opaque, a value of 0 means invisible. An invisible object is not possible to select.

**Set Rotation Center** sets the rotation center to the selected component.

### 5.3.16 Context menu: Visualizer window

The visualizer window is introduced in “Visualizer window” on page 377.

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.

**Change projection type** will toggle between orthogonal projection (angles are preserved) and perspective projection.

**Reset** will reset the objects in the window.

**Print...** will print the window.

**Export Image...** saves an image of the contents of the visualizer window (without window borders) as a .png, .xpm or .jpg file. The user is prompted for a file name. (Please note that this command gives more possibilities – more file formats - than using the command **File > Export... > Image...)**

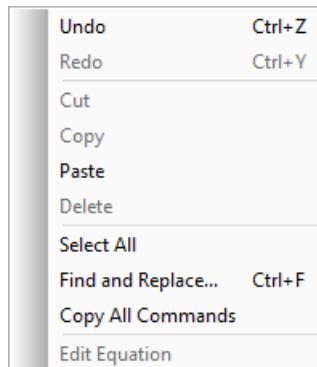
The image is identical to the image shown in the window, so the size and representation can be changed by first resizing the window.

Exported images are included in the command log.

### 5.3.17 Context menu: Command window – Command log pane

The command log pane of the command window is introduced in “The command log pane” on page 379.

By right-clicking in the command log pane of the command window, the following context menu will be shown:



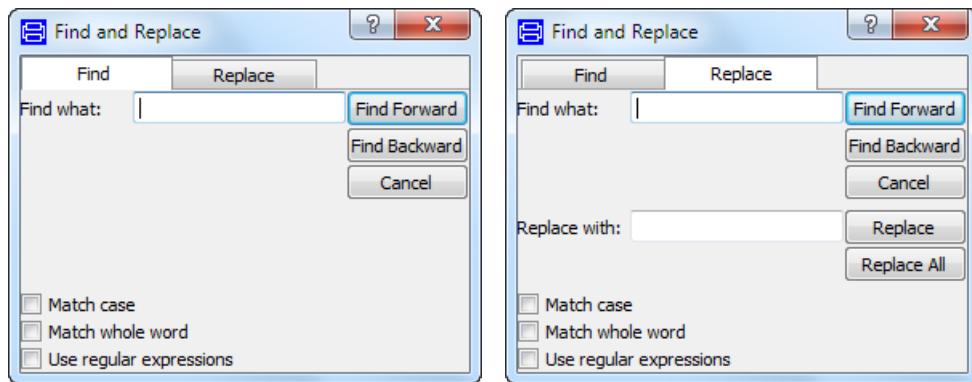
**Undo**, **Redo** undoes and redoes the last editing operation in the text editor.

**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...** – search/replace functionality for a specific text. The menu looks like:



The **Find/Replace** operation will automatically scroll the window to make the found text visible.

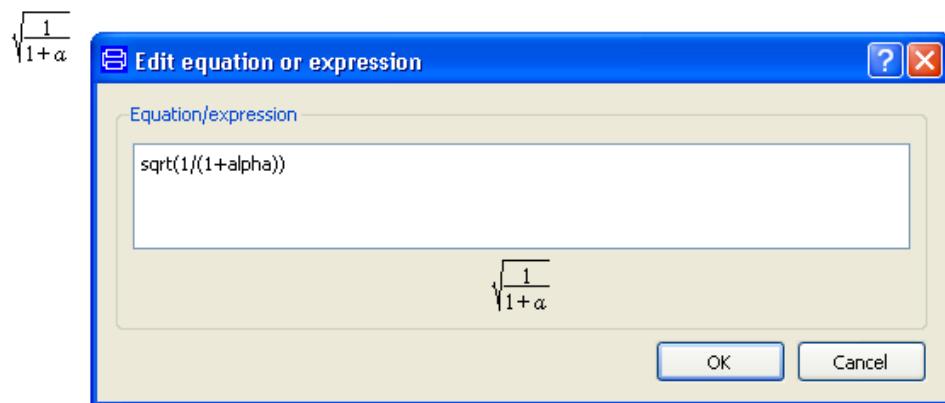
Regular expressions can be used in the search if this checkbox is checked. Special symbols in the regular expression are

*	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.

**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.

**Edit equation** is only available in the context menu of the documentation editor of the command log pane. If an equation or expression created using the button **Insert equation or expression** is selected, this entry will pop an editor for the equation/expression.

**Edit equation on a square root expression.**

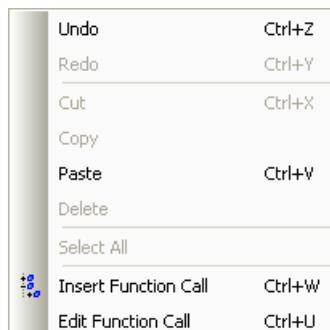


The result of the editing is shown below the editing pane. If an equation/expression is not concluded, the text Incomplete equation/expression is displayed. Clicking **OK** the edited equation/expression will replace the previously selected one.

### 5.3.18 Context menu: Command window – Command input line

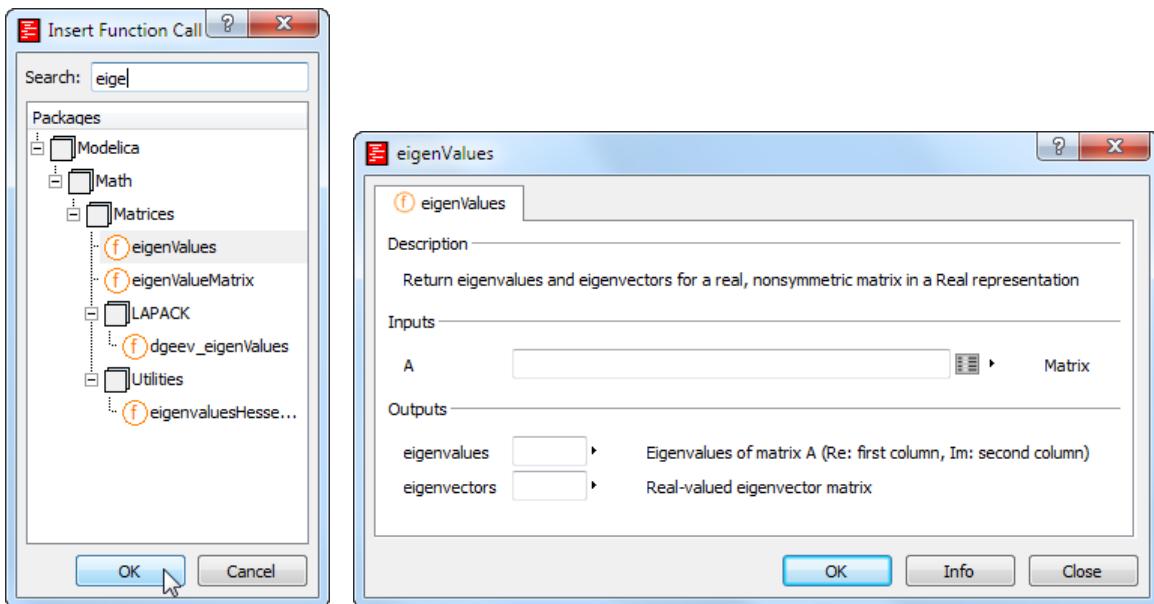
The command input line of the command window is introduced in “The command input line” on page 380.

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 two, please see the above section.

**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):



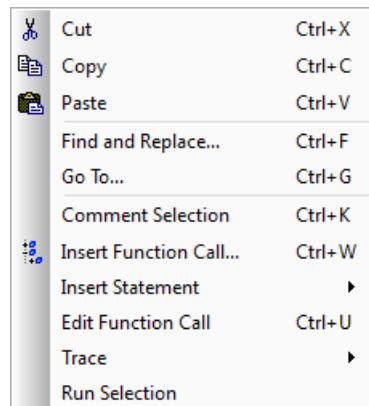
For more information how to use this function, please see section “Basic facts about scripting in Dymola” starting on page 529.

**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.

### 5.3.19 Context menu: Script editor

The Dymola script editor was introduced in “Script editor” on page 380.

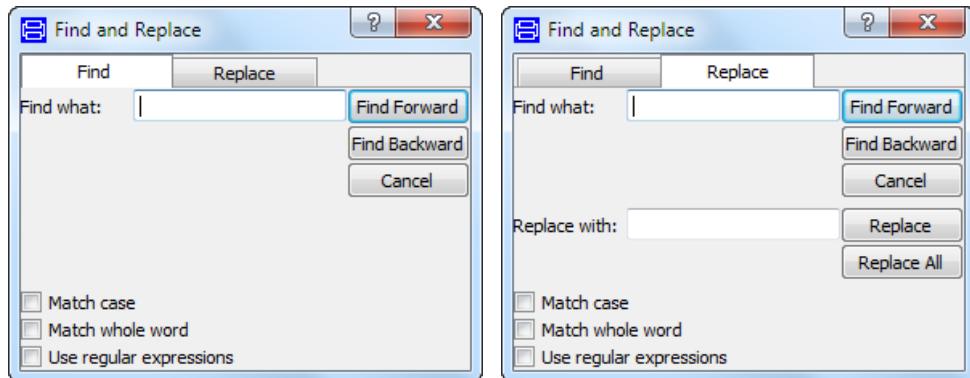
Right-clicking in the script editor displays a context menu with the following choices:



**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. Enter the text you search for and press **OK**.

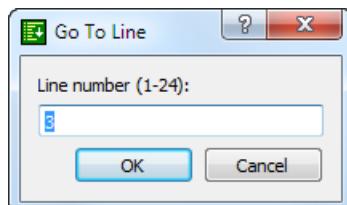
The **Find and Replace** operation will automatically scroll the window to make the found text visible. **Find/Replace** in Modelica text gives warning if search string is not found.



Regular expressions can be used in the search if this checkbox is checked. Special symbols in the regular expression are

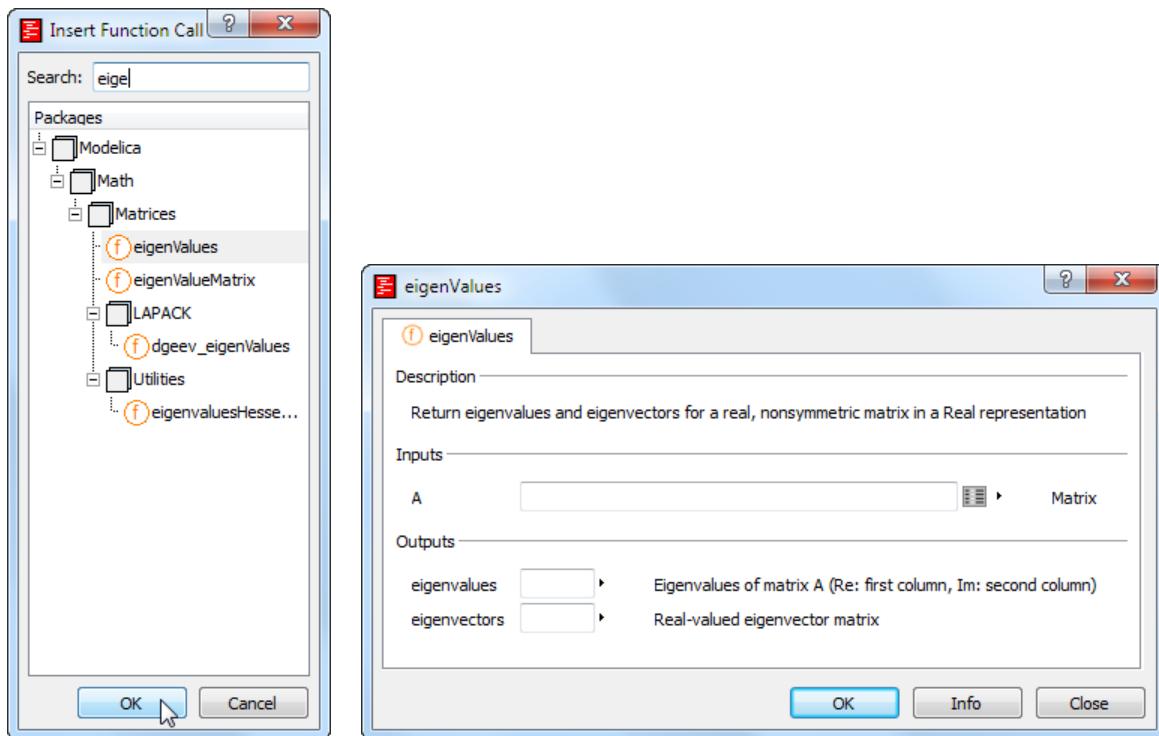
*	Match everything.
?	Match any single character.
{ab}	Match characters a or b.
{a-z}	Match characters a through z.
{^ab}	Match any characters except a and b.
E+	Match one or more occurrence of E.
(ab cd)	Match ab or cd.
\d	Match any digit.
\w	Match any digit or letter.
^	Match from start.
\$	Match at end.

**Go To...** sets the cursor at specified line and scrolls window if necessary. The number of the current line is shown as default.



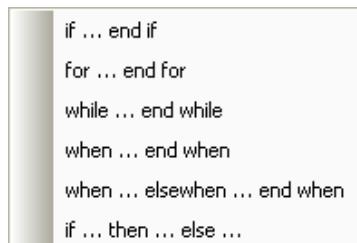
**Comment Selection** can be used to comment out selected rows.

**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):



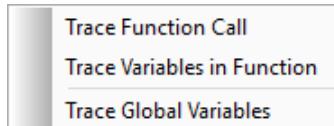
For more information how to use this function, please see “Basic facts about scripting in Dymola” on page 529.

**Insert Statement** presents a submenu with common Modelica constructs. This makes it easier to enter Modelica code.



**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.

**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 553.

**Run Selection** executes the selected part of the displayed script as was it a separate script.

### 5.3.20 Context menu: Message window

The message window is introduced in “Message window” on page 382.

Right-clicking in the message window presents a menu with the following choices.



The image to the left above shows alternatives of all tabs except the Translation tab, the image to the right shows alternatives of Translation tab.

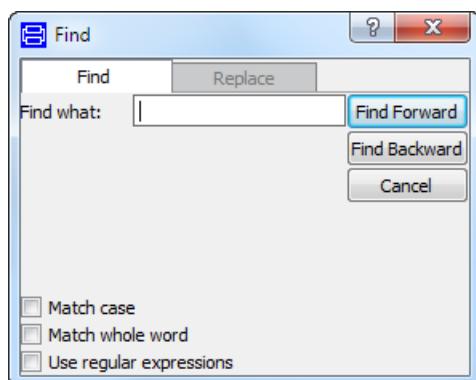
**Copy** – copies the selected text to the clipboard.

**Copy Link Location** – copies selected link to clipboard (icons cannot be copied).

**Select All** – selects all text.

**Expand All** – expands all nodes in the messages tree structure.

**Find...** – search functionality for a specific text. The menu looks like:



Concerning use of regular expressions, please see section “Context menu: Command window – Command log pane” starting on page 513.

The **Find** operation will automatically scroll the window to make the found text visible.  
(The **Replace** tab is dimmed and cannot be used.)

---

## 5.4 Dynamic Model Simulator

### 5.4.1 Overview

#### What is Dymosim?

Dymosim stands for **Dynamic model simulator** 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 or OPC server with real-time capability. For more information, please see “Dymola User Manual Volume 2”, chapter “Other 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); dymosim.exe compiled as 32-bit, dymosim.dll, dymosim with DDE server, and FMUs compiled as 64-bit.
- 1 All applications above compiled as 32-bit.
- 2 All applications above compiled as 64-bit.

Dymosim as OPC server can presently only be compiled as 32-bit, setting `Advanced.CompileWith64=2` is not supported when compiling dymosim as OPC server.

### 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 470 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 389 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 manual “Dymola User Manual Volume 2”, chapter “Other simulation environments”.)

### **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`.

### **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:

<code>.fname</code>	<i>the file name</i>
<code>.pname</code>	<i>the path name</i>
<code>.nnames</code>	<i>the number of variable names</i>
<code>.ndatamat</code>	<i>the number of data matrices</i>
<code>.name</code>	<i>the names of the variables (each row a name)</i>
<code>.description</code>	<i>the description of the variables (each row a description)</i>
<code>.dataInfo</code>	<i>The dataInfo-array</i>
<code>.data_#</code>	<i>The data matrices. # ranges from 1 to .datamat</i>

The command “`d=dymload`” loads data from `dsres.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 (x86)\Dymola 2015 FD01\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 527 (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”.

If the file `dsu.txt` is present when Dymosim is started, Dymosim reads the trajectories of input signals from this file. The file can either be in ASCII format or can be a Matlab mat-file in the format of the result file `dsres.mat`, i.e., the result file can be at once used as input file of another model. 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).

### 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  $\mathbf{x}(t)$  should be computed. The tolerance must be greater than zero and is used in a local error test for each component<sup>7</sup> of the state vector  $x_i$ , roughly requiring at each step that

$$|\text{local error}| < \text{tolrel} * |x_i| + \text{tolabs}$$

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  $\mathbf{x}$  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  $\mathbf{x}$  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

---

<sup>7</sup> 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.

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 a `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  $\mathbf{x}$  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 parameter `algorithm(hmax)` in the input file, additionally it is possible to turn off *dense output*.

### Variable order

Integration algorithms approximate the solution  $\mathbf{x}(t)$  internally by a polynomial of order  $k_{\text{ord}}$ . 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  $\mathbf{x}(t)$  can be differentiated at least  $k_{\text{ord}}+1$  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  $\mathbf{x}(t)$  (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<sup>8</sup>:

---

<sup>8</sup> (The algorithms from Radau IIa and onwards in this table are sometimes referred to as “Godess solvers” (from Generic Ordinary Differential Equations Solver Systems).)

Algorithm	Order	Stiff	A-stable	Dense output	Root finder	Method
LSODAR	1-12, 1-5	Both	No	Yes	Yes	Multiple step/ Adams methods
DASSL	1-5	Yes	No	Yes	Yes	Multiple 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 <sup>9</sup>	Variable	Yes	No	Yes	Yes	Multi-step

### Fixed step size integration algorithms

There are also four different fixed step size integration algorithms, intended for real-time simulations:

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.

---

<sup>9</sup> For further details about the SUNDIALS CVODE solver, see <https://computation.llnl.gov/casc/sundials/main.html>.

## 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 (x86)\Dymola 2015 FD01\mfiles, Program Files (x86)\Dymola 2015 FD01\mfiles\dymtools and Program Files (x86)\Dymola 2015 FD01\mfiles\traj. (More information about using Dymola and Matlab is given in the manual “Dymola User Manual Volume 2”, chapter “Other 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:  <code>s = [ 0 0 0 1 1 2 2 4 4 3 9 6];</code>
N	A string matrix, where row “j” of “n” is the name of the signal of column “j” of “s”. Example:  <code>n = [ 'time' 't*t ' '2*t ']</code>

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 (x86)\Dymola 2015 FD01\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.

<i>Functions on trajectories (data + names):</i>	
dymbrowse	interactively browse, plot and compare Dymola simulation results.
Dymosim	perform time simulation of a Dymola model.

<i>Other functions:</i>	
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

<i>Functions on trajectories (data + names):</i>	
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.

<i>Functions on trajectory name matrix:</i>	
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.

<i>Other functions:</i>	
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:

<i>-s</i>	simulate (default).
<i>-h</i>	list command summary. If the executable was generated using binary model export or if it will require a runtime license will be displayed (after the command summary), as will any licensed libraries used.
<i>-d file</i>	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>-i</i>	generate Dymosim input file (containing e.g. the names and default / initial values of all parameters and state variables).
<i>-ib</i>	generate Dymosim input file in (Matlab) binary format (mat-file).
<i>-p</i>	precede Dymosim output lines by program name (for real-time Dymosim, in order to identify the process which prints the line).
<i>-v</i>	verify input, i.e. store complete input in file “dsinver.txt”.
<i>-c cmd</i>	execute command <i>cmd</i> 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.

The script facility makes it possible to e.g. load model libraries, set parameters, set start values, simulate and plot variables.

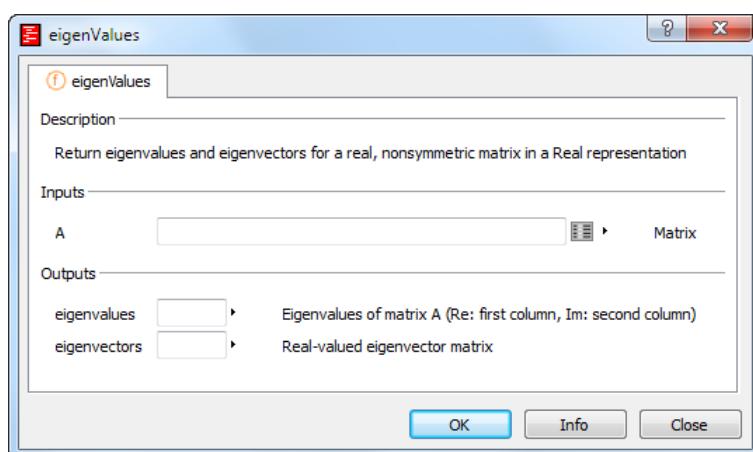
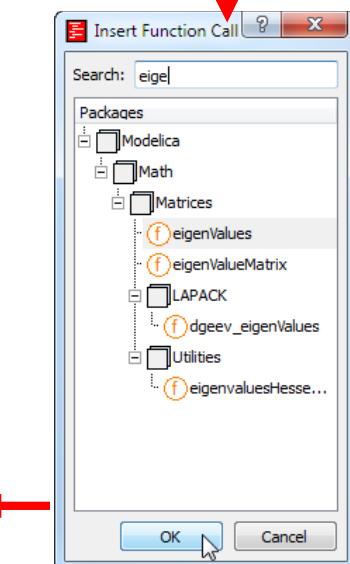
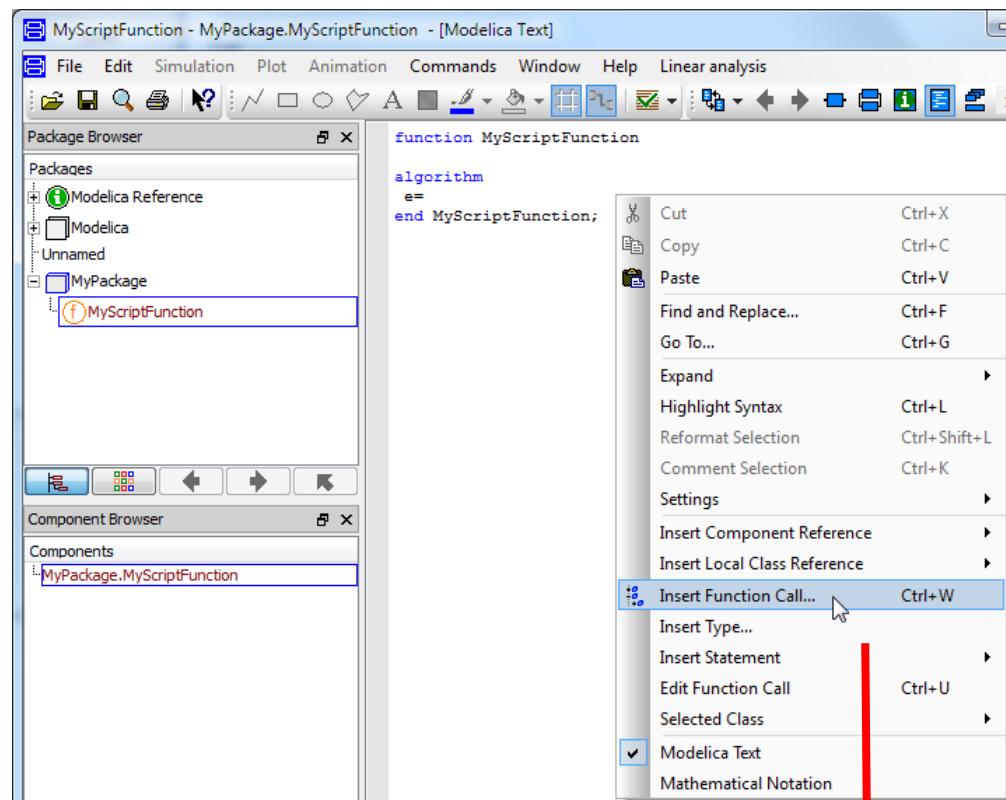
Scripting can be seen as a way of storing a successful (or promising) sequence of interactive events. The user experiments with certain commands and settings and suddenly finds something worth saving for later reuse (and development).

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`.



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 557.

## 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 538.

### 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 544.

### 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 switches. 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 557.

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.

#### 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 473.

### 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"});
```

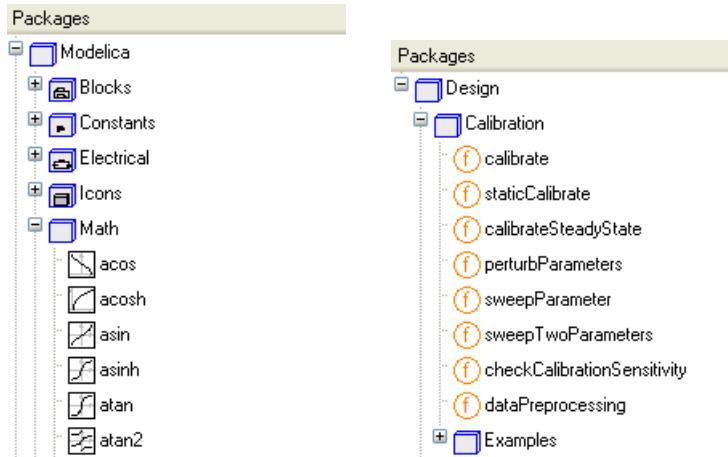
## 5.5.4 What is a Modelica function?

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.

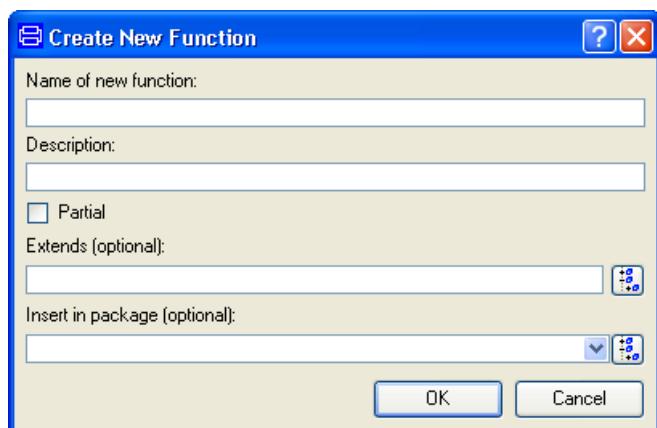
**Examples of functions  
from Modelica.Math  
and Design.Calibration.**



### 5.5.5 Creating a Modelica function

A Modelica function is created in Modeling mode used 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 – Modeling mode”.)



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 **Commands > Add Command**. This is described in the chapter “Developing a model”, sub-section “Editor command reference – Modeling mode”, sub-section “Main window: Commands menu”.

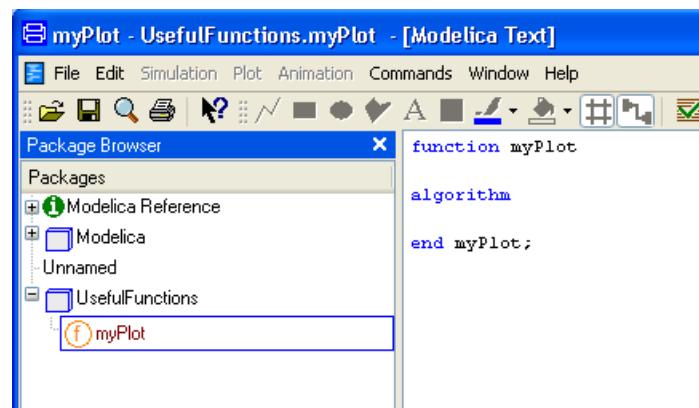
If you for some reason change the location of a model containing commands corresponding to functions called, please check that the commands still work after the change of location.

## 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. E.g., if a function `myPlot` has been created in the package `UsefulFunctions`:

Example of newly created 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 529.

## 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 557.

### Handling of nested function calls

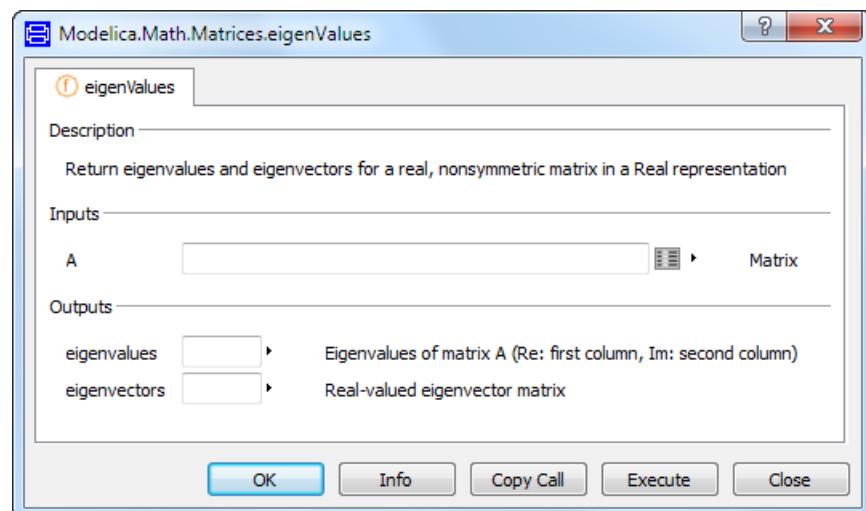
Nested function calls can be created. The main advantage with nested functions is the extensive information exchange between the primary (“calling”) function and the nested (“called”) function. The nested function can use and alter its input values, and can also create new ones. After the call, the primary 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

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 **File > 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.

Scripts files can be handled in both Modeling and Simulation mode.

#### 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:

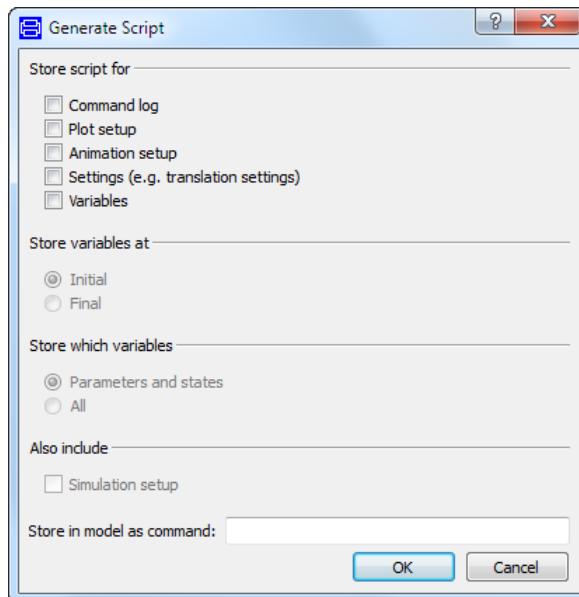
- To create them from scratch using the Dymola Script Editor. See section “Creating, editing and saving scripts using the Dymola script editor” starting on page 550
- To create them automatically using the command **File > Generate Script...** in Dymola.

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 **File > Generate Script...** in Dymola to create a script file

When creating a script file using Dymola, the command **File > 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 **File > 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 first 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.

The second is that if the user chooses to run the script file by the **Run Script** command button the working directory of Dymola will be changed to where the script file is located, which might not be a good idea. (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.)

(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. This is done automatically in Simulation mode (as a sub-window of Dymola main window), in Modeling mode select **Window > Tools > Commands** to display the Command window in the same way. If the Command window should be displayed as a stand-alone window, select **Window > New Command Window**.
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 544 for details. (If no operations are performed before saving the file, you will create an empty script file.)
3. Use the command **File > 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 **File > Generate Script...**, ticking **Plot setup**.

Note that presently zooming cannot be captured in the script file; the curves are always shown as initially presented (“rescaled”).

#### 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 **File > Generate Script...**, ticking **Animation setup**.

#### 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 the user can select to also include **Simulation setup**. By ticking this alternative the start time, stop time and integrator is also saved.

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 559.

## 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 **Commands > Add Command**. This is described in the chapter “Developing a model”, sub-section “Editor command reference – Modeling mode”, sub-section “Main window: Commands menu”.

When creating a script file using the command **File > 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 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.]
2. [Use the command **File > Clear log** to clear the content of the command log.]
3. Open the model (e.g. Pendulum).
4. Change the simulation time (to e.g. 10 s).
5. Simulate the model.

6. Show certain signals in a plot window (e.g. phi and 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 **File > 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. Test the script file.

The resulting script file might look like (opening the resulting file `TestPendulum.mos` by the Dymola script editor, using the command **Simulate > Script > Open Script...**):

```
// Script generated by Dymola Mon Mar 31 16:42:33 2014
simulateModel("Pendulum", stopTime=10, method="dassl", resultFile="Pendulum");
plot({"phi","der(phi)"}, colors={{0,0,255},{255,0,0}});
```

The first line is a comment automatically generated by Dymola. The four items of action above is 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 557.

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 cases, 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 549

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 529.

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 button **Run Script**  It is possible to browse for the script file, but please note that the search path of Dymola will be changed to location of the script file. (The **Run Script** command button is by default available in Simulate mode, but can be made available in Modeling mode as well using the command **Window > Tools > Simulation.**)
2. If the script file has been associated with a command (“*commandname*”) that command can be used to run the script file using **Commands > commandname**. (A script file can be associated with a command either by using **Store in model as command** in the command **File > Generate Script** when the script file was created or by using the command **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 552.
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.

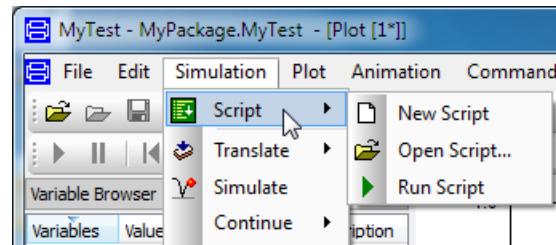
## 5.5.16 The Dymola script editor

### Accessing the Dymola script editor

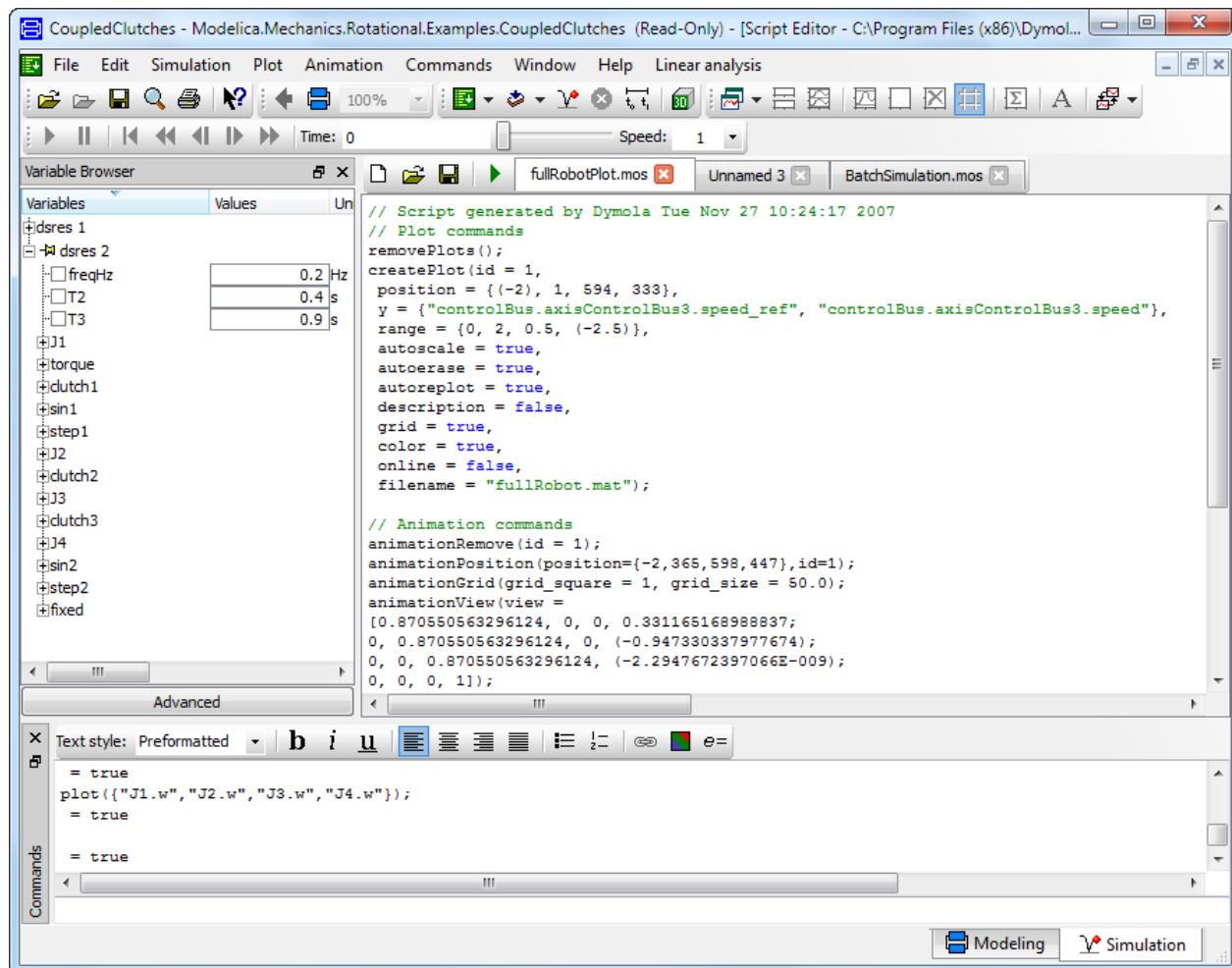
The following commands in Simulation mode will open the Dymola script editor.

- **File > Open...**
- **Simulation > Commands > New Script**
- **Simulation > Commands > Open Script...**

The last two commands are also available as selections from the **Run Script** button:



An example of the Dymola script editor:



Note the tab system in the script editor.

The script editor is by default displayed maximized, but can also be displayed as a subwindow or minimized (see 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 two ways to create a new empty script in the script editor:

- Using any of the command **Simulation > Commands > New Script** or the related **Run Script** button alternative **New Script**.
- Clicking the **New Script** button  in the header of the script editor.

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.

### **Opening an existing script**

There are three ways to open an existing script:

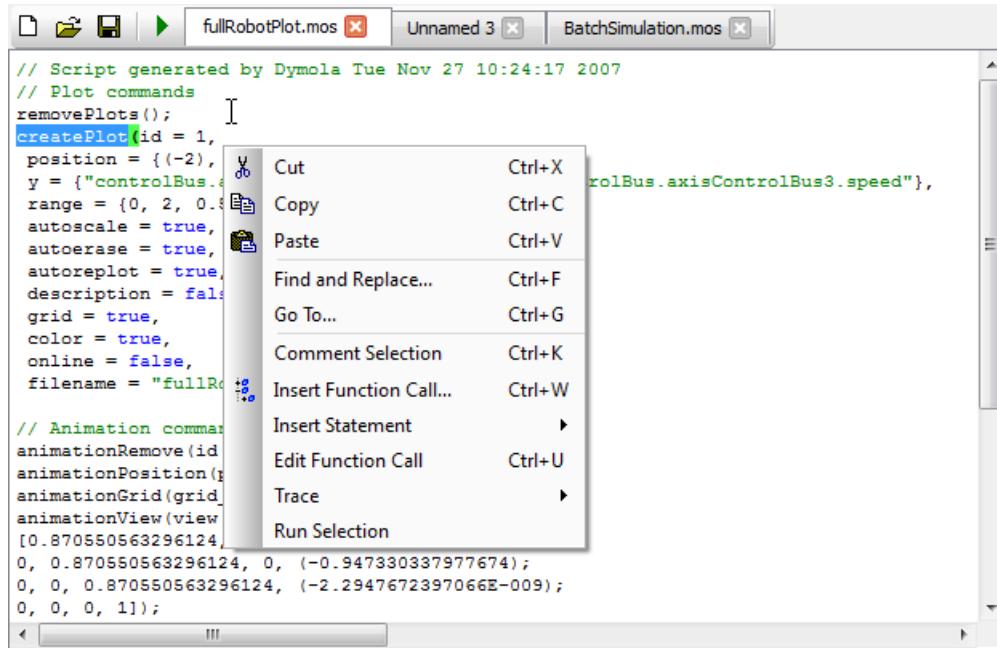
- By the command **File > Open....**
- By the command **Simulation > Commands > Open Script...** or the related **Run Script** button alternative **Open Script....**
- If the script editor is already opened, an existing script can be opened using the **Open Script** button  in the script editor header.

A new tab is created in the script editor containing the script; the name of the script is displayed in the tab.

### **Editing a script**

Any editing of a script not being saved is indicated by a star in the tab.

Apart from ordinary text editing, a context menu provides a number of options for editing:



The options (except the last two) 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 516.

### Saving a script

Having edited a script in the scripting editor, it can be saved two ways

- Using the **Save Script** button  in the script editor header.
- Using the command **File > Save** or the command **File > Save As...** (if the file should be saved with another name).

### 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.

## 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 563. 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 Dymola User Manual Volume 2.

#### Setting parameter values before simulation using a text file

This can be obtained by using the command **File > Generate Script... > Variables**. Further selection depends on what the user want. (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 (x86)\Dymola 2015 FD01\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 “Dymola User Manual Volume 2”, 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.CoupledClutches", 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 (x86)\Dymola 2015 FD01\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 are needed, and in some cases the size of the result file containing all signals will just be too big to work with.

## 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 419 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” in the manual “Dymola User Manual Volume 2”.

## **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.
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.
Documentataton	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 `getLastErrorHandler()`; 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( "func" )
```

The function document gives full documentation for a function, 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;
```

## help

```
help()
```

Gives a short overview of these commands

```
help functionname
```

Write the Modelica declaration of the given built-in function, and describe the function. This command 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;
```

## Simulator API

Function	Description
checkModel	"Check a model"
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"
getExperiment	"Get current experiment setting"
importFMU	"Import an FMU"
importInitial	"Import start values from file"
importInitialResult	"Import start values from file"
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 switches. 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)"
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.

### 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.

### 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)
```

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”.

### **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 468 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.

### **initialized**

```
initialized(allVars=false, isInitialized=true)
```

This function sets states and parameters to fixed and all other variables to free. It is used before setting initial values for states and parameters. `isInitialized=true` is default

(and roughly corresponds to continuing a simulation). If false it will initialize according to the initial equations at the start of the simulation.

### 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 469. 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. Also interactive settings of translator switches such as Evaluate are listed.

The output from the function is in alphabetical 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="*")
```

Writes a list of the built-in functions and their descriptions to the screen.

The wildcards \* and ? are supported.

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 prompted for removing the present one and open it again. The default value `false` can be useful in scripting, 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 548. `silent` means that commands are not echoed if this setting is true.

### **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).

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
= {6.2134129586543,0.9999999999999999}

```

It can be seen that the function call was executed successfully (= `true`); then the value of `J1.w` (6.213...) and `J4.w` (0.9999...) 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 created. If `parentName` is an empty string, a top level class is created.

### **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` and `simulateExtendedModel` 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
<code>simulateModel</code>		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.

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).

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:

$$\begin{aligned} &\text{true} \\ &= \begin{bmatrix} 6.2134129586543 & 1.0 \\ 7.4835581910107 & 1.0 \\ 8.1074463797378 & 1.0 \end{bmatrix} \end{aligned}$$

### 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).

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 corresponds to **Translate (Normal)** in the menus.

Note: If you give the name of the model you can skip `translateModel` and go directly to `simulateModel`.

Note: `translateModel`, `simulateModel` and `simulateExtendedModel` 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` and `simulateExtendedModel` 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**, and corresponding drop-down selection of the **Translate** button.

This functionality demands license. For more information, please see the manual “Dymola User Manual Volume 2”, chapter 6 “Other Simulation Environments”, section “Code and Model Export”.

### translateModelFMU

```
translateModelFMU(modelToOpen, storeResult, modelName,
fmiVersion, fmiType, includeSource)
```

Translates a model to an FMU. The input string `model` defines the model to open in the same way as the traditional `translateModel` command in Dymola.

The Boolean input `storeResult` is used to specify if the FMU should generate a result file (`dsres.mat`). 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. 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 model identifier (e.g. 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` define 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. Note – Co-simulation using Dymola solvers requires Binary Model Export license, and is currently only available on Windows. For this option it might also be noted that also the selected tolerance in Dymola will be used by the Co-simulation FMU, and, source code generation FMU is not supported by this alternative.

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.

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. Exporting an FMU using the 64-bit version of Dymola will create both 32-bit and 64-bit binaries if possible.

In Dymola 2013 and later, the `translateModelFMU` command will generate an FMU that supports both the FMI for Model Exchange specification and the FMI for Co-Simulation slave interface (all functions will be present in the DLL).

On Linux, note that FMU export requires the Linux utility “zip”. If not already installed, please install using your packaging manager (e.g. apt-get) or see e.g. <http://info-zip.org/Zip.html>.

This built-in function corresponds to the commands **Simulation > Translate > FMU** and corresponding drop-down selections of the **Translate** button. Refer to section “Simulation > Translate > FMU” on page 467.

For more information about FMI, please see the manual “Dymola User Manual Volume 2”, chapter 6 “Other Simulation Environments”, section “FMI Support in Dymola”.

### variables

```
variables(filename="VariableValues.txt", variables{ "Var1",
"Var2" })
```

Works as `list` above, but does not list interactive settings of translator switches.

## System

Function	Description
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"
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"

### classDirectory

```
classDirectory()
```

Returns current directory (the local directory where the call resides). This can be used to e.g. access local external files.

However, the presently recommended way is to use URIs instead, as available in the package ModelicaServices 1.2 or later. In this package a function ModelicaServices.ExternalReferences.loadResource is available, returning a file reference when an URI is input.

### **clearFlags**

```
clearFlags()
```

Resets all flags and integer constants to their default values.

### **eraseClasses**

```
eraseClasses({ "class1", "PackageA.model2", ... })
```

Erases the given classes. It requires that no classes 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. Corresponds to **Delete** 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

### **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. This latter alternative corresponds to the **File > Save...command**, ticking only the alternative **Command log**.

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)
```

The function saveSettings corresponds to the command **File > 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 “**File > Generate Script...**” on page 464 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.

## Plot

Function	Description
clearPlot	"Erase curves and annotations in the diagram"
createPlot	"Create a plot 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"
plotExpression	"Plot an expression"
plotHeading	"Add a heading to a plot window"
plotParametricCurve	"Plot parametric curve"
plotParametricCurves	"Plot parameteric curves"
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"
plotWindowSetup	"Generate a createPlot command of a plot window"
printPlot	"Plot and print given variables"
printPlotArray	"Plot and print given data"
printPlotArrays	"Plot and print given data"
removePlots	"Remove all plot windows"
removeResults	"Remove all result from plot windows"
signalOperatorValue	"Return the value of a signal operator for a given variable"

### clearPlot

```
clearPlot(id=0)
```

Erases curves and annotations in the diagram. `Id` is identity of window (0 means last).

### createPlot

```
createPlot(id, position[], x, y[:,], heading, range[], 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)], _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 output parameters except the last one. Some parameters are further commented in notes below the table.

Note that if having a plot already, the command **File > Generate Script... > Plot setup** will produce a script (.mos) file with relevant flags and the corresponding `createPlot` function call for the plot. See example below.

Parameter	Type and default	Comment
id	Integer = 0	Window id.
position	Integer[4]	Window position. See (1).
x	String = "Time"	Independent variable.
y	String[:]	Variables. See (2).
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 (3)
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.
legendLocation	Integer = 1	Where to place the legend. (1=above, 2=right, 3=below, 4-7=inside). See (4).
legendHorizontal	Boolean = true	Horizontal legend.
legendFrame	Boolean = false	Draw frame around legend.
supressMarker	N/A	Deprecated.
logX	Boolean = false	Logarithmic X scale.
logY	Boolean = false	Logarithmic Y scale.
legends	String[size(y,1)]	Legends. See (5).
subPlot	Integer = 1	Subplot number. See (6).
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.
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.
colors	Integer[size(y,1),3]	Line colors. See (5).
patterns	LinePattern[size(y,1)]	Line patterns. See (5).
markers	MarkerStyle[size(y,1)]	Line markers. See (5).

thicknesses	Real[size(y,1)]	Line thicknesses. See (5).
_window	Integer	Output parameter. See (7).

Notes:

1. 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.
2. 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".
3. This parameter is not influenced by the flag Advanced.DefaultAutoErase; this flag is only used when creating plot window by GUI commands.
4. This parameter corresponds to the Location group in the Legend tab in the plot setup. 4=Top-Left, 5=Top-Right, 6=Bottom-Left, 7=Bottom-Right.
5. See the built-in function `plot` below for description.
6. To create a plot with subplots several `createPlot` commands with the same id but different values of subplot have to be executed.
7. This output parameter will have the same value as the parameter id.

**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 **Commands > Simulate and Plot** to get a plot window.

Now use the command **File > Generate Script... > Plot setup** to save a script (.mos) file. 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 Aug 06 14:31:09 2014
// Plot commands
removePlots();
Advanced.FilenameInLegend = false;
Advanced.SequenceInLegend = true;
Advanced.PlotLegendTooltip = true;
Advanced.FullPlotTooltip = true;
Advanced.DefaultAutoErase = true;
Advanced.Legend.Horizontal = true;
createPlot(id = 1,
    position = {15, 10, 476, 337},
    y = {"J1.w", "J2.w", "J3.w", "J4.w"},
    range = {0.0, 1.2000000000000002, -2.0, 12.0},
    grid = true,
    filename = "dsres.mat",
    leftTitleType = 1,
```

```

bottomTitleType = 1,
colors = {{0,0,255}, {255,0,0}, {0,128,0}, {255,0,255}});

```

### ExportPlotAsImage

```

ExportPlotAsImage(filename="PlotImageName.png", id=-1,
includeInLog=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.

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)])

```

Plot the given variables in the plot window. It is currently not possible to set ranges or independent variable.

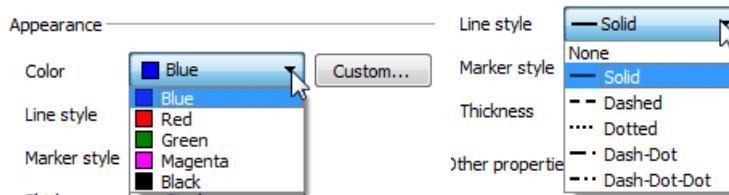
Note: the argument is a vector of strings; the names correspond to the names when selecting variables in the plot window. Subcomponents are accessed by dot-notation.

Variable names with and without result file name and sequence name (in brackets) are supported. 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,  $1 \leq i \leq \text{ndims}(X)$ ,
- `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 and thickness are supported; the possible selections in the GUI:



corresponds to the following input parameter values in the function:

Colors:

```

{0,0,255}
{255,0,0}
{0,128,0}

```

Patterns:

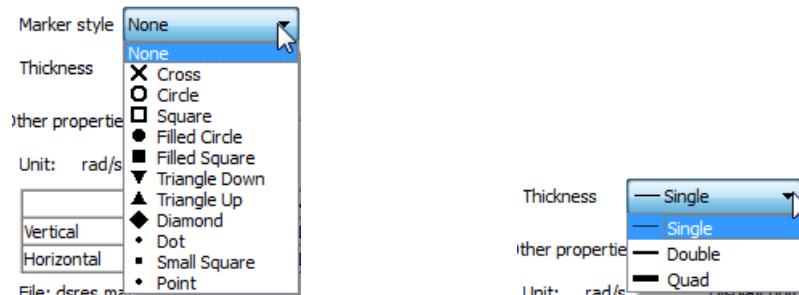
```

LinePattern.Solid
LinePattern.Dash
LinePattern.Dot

```

```
{255,0,255}  
{0,0,0}
```

```
LinePattern.DashDot  
LinePattern.DashDotDot
```



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
```

Thicknesses:

```
0.25  
0.5  
1.0
```

The corresponding GUI is described in the section “Plot > Setup...”; the **Variables** tab, on page 489.

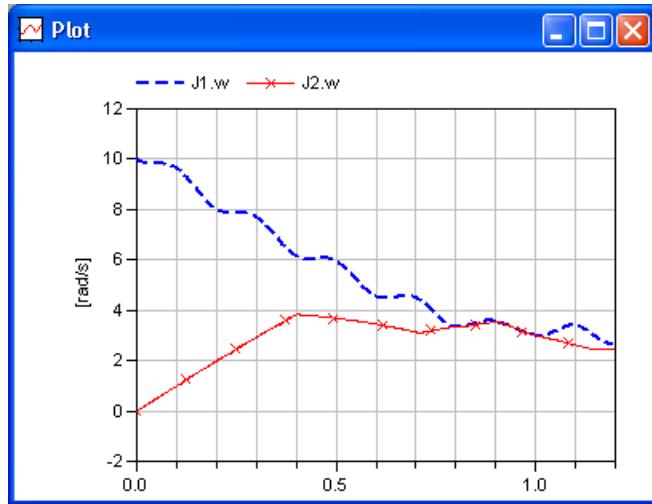
Example

Example of `plot()`:

Using the demo `CoupledClutches`, the function call

```
plot({"J1.w", "J2.w"}, colors={{0,0,255}, {255,0,0}},  
patterns={LinePattern.Dash, LinePattern.Solid},  
markers={MarkerStyle.None, MarkerStyle.Cross},  
thicknesses={0.5, 0.25});
```

corresponds to:



### plotArray

```
plotArray(x[:,], y[size(x,1)], style, legend, id, color[3],
          pattern, marker, thickness, erase)
```

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]" } )
```

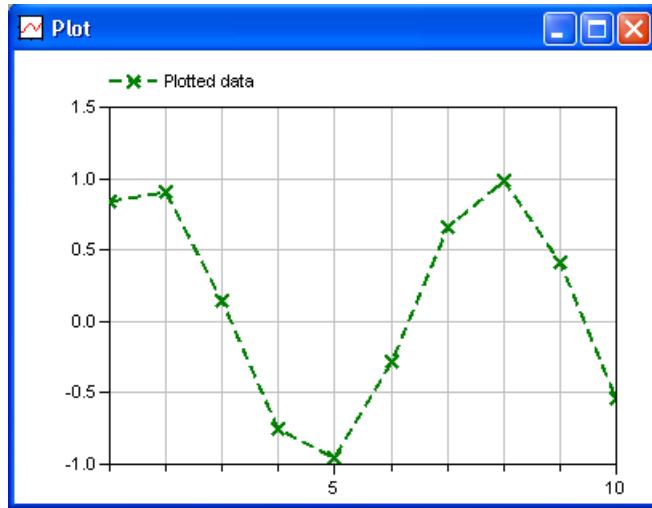
The `id` is the identity of the window (default 0 means last). Concerning size, pattern, marker and thickness, please see the `plot()` built-in function above. (The input `style` is deprecated.)

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)])
```

X-y plot for plotting of data computed in functions or scripts. Note similarity with the above function. (The input `style[:,]` is deprecated.)

### **plotExpression**

```
plotExpression(mapFunction, eraseOld, expressionName, id)
```

The function plots an expression in a specified plot window. The corresponding GUI handling is described in section ‘‘Plotting general expressions’’ starting on page 429.

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.

#### **Example**

```
plotExpression(apply(CoupledClutches[end].J1.w+CoupledClutches[
end-1].J1.w), false,
"CoupledClutches[end].J1.w+CoupledClutches[end-1].J1.w", 1);
```

### **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

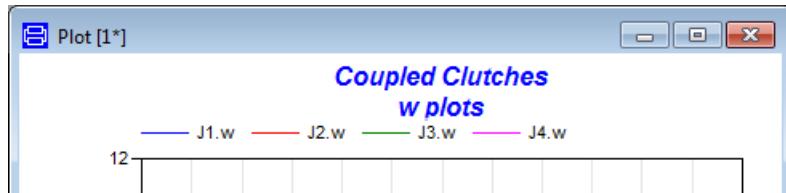
about font size, and about `textStyle` and `horizontalAlignment`, see section “`plotText`” on page 579.

### 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:



### plotParametricCurve

```
plotParametricCurve(x[:,], y[size(x,1)], s[size(x,1)], xName,
yName, sName, legend, id, color[3], pattern, marker, thickness,
labelWithS, erase)
```

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 434.

`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)
```

The function plots curves defined by  $x(s) - y(s)$ . The function is an extension of the function above, covering multiple curves.

### 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 421. 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.

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

```
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.

### **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 421.

**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)
```

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 **Edit > Options, Appearance** 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)

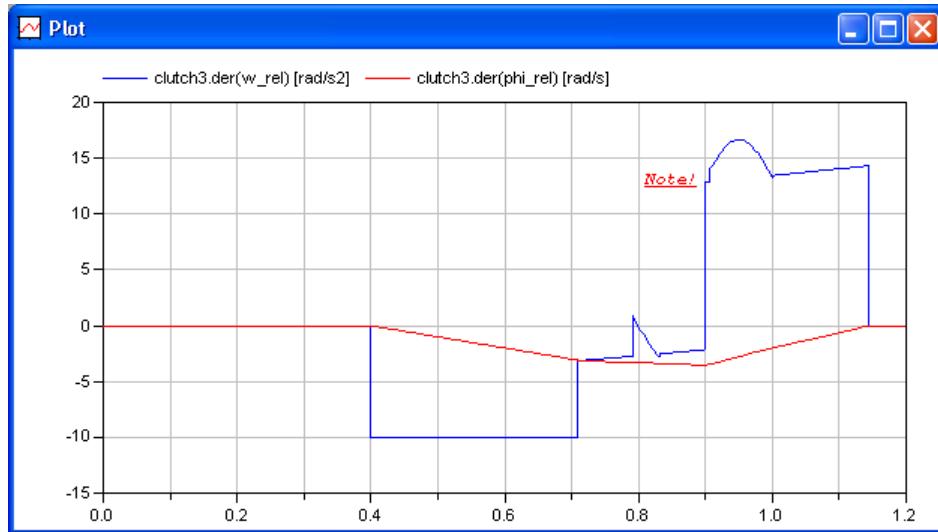
```
TextAlignment.Left  
TextAlignment.Center  
TextAlignment.Right
```

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 436.

### **printPlot**

```
printPlot({ "plot1", "plot2", "plot3" ... })
```

Plot the variables and furthermore prints the resulting plot on the default printer.

### **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.ArithmetricMean)
```

## Trajectories

Function	Description
existTrajectoryNames	"Check if provided names exist in trajectory 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 553.

## 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 “Dymola User Manual Volume 2”, chapter “Other 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)
```

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.

### 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. Use mml as file extension for MathML.

### exportIcon

```
exportIcon(path, width=80, height=80, trim=true)
```

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.

### getClassText

```
getClassText(fullName, includeAnnotations=false,  
formatted=false)
```

Returns the Modelica Text (as a string classText) for a given model, and in 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.

## Others

Function	Description
DefaultModelicaVersion	"Set default Modelica version"
DymolaVersion	"Return version and date of Dymola"
DymolaVersionNumber	"Return version number of Dymola"

---

## 5.6 Debugging models

This section contains information about how to solve problems of different kind.

We will in this section 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 “Plot > Open Result...” on page 487), 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.

### 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 manual “Dymola User Manual Volume 2”, 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 **Edit > Check** to check models and components to detect errors; see previous chapter, section “Editor command reference – Modeling mode”, sub-section “Main window: Edit menu”, command “Edit > 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 **Rename...** in the context menu for the class in package browser to move the class in package hierarchies. Use **File > SaveAs...** or **Duplicate** in the context menu for the class in package browser to copy a class in package hierarchies. (**Edit > Copy / Edit > Paste** and **Edit > Cut / Edit > 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 389.
- Try to provide good guess values for non-linear iteration variables.
- 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.

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 message window, which will pop up automatically when any error/warning should be displayed (see “Message window” on page 382 for more information). To display it other times, use the command **Simulation > Show Log** to display the window.

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 now also includes statistics for the initialization problem.

Concerning the diagnostics for nonlinear system of equations:

- 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 Dymola fails to solve a nonlinear system during simulation, the error message will include

```
Nonlinear system of equations number = xxx
```

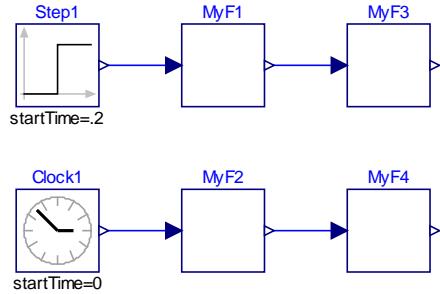
where `xxx` will be a number. In order to facilitate the location of the system, a corresponding line containing the number `xxx` is included in the file containing the C code, `dsmodel.c`. If flat Modelica code is generated in a `dsmodel.mof` file, the corresponding line will be present also in that file.

## 5.6.3 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 log

The log message 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 will in some cases not be the exact same variable, but can be an identical variable in the same or a connected sub-system.

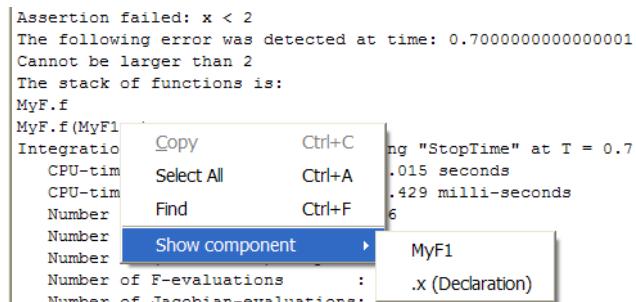
This is active as default but it is possible to de-activate this additional diagnostics, since it adds to the size of the generated C code and is only interesting if the model fails in some function.

### The direct link in the error log to the variables in the model window

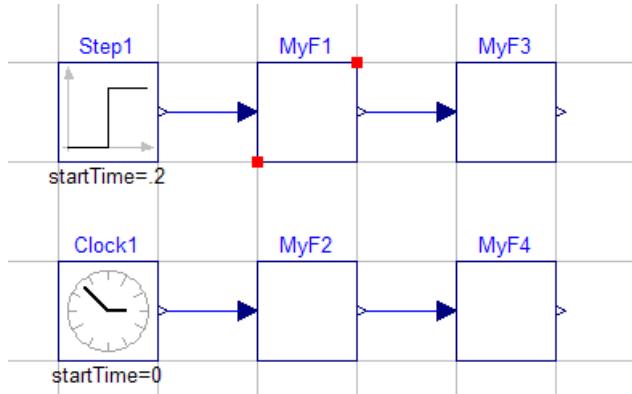
When looking at the error message for the WhichOne model there is a tooltip for the component as follows:

```
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
CPU-time [Context menu link: MyF1.x] : 0.015 seconds
CPU-time for one GRID interval: 0.429 milli-seconds
```

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.

### 5.6.4 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 “Event Logging” one will see a log where each expression causing an event is logged, see “Debug tab” on page 477. This makes it possible to track down errors in the discrete part of a hybrid system.

### 5.6.5 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 475, 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 **Simulate > Setup....**, selecting the **Output** tab and then unchecking all entries in the **Store** group except **State variables**. Please see section “Output tab” on page 475 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

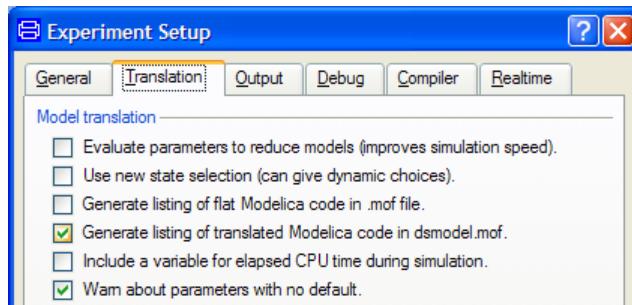
page 469. By examining the eigenvalues it is in general possible to track down the instability to some state.

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).

## 5.6.6 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. 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.

Below some examples are given for illustration.

## **Examples**

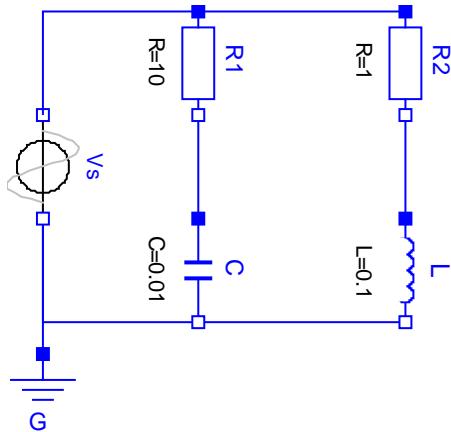
Below, some examples will be given to illustrate and to discuss the information given by dsmode1.mof.

- **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
```

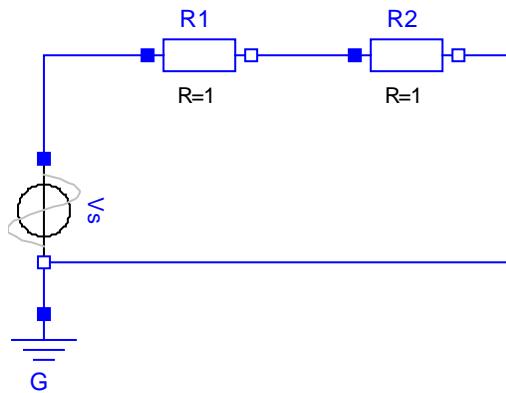
the last part of the listing becomes

```
// 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

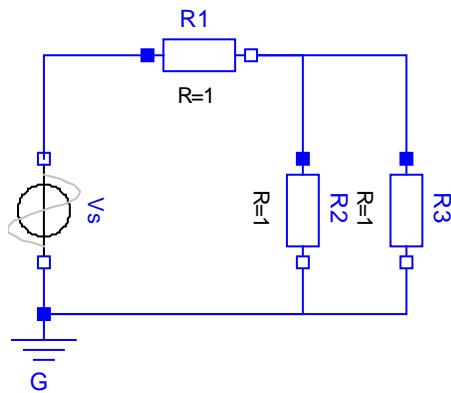
```
R1.i := Vs.v/(R1.R+R2.R);
```

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.

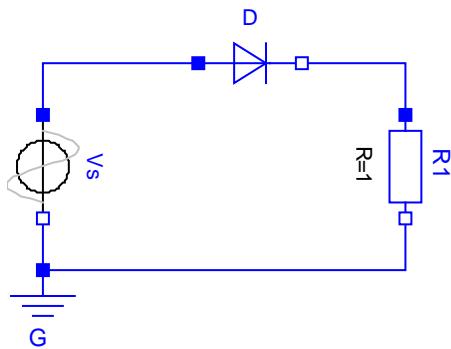
```
// Linear system of equations
// Matrix solution:
/* Original equations:
R1.R*R1.n.i = -R1.v;
R2.R*R2.p.i = R3.v;
*/
J[1, 1] := -(R1.R+R3.R);
J[1, 2] := -R1.R;
J[2, 1] := -R3.R;
J[2, 2] := R2.R;
b[1] := -Vs.v;
x := Solve(J, b); // J*x = b
{R3.p.i, R2.p.i} := x;
// Torn part
R1.n.i := -(R2.p.i+R3.p.i);
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  $D.\text{off}$ , 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
```

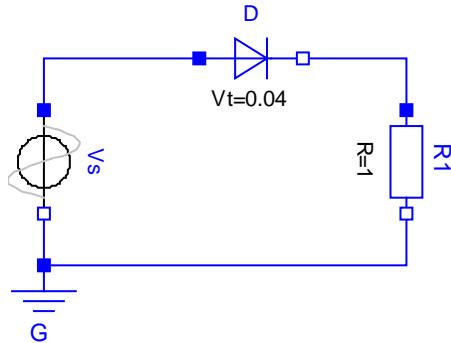
Dymola solves such a system by iterating. Assuming the Boolean variable to be known and removing the Boolean equation, the rest is a linear problem with four real unknowns. As seen, Dymola solves the linear part symbolically. During simulation, Dymola iterates, if the assignment  $D.\text{off} := D.s < 0$  changes the value of  $D.\text{off}$ .

## 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.

## 5.6.7 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,  $\text{der}(x)$ , 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 than 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. Also 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.

## 5.6.8 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 477 for information about the Debug tab.)



Note that types in Modelica.SIunits contain min values, and thus by using Modelica.SIunits some min values are automatically applied. However, based on the actual model it might make sense to add stricter limits.

## 5.6.9 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.

## 5.6.10 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 585.

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 583); 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 **Simulate > Setup...**, selecting the **Output** tab and deselecting storing of **State variables**, **Derivatives**, **Output variables** and **Auxiliary variables** (see section “Output tab” on page 475) 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 **Simulate > Setup...**, selecting the **Debug** tab and deselecting the entries in the **Event logging** group (see section “Debug tab” on page 477) 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 471. 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

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.)

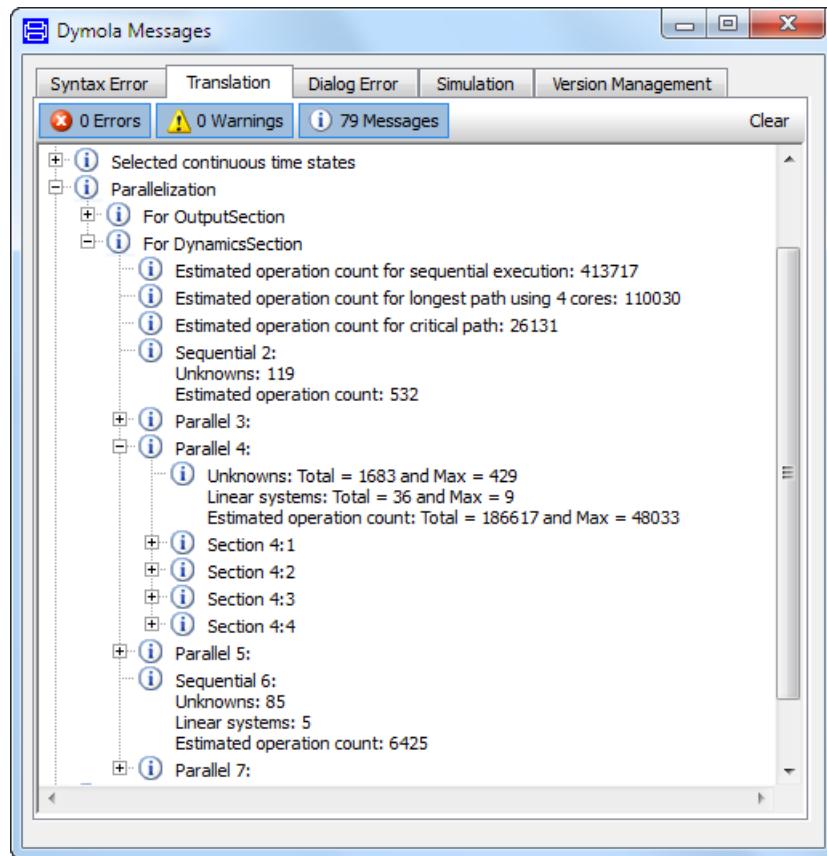
Notes:

- The compiler used must support OpenMP. For Visual Studio this means that you must use Visual Studio Professional 2010 or later, or Visual Studio Express 2012 or later. On Windows, the GCC version that Dymola supports, also supports OpenMP.
- Multi-core simulation is currently only supported for dassl, lsodar, euler and rkfix, when neither using DLL nor embedded server (DDE).

- For source code generation the generated C-code will contain standard OpenMP pragmas that are supported by many compilers (see their documentation).

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. Elmquist, 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/ECP140963\\_63\\_ElmqvistMattssonOlsson.pdf](https://www.modelica.org/events/modelica2014/proceedings/html/submissions/ECP140963_63_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.

### More efficient event handling

Setting the flag

```
Advanced.EfficientMinorEvents = true;
```

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 slight changes in the simulation intervals for the numerical solvers, the results may change within the tolerance.

### 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 475 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 390.

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 477 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 479.

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.**

```

Variable clutch1.mode = 0 at time 3674.29
Iterating to find consistent restart conditions.
Expression clutch1.sa < -clutch1.tau0_max became false < (clutch1.sa)-< -clutch1.tau0_max> = 6.50971
Variable W_[10] = 1 at time 3674.29
Iterating to find consistent restart conditions.
during event at Time : 3674.29193890593
Expression clutch1.sa > clutch1.tau0_max became true < (clutch1.sa)-<(clutch1.tau0_max> = 1.00269e-007
Variable W_[8] = 1 at time 3675.73
Variable W_[10] = 0 at time 3675.73
Expression clutch1.w_rel > 0 became true < (clutch1.w_rel)-<0> = 1.05953e-008
Variable clutch1.mode = 1 at time 3675.73
Variable clutch1.startForward = 1 at time 3675.73
Iterating to find consistent restart conditions.
Variable W_[8] = 0 at time 3675.73
Variable clutch1.startForward = 0 at time 3675.73
Iterating to find consistent restart conditions.
during event at Time : 3675.734636384474

Integration status probed at T = 3675.734636
CPU-time for integration : 66.8 seconds
CPU-time for one GRID interval: 363 milli-seconds
Number of result points : 26649
Number of GRID points : 185
Number of <successful> steps : 398477
Number of F-evaluations : 1429612
Number of H-evaluations : 508876
Number of Jacobian-evaluations: 79835
Number of <model> time events : 2
Number of <U> time events : 0
Number of state events : 13230
Number of step events : 0
Minimum integration stepsize : 9.98e-006
Maximum integration stepsize : 0.0213
Maximum integration order : 5
Enter command: continue(c), quit(q).
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

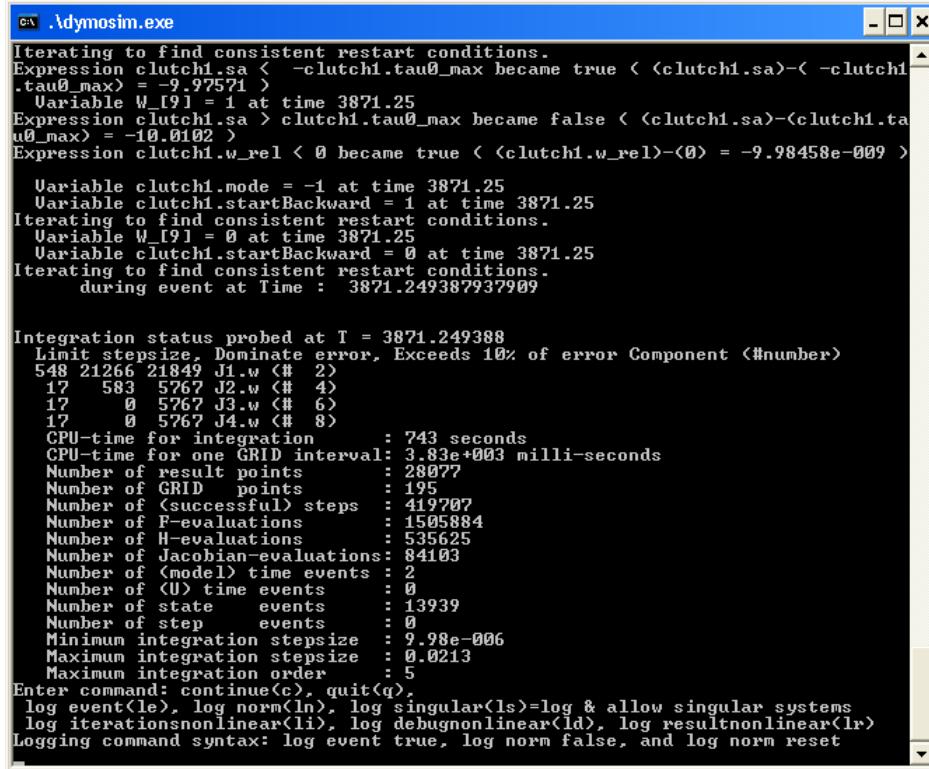
```

**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 (#number)	Name of the state. State number.

**Example of listing of integrator error statistics.**



```
.\dymosim.exe
Iterating to find consistent restart conditions.
Expression clutch1.sa < -clutch1.tau0_max became true < (clutch1.sa)-<-clutch1.tau0_max> = -9.97571
  Variable W_[9] = 1 at time 3871.25
Expression clutch1.sa > clutch1.tau0_max became false < (clutch1.sa)-<-clutch1.tau0_max> = -10.0102
Expression clutch1.w_rel < 0 became true < (clutch1.w_rel)-<0> = -9.98458e-009

  Variable clutch1.mode = -1 at time 3871.25
  Variable clutch1.startBackward = 1 at time 3871.25
Iterating to find consistent restart conditions.
  Variable W_[9] = 0 at time 3871.25
  Variable clutch1.startBackward = 0 at time 3871.25
Iterating to find consistent restart conditions.
  during event at Time : 3871.249387937909

Integration status probed at T = 3871.249388
Limit stepsize. Dominate error, Exceeds 10% of error Component <#number>
548 21266 21849 J1.w (# 2)
17 583 5767 J2.w (# 4)
17 0 5767 J3.w (# 6)
17 0 5767 J4.w (# 8)
CPU-time for integration : 743 seconds
CPU-time for one GRID interval: 3.83e+003 milli-seconds
Number of result points : 28077
Number of GRID points : 195
Number of <successful> steps : 419707
Number of F-evaluations : 1505884
Number of H-evaluations : 535625
Number of Jacobian-evaluations: 84103
Number of <model> time events : 2
Number of <U> time events : 0
Number of state events : 13939
Number of step events : 0
Minimum integration stepsize : 9.98e-006
Maximum integration stepsize : 0.0213
Maximum integration order : 5
Enter command: continue(c), quit(q),
log event(lc), log norm(ln), log singular(ls)=log & allow singular systems
log iterationsnonlinear(lin), log debugnonlinear(lnd), log resultnonlinear(lrn)
Logging command syntax: log event true, log norm false, and log norm reset
```

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 determine why 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.

This can be accomplished under Windows by first selecting **DDE server**, since it has access to highly accurate timers (accuracy in micro-seconds or nanoseconds), see “Compiler tab” on page 479. Selecting other compilers will only generate timers with millisecond resolution, which often is too inaccurate.

### Basic profiling

To turn on profiling write the following at Dymola’s command input

```
Advanced.GenerateBlockTimers=true
```

Then translate the model, and run the simulation. At the end of the log file a 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.

By running e.g. “Kinematic loop of an engine” demo (which takes about 5 seconds without profiling) one gets results such as

```
Profiling information for the blocks.
Estimated overhead per call      4[us] total      0.939[s]
the estimated overhead has been subtracted below.
Block, Total CPU[s], Mean[us] ( Min[us] to Max[us] ), Called
  0,      5.002,      252 (      38 to    30434),     22058
  1,      0.047,      329 (      237 to    9271),      149
  2,      0.030,       2 (       0 to     8632),     22058
  3,      0.933,       47 (      11 to    45327),     22057
  4,      0.000,       15 (       3 to      40),        4
  5,      0.000,       30 (       8 to      61),        4
  6,      0.000,       1 (       3 to      1),        1
  7,      0.067,       3 (       3 to    5045),     21812
  8,      0.562,       29 (      21 to    8675),     21812
  9,      0.036,       2 (       1 to    5117),     21812
 10,      0.063,       3 (       2 to    8843),     21812
 11,      0.178,       9 (       4 to   25083),     21812
 12,      0.154,       8 (       6 to    9243),     21812
 13,      1.774,       90 (      76 to   30257),     21812
 14,      0.015,       1 (       0 to    246),      7346
 15,      0.069,       9 (       8 to    179),      7346
```

The first lines state that we have estimated the overhead of the timers to 4 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 DymolaEndTimer(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 these blocks we have only subtracted the overhead for their own timers and not for additional timer calls that occur during their call. Thus the total time all model evaluations should more accurately be around 5s-0.9s (total overhead) = 4.1s. We can also estimate this by first measuring the total CPU time before including profiling, in this case 5s, and subtracting the time outside of the model evaluation, (block 3), we get 5s-0.933s = 4.1s.

In this example the event iterations are insignificant compared to the rest, even though there are 72 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 major costs are block 8, which is a non-linear equation for solving the angle of the cylinder, and block 13, which is the linear system for the loop (three unknowns). Together they account for about 2.3s (out of the models total of about 4.1s). Thus these two blocks explain more than half of the time spent evaluating the model. Since the model contains a kinematic loop this was as expected.

Note that block 5 has the largest average time of the normal blocks, but no 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 four times.

One should remember to reset the switch

```
Advanced.GenerateBlockTimers=false
```

in order to avoid the overhead for the next model. This switch is not cleared by **Clear** or **Clear All** commands.

The timer overhead also affects the total simulation time as well, and 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 some other 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. In that case 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.

By running e.g. "Kinematic loop of an engine" demo (takes about 5s without profiling) one gets results such as

```
Profiling information for the blocks.  
Estimated overhead per call      4[us] total      9.930[s]  
the estimated overhead has been subtracted below.  
Block, Total CPU[s], Mean[us] ( Min[us] to Max[us] ), Called  
    0,      28.284,      1425 (     465 to     51172),      22058  
    1,      0.315,      2191 (     1930 to     10815),      149  
    2,      0.020,      1 (       0 to      5621),      22058  
    3,      1.024,      52 (      12 to     41656),      22057  
    4,      0.000,      17 (       4 to       50),          4  
  
280,      0.022,      1 (       0 to     8116),      21812  
281,      0.611,      31 (      22 to     9959),      21812
```

282,	0.014,	1 (	0 to	99),	21812
301,	0.028,	1 (	0 to	8533),	21812
302,	0.132,	7 (	4 to	8530),	21812
303,	0.032,	2 (	0 to	8680),	21812
348,	0.019,	1 (	0 to	464),	21813
349,	1.835,	94 (	80 to	50319),	21813
350,	0.024,	1 (	0 to	8747),	21813

The two dominating blocks, 281 and 349, are the equation blocks found with the coarser grained profiling.

However, note that block 302 is a complex equation giving the gas force of the engine. Since the equation has 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, one evaluates both pieces at the switch and observes that they give the same result.

There is also an outer `if v_rel<0`, which is not continuous. However, for this complete model ‘v\_rel’ changes sign when ‘x’ is at the minimum (0) and thus for this specific model that expression is in fact continuous. Using such global information breaks the idea of object-oriented models and is thus not a good idea in modeling.

**Turning events off may increase simulation time.**

When using noEvent for performance reasons one should always measure the total CPU time since turning off events can cause a drastic increase in the number of steps. This did not occur for this example, but no substantial improvements were found and there is no reason to introduce noEvent.

Examining the average time for block 302 after the change one observes that it is not influenced by this change. Thus the overhead for generating events for ‘>’ and ‘<’ is only marginal, and no reason for using noEvent.

On the other hand, rewriting the polynomials with Horner’s rule decreased the average from 7 to 3 microseconds. Even that had marginal influence on the total CPU-time in this example.

One should remember to reset the switch

```
Advanced.GenerateTimers=false
```

in order to avoid the overhead for the next model. Note that this switch is not cleared by **clear** or **Clear All** commands.

## 5.7.6      **Inline integration**

**Inline integration requires the Realtime option.**

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 Elmquist *et al.* (1995).

### Inline integration

Consider an ordinary differential equation on explicit state space form

$$\dot{x} = f(x, t); \quad x(t_0) = x_0$$

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:

$$x_{n+1} = x_n + h \cdot f(x_n, t_n); \quad x_0 \text{ is known}$$

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

$$x_{n+1} = x_n + h \cdot f(x_{n+1}, t_{n+1})$$

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 <sup>10</sup>	4
Implicit Runge-Kutta	5
Explicit Runge-Kutta	6

#### 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 implicit Runge-Kutta order can be selected between 2 and 4. Optimize code for step size can be selected. For more information, please see section “Realtime tab” on page 482. 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;
```

---

<sup>10</sup> The mixed explicit/implicit Euler is particularly useful for mechanical systems.

```
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 482.

## More information and advanced options

Real-time simulation in other simulation environments is described in the manual “Dymola User Manual Volume 2”, chapter “Other 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 Handling initialization and start values

Please also see 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.

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 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  $v_0 \approx$ , 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.

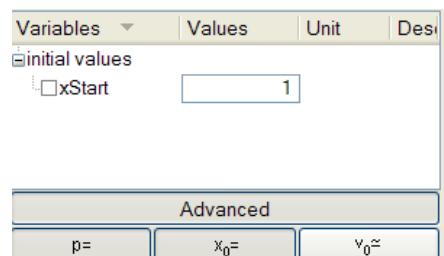
### Example

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 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

Variables	Values	Unit	Description
initial values			
<input type="checkbox"/> z	0		
<input type="checkbox"/> xStart	1		

Advanced		
p=	x <sub>0</sub> =	y <sub>0</sub> ≈

Setting 'z' to 1 generates another solution for this non-linear system of equations.

# **6 APPENDIX - INSTALLATION**



# 6 Appendix — Installation

This chapter describes the installation of Dymola on Windows and Linux, and related topics.

The content is the following:

In section 6.1 **“Installation on Windows”** starting on page 624 the installation on Windows is described, including installation of Dymola software, C compiler and license (shareable or node-locked). The sub-section “Additional setup” starting on page 635 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 6.2 **“Installation on Linux”** starting on page 644 the installation on Linux is described, in a similar way as the previous section. The sub-section “Additional setup” starting on page 645 describes e.g. compilation of model code and simulation from the command line.

In section 6.3 **“Dymola License Server on Windows”** starting on page 647 the installation of a license server on Windows is described, as is the borrowing of licenses.

In section 6.4 **“Dymola License Server on Linux”** starting on page 656 the installation of a license server on Linux is described, as is the borrowing of licenses.

In section 6.5 **“Utility programs”** starting on page 657 a utility program for finding a host id on a computer is described.

In section 6.6 **“System requirements”** starting on page 658 the hardware and software requirements/recommendations are listed.

In section 6.7 “**License requirements**” starting on page 660 the license requirements for various features are listed.

In section 6.8 “**Troubleshooting**” starting on page 665 the solution to various problems are described. It might be license file problems, compiler problems, issues with Simulink, change of language etc.

---

## 6.1 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.

### 6.1.1 Dymola as 32-bit and 64-bit application

The Dymola program is available both as 32-bit and 64-bit applications. Both are installed when installing Dymola on Windows. The 64-bit Dymola program (and its associated DLLs) is located in the folder `Program Files (x86)\Dymola 2015 FD01\bin64` after installation.

### 6.1.2 Installing the Dymola software

Dymola and appropriate libraries is distributed on a single CD or downloaded electronically.

#### Starting the installation

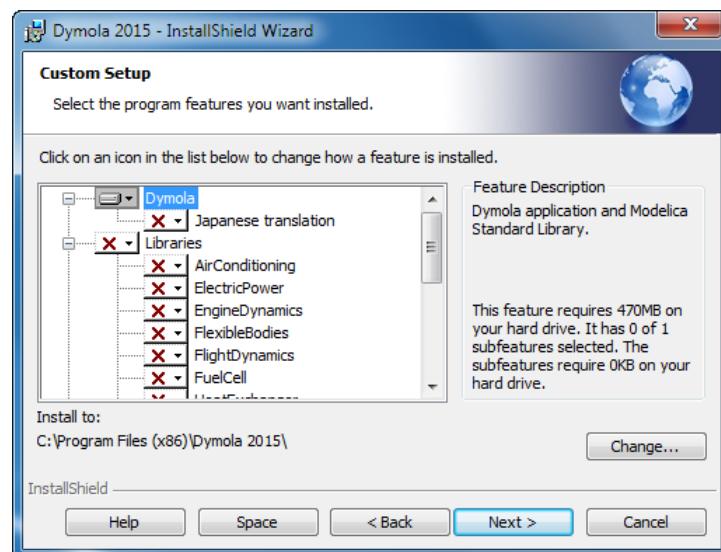
Please note that Administrator privileges are required for this installation. When Dymola has been installed, any user can run it.

The installation normally starts when you insert the distribution CD. If autostart has been disabled, please start `D:\setup.exe` (assuming your CD 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:



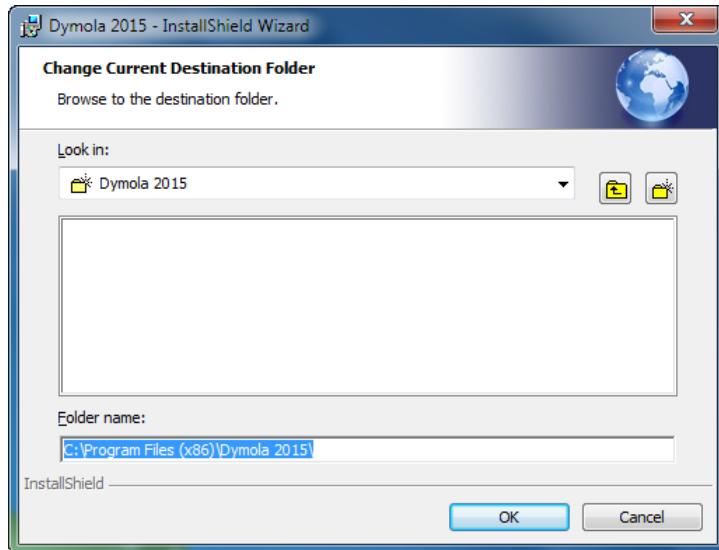
### **Location of directory**

The first choice in the installation procedure is the type of installation and the name of the Dymola installation directory. The default is:

- On 64-bit computers: C:\Program Files (x86)\Dymola + the version number of Dymola.
- On 32-bit computers: 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.



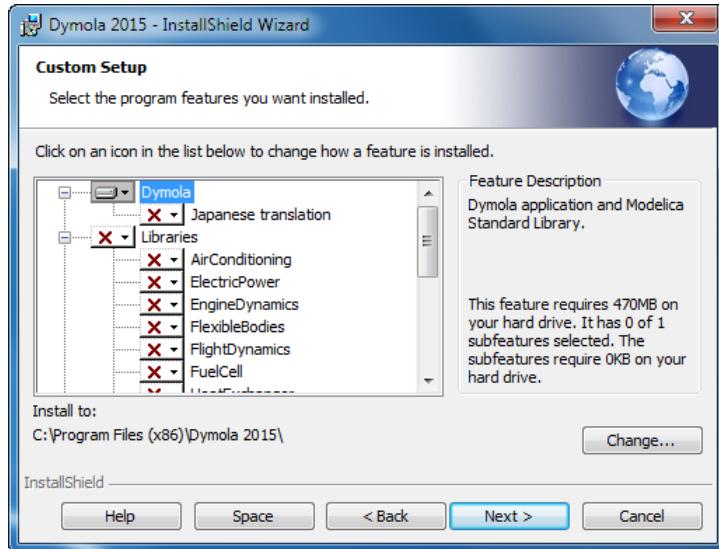
Dymola defines an environment variable **DYMOLAWORK** which value is the Dymola working directory. This is the default location where Dymola starts.

Dymola by default starts in the directory **My Documents\ Dymola** (that subdirectory will be created if it doesn't exist). Please note that this cannot be an UNC path (i.e. **\server\...**).

### Selecting components

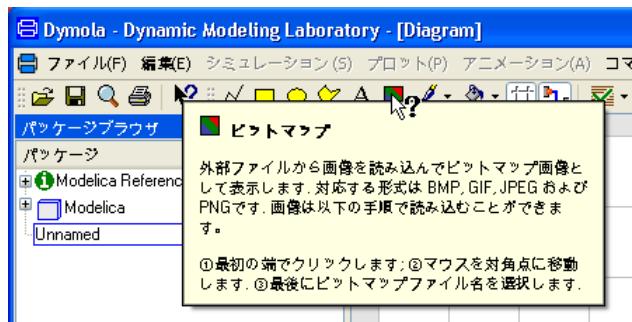
The second choice is to select optional components of the distribution. By unselecting components some space can be saved.

## 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).

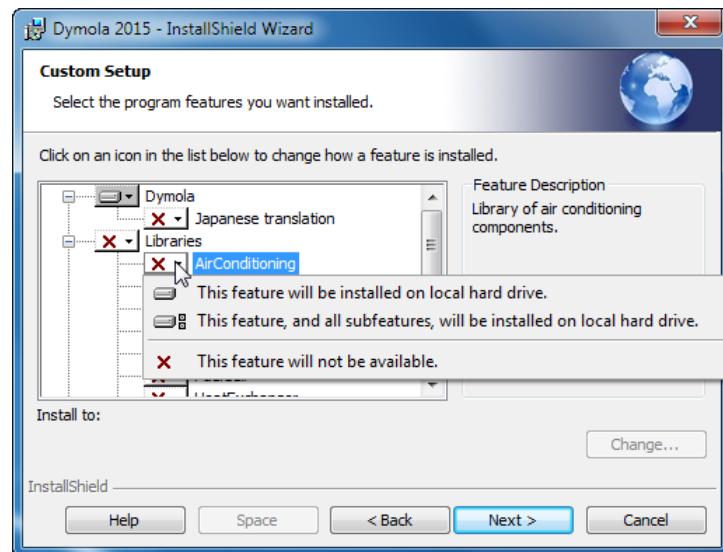
The **Japanese translation** of menus and dialogs requires fonts that support all symbols.



The **Libraries** section contains several commercial libraries which 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.**



### **6.1.3      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.

#### **Microsoft compilers**

Dymola supports Microsoft Visual Studio 2013, both the Professional edition and the Express edition. Dymola also supports older Microsoft compilers (Visual Studio 2012, Visual Studio 2010 and 2008 Professional edition and Express edition, and Visual Studio 2005 Professional edition).

To download the free Visual Studio 2013 Express edition compiler please visit

<http://www.Dymola.com/compiler>

where the latest links to Microsoft's website are available. 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.

**Please note** that earlier free versions of the Microsoft compiler are not supported; the reason is that they do not include a full set of Windows libraries. We recommend Visual C++ 2010 or later (see above).

#### **Improving the code efficiency when using the Visual Studio 2013 and 2012 compilers**

The Visual Studio 2013 and 2012 compilers are fully supported. However, these compilers by default generate a bit less efficient code than previous versions of the compiler, with the selected optimization settings. As a temporary work-around you can set the flag

`Advanced.Define.GlobalOptimizations = 2;`

before generating code, to activate global optimization in the compiler. (The default value of the flag is 0.)

This flag works the same for all Visual Studio compilers, but the effect on compilers of previous versions is small. For the Visual Studio 2013 and 2012 compilers, however, the simulation performance is restored, but the compilation of the code might take substantially longer for large models.

The setting above corresponds to the compiler command `/Og`.

#### **GCC compiler**

Dymola 2015 FD01 has limited support for the MinGW GCC compiler, with a GCC version compatible with 4.8. To download this free compiler, please visit

<http://www.Dymola.com/compiler>

where the latest link to downloading the compiler is available. Note that you need administrator rights to install the compiler.

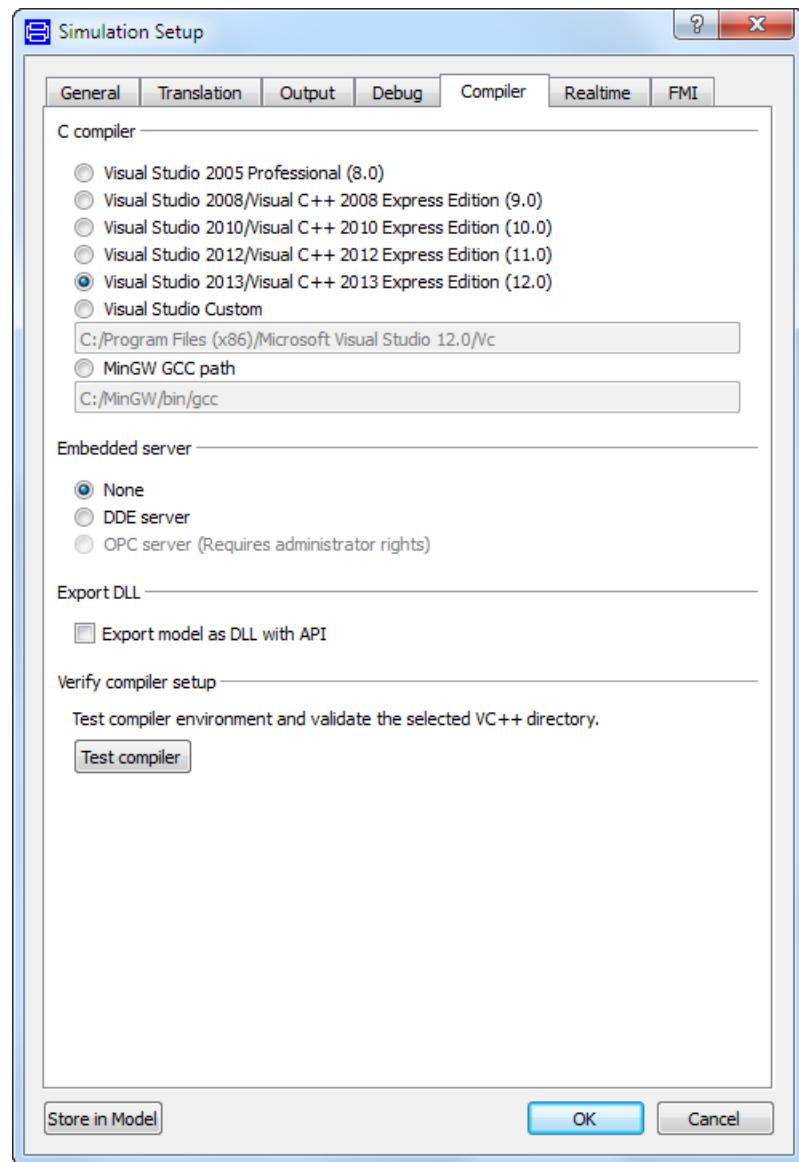
***Please note:***

- To be able to use other solvers than Lsodar and Dassl, 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:
  - Only ordinary simulation, and DLL, is currently supported (no FMU Export, DDE or OPC servers).
  - Only 32-bit simulation is supported
  - Commercial libraries: Only limited testing has been done; no support for external library resources.
  - No support for run-time license.

## **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 chapter “Simulating a model”, section “Editor command reference – Simulation mode”, sub-section “Main window: Simulation menu”, command “Simulation > Setup...”. (Below is an example of the **Compiler** tab).



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.

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. The reason is that single-threaded compilation is phased out in Visual Studio 2005, and multi- and single-threaded libraries reading from files are not link-compatible. Thus Dymola only supports linking with multi-threaded libraries in Microsoft Visual Studio compilers.

For information about possible compiler problems, please see the troubleshooting section “Compiler problems” on page 667.

## 6.1.4 Installing the Dymola license file

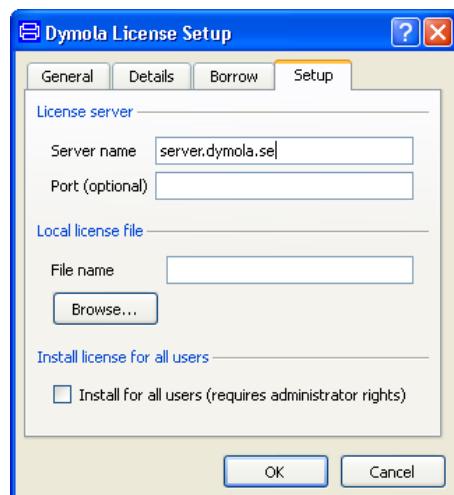
After installation Dymola will initially start in “demo” mode. While running in demo mode you can continue with installing the license file.

### 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.

Start Dymola and select **Help > License...**, 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.

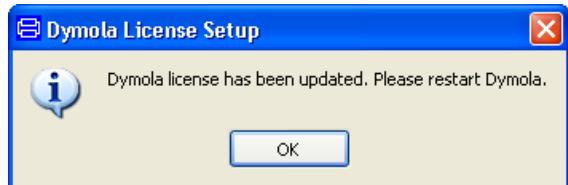


You have the option of installing the license file only for the currently logged in user, or for all users on this computer. The latter requires administrator rights.

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.



## 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

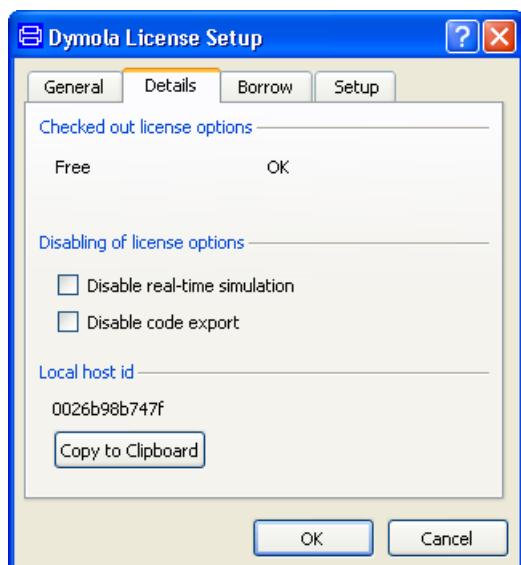
To order a *node-locked license key*, the relevant host id of the computer where Dymola should run must be supplied to your Dymola distributor. The license that you will receive will contain this information.

There are two ways finding out this host id, depending on whether a Dymola demo is installed before or not. The host id can always be found using the utility program `hostid.exe`. Please see section “Obtaining a host id” on page 657 for more information about this program.

If the Dymola demo has already been installed, Dymola can be used to find the host id. Start Dymola and select **Help > License...**, 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.**



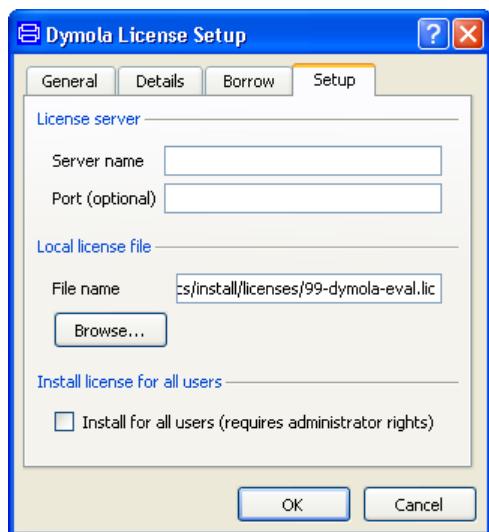
Compose an e-mail containing your local host id (host id's) and send it to your Dymola distributor.

### Installing the node-locked license

When you have received your license file, do save the license somewhere on your computer.

Start Dymola and select **Help > License...**, 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.

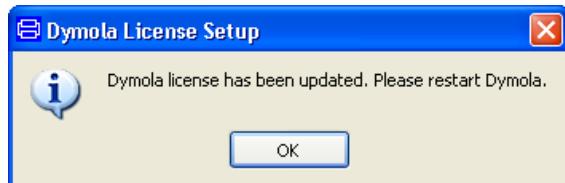


You have the option of installing the license file only for the currently logged in user, or for all users on this computer. The latter requires administrator rights.

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.



## Run-time licenses

Models developed by users that lack export options can still be run at other computers using a run-time license. Dymola run-time requires the user of the model to have the option `DymolaRuntime`. The license file containing the run-time license should be defined by the environment variable `DYMOLA_RUNTIME_LICENSE`, for example

```
set DYMOLA_RUNTIME_LICENSE=C:\My Documents\dymolaRT.lic
```

For information about license requirements in general, see section “License requirements” starting on page 660.

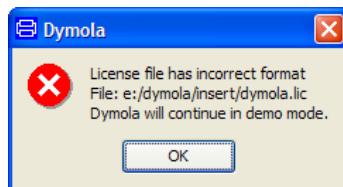
For more specific information about export options in particular, see the manual “Dymola User Manual Volume 1”, chapter 6 “Other Simulation Environments”, section “Code and Model Export”.

## 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 demo mode.

**Dymola has started  
with an old license file.**



## 6.1.5 Additional setup

### Creating shortcuts to Dymola

**Shortcuts start Dymola  
in the right directory.**

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 (x86)\Dymola 2015 FD01\bin\dymola.exe` by default for 32-bit Dymola on a 64-bit computer, `Program Files (x86)\Dymola 2015 FD01\bin64\dymola.exe` for 64-bit Dymola on a 64-bit computer).

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. Specify a working directory in the **Start in** field.

### **Remote installation of Dymola**

Dymola (whether downloaded as a zip file or on CD) consists of a number of files (.msi and .cab). Remote installation of dymola.msi is possible using the appropriate tools, such as msiexec. For example, the following command makes a quiet installation of Dymola and all libraries with Modelica version 3:

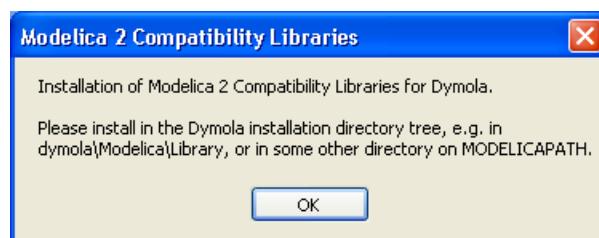
```
msiexec /i dymola.msi INSTALLLEVEL=201 /quiet
```

The value of the `INSTALLLEVEL` property controls which components are installed according to the table:

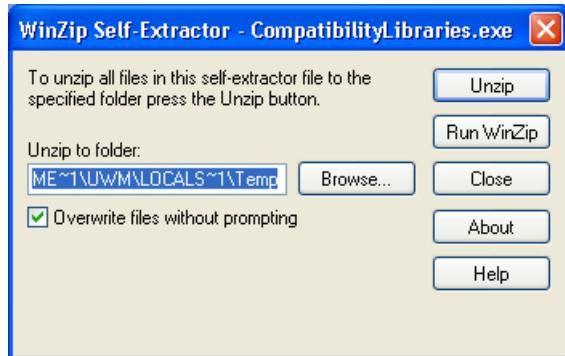
<b>INSTALLLEVEL</b>	<b>Description</b>
unspecified	Installs Dymola and standard libraries
201	As above and also installs commercial libraries compatible with Modelica language version 3.
301	As above and also installs commercial libraries compatible with Modelica language version 2.2.2
1001	As above and also installs Japanese translations of dialogs and menus

### **Installing Modelica 2-compatible libraries**

Modelica 2-compatible libraries that might be needed to work with an older Modelica model that has not been converted to the new libraries have to be installed separately. However, they are included in the Dymola media as `extra\CompatibilityLibraries.exe`. Running this file will display a message:



Selecting **OK** will display:



Typically a path to browse to is

C:\Program Files (x86)\Dymola 2015 FD01\Modelica\Library

Selecting **Unzip** will unzip the files to this folder.

### **Adding libraries and demos to the File menu**

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.

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.

Dymola will find libraries by searching all directories in the environment variable MODELICAPATH. If not set by the user, MODELICAPATH contains dymola/Modelica/Library.

### **Using library information**

Associated with each package is a Modelica script which 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 stored in subdirectory Scripts. Assuming the package is stored in dymola/Modelica/Library/XYZ, the script is called dymola/Modelica/Library/XYZ/Scripts/libraryinfo.mos. Alternatively, the file can be stored in the same directory as the library, e.g. dymola/Modelica/Library/XYZ/libraryinfo.mos.

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.:

```
LibraryInfoMenuCommand(category="libraries",
    text="Hydraulics",
    reference="Hydraulics",
    isModel=true,
    description="Hydraulics Library 4.1 by Modelon",
    version="4.1",
    ModelicaVersion=">=3.2"
    pos=3450);

LibraryInfoMenuCommand(category="demos",
    text="My Demo",
    reference="MyDemo.MyDemoModel",
    isModel=true,
    description="My demo, basic",
    pos=9999);
```

The attributes have the following meaning:

Attribute	Meaning
category	Primary menu category ("libraries", "demos", or "persistent")
subCategory[:]	Optional sub-categories
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 <b>Important</b> 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.

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.

## 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.

### **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 manual “Dymola User Manual Volume 2”, chapter 7 “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 demo 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 32-bit Dymola with default location on MS Widows on a 64-bit computer, Dymola could be started with the following command line using the Command Prompt in Windows:

```
"C:\Program Files (x86)\Dymola 2015 FD01\bin\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 (x86)\Dymola 2015 FD01\bin\Dymola.exe" -days  
-5
```

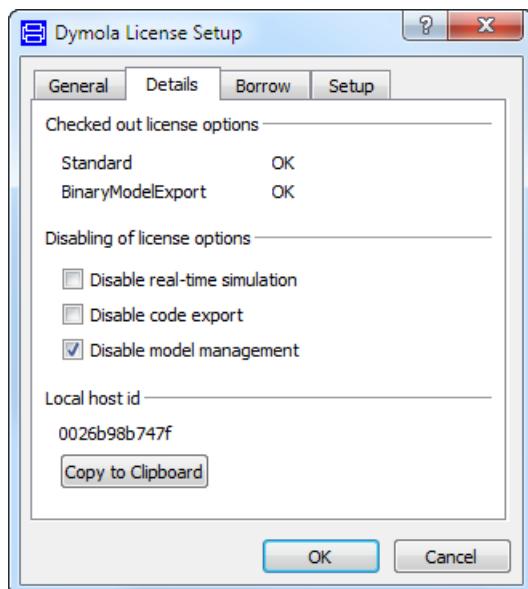
1 day is the minimum start time for warning of license expiration; the warning cannot be completely disabled.

### **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 **Help > License...** 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:

- By ticking any of the options in the **Disabling of license options** group in the menu above. The second setting will prevent both BinaryModelExport and SourceCodeGeneration to be checked out. These settings will be remembered between Dymola sessions. They can also be accomplished using the following flags (the below corresponds to prevent check-out of the options, respectively).

```
Advanced.EnableRealtimeSim=false;  
Advanced.EnableCodeExport=false;  
Advanced.EnableModelManagement=false;
```

For more information concerning real-time simulation and code export, please see the manual “Dymola User Manual Volume 2”, chapter “Other Simulation Environments”.

- By modifying the shortcut to Dymola.
- By starting Dymola with a certain 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

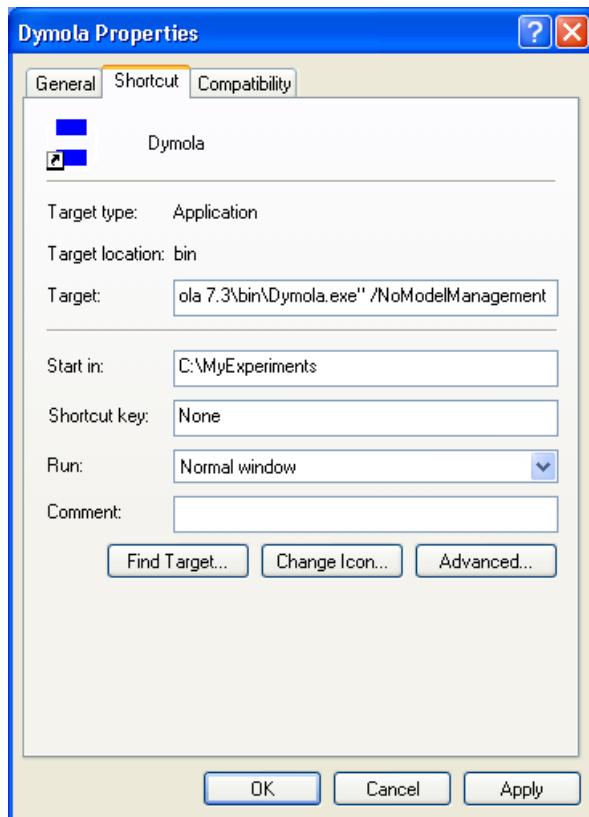
way is to look at the checked out options using the command above, and mark and copy the name of the option that should not be checked out, to insert that name when using any command.

### Modification of the shortcut to Dymola

If a new shortcut is needed, please look at the section “Creating shortcuts to Dymola” above.

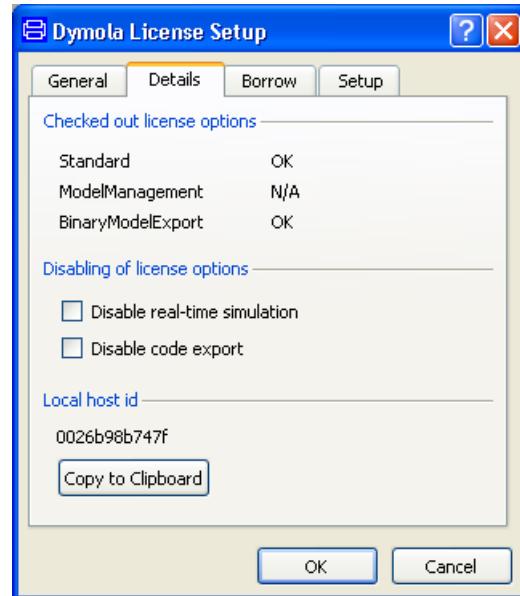
To modify the shortcut to prevent checking out a certain option, right-click the shortcut and modify the **Start in:** by adding <space>/No<optionname> in the end of the command. If the option ModelManagement in the figure above should not be checked out, the shortcut should be modified like in the figure below.

#### 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 ModelManagement will not be possible to check out. As long as the shortcut is not modified, ModelManagement will not be possible to check out from Dymola started by that shortcut.

To enable check out of ModelManagement, Dymola must be closed and then restarted using a shortcut without the command line option for ModelManagement.

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 ModelManagement cannot be checked out like in the example above, the command in the command prompt will look like:

A screenshot of a Windows Command Prompt window titled 'Command Prompt'. The command entered is 'C:\MyExperiments>"C:\Program Files\Dymola 7.3\bin\Dymola.exe" /NoModelManagement'. The window has standard Windows title bar and control buttons.

## 6.1.6 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 **Add or Remove Programs**. 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.

### 6.1.7 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.

### 6.1.8 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.

---

## 6.2 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 632.

The default directory for installation on Linux is `/opt/dymola-<version>-<architecture>`, where the architecture is RPM standard, `i586` for 32-bit, and `x86_64` for 64-bit application. As an example, the default directory for installation of 64-bit Dymola 2015 FD01 on Linux is `/opt/dymola-2015FD01-x86_64` (the package manager on the target system however typically allows choosing another default location).

Dymola 2015 FD01 runs on SUSE Linux (Release 11), 32-bit and 64-bit, with gcc version 4.3.4, and compatible systems.

In addition to gcc, the model C code can also be compiled by clang. To change compiler, change the variable `CC` in

```
/opt/dymola-<version>-<architecture>/insert/dsbuild.sh
```

(See above concerning `<architecture>`.)

Dymola 2015 FD01 is supported as a 32-bit and a 64-bit application on Linux. Corresponding support for 32-bit and 64-bit export and import of FMUs is included.

#### Notes

- For rendering of jpg files, libjpg62 must be installed.
- 32-bit compilation might require explicit installation of 32-bit libc. E.g. on Ubuntu:  
`sudo apt-get install g++-multilib libc6-dev-386`

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.

More Linux-specific notes are available using the command

```
man dymola
```

### 6.2.1

## 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 -k name-of-distribution.rpm
```

### Setup and environment variables

The shell script `/usr/local/bin/dymola-<version>-<architecture>` (see above concerning “`-<version>-<architecture>`”) 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>-<architecture>`).

**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 “Adding libraries and demos to the File menu” on page 637.

(Dymola defines an environment variable DYMOLAWORK which value is the Dymola working directory.)

### 6.2.2

## 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>-<architecture>/insert/dsbuild.sh` (see above concerning “`<version>-<architecture>`”). 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>-
<architecture>/bin/lib
```

Security-Enhanced Linux (SELinux) might display the message (below example for 64-bit application):

```
dymosim: error while loading shared libraries: /opt/dymola-
2015FD01-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-2015FD01-
x86_64/bin/lib/libds.so
# chcon -t textrel_shlib_t /opt/dymola-2015FD01-
x86_64/bin/lib/libGodessMain.so
```

Note that running simulations in the Dymola environment do not require these changes.

## **Adding libraries and demos to the File menu**

Please see corresponding section for Windows installation on page 637.

## **Preventing checking out 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 642.

### **6.2.3      Removing Dymola**

Remove the Dymola distribution by using the `rpm -u` command.

## 6.3 Dymola License Server on Windows

### 6.3.1 Background

**This section refers only to the Windows version of Dymola.**

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 (x86)\Dymola 2015 FD01\bin`.

Dymola is installed on all machines which 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 `lmgard.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.

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 `lmgard.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 11.11. A recent version of `lmgard.exe` is part of the Dymola distribution.

If needed, the latest available license daemon can be downloaded from the website of Flexera Software when having completed a form:

[http://mktg.flexerasoftware.com/mk/get/lmgard\\_reg](http://mktg.flexerasoftware.com/mk/get/lmgard_reg)

### 6.3.2 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.

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 657 for more information.

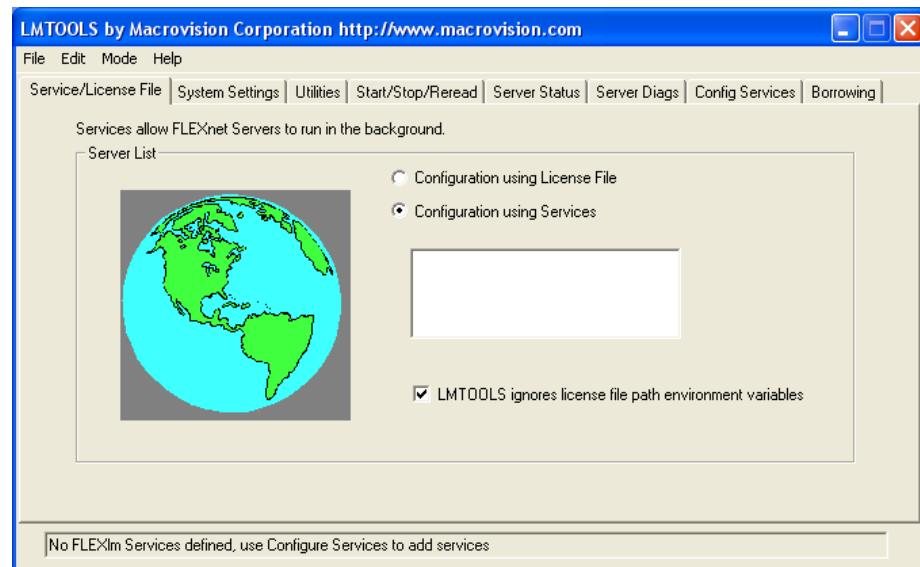
1. Before installation of the license server, the Dymola license file (`filename.lic`) may have to be updated with the actual name (or IP-number) of the server, if the license file contains a line identifying the server:

```
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 (x86)\Dymola 2015 FD01\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**.

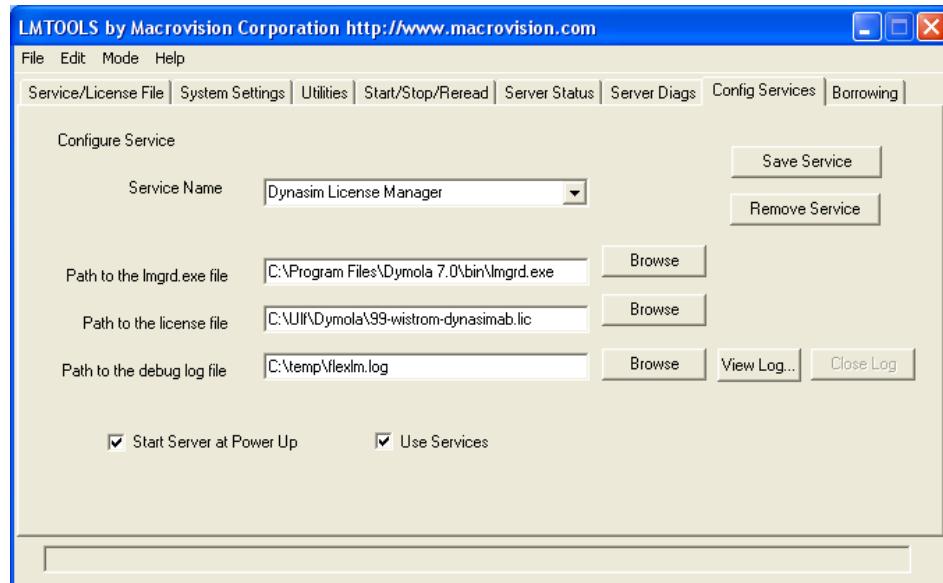
#### License server setup.



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 2015 FD01\bin\lmgrd.exe`.
  - c. Enter the path to your server license file.

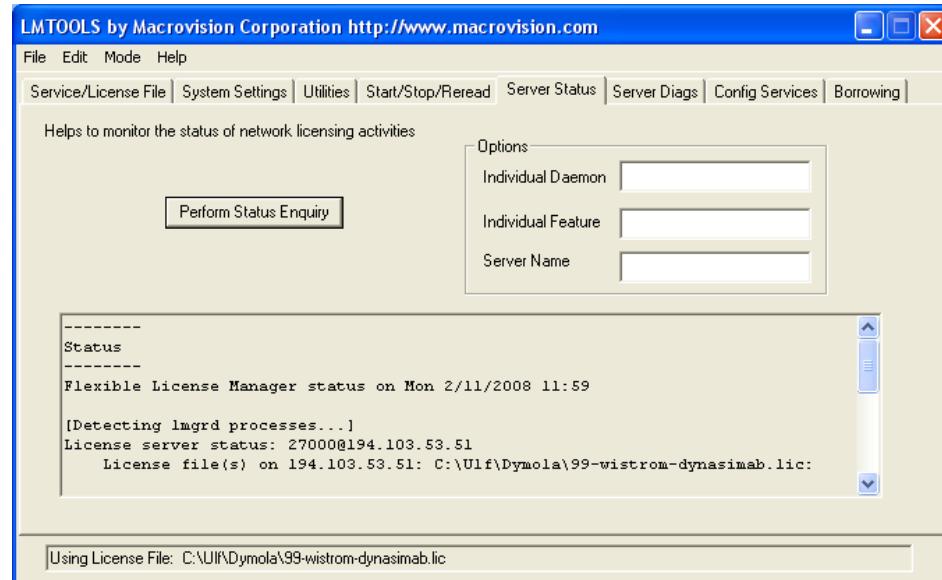
- d. Enter the path to a debug log file (anywhere you want).
- e. Enable **Use Services** and then **Start Server at Power Up**.
- f. Click on **Save Service**. Click on **Yes** to confirm.

**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\Dynamic\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.

### 6.3.3 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.

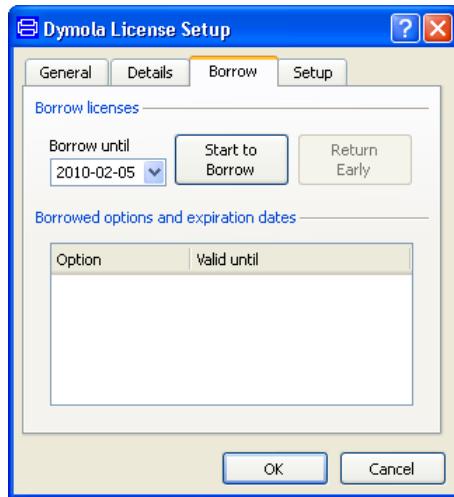
The license server keeps the borrowed license checked out. The client application automatically uses the local secured data file to do checkouts during the borrowing period. Upon the expiration of the borrowing period or the early return of a borrowed license, the local data file no longer authorizes 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

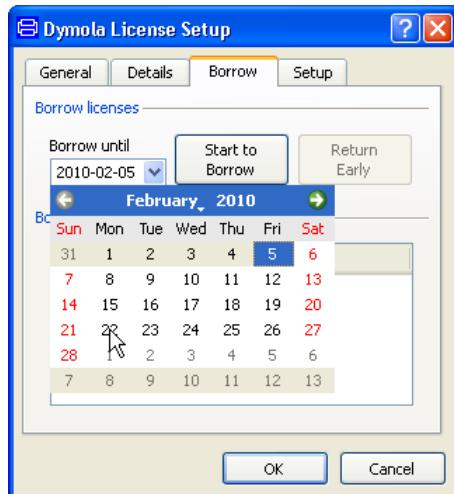
License borrowing and early returns are performed from Dymola.

In order to borrow, do the following:

1. While Dymola is connected to the server, use the command **Help > License...**, and select the **Borrow tab**.

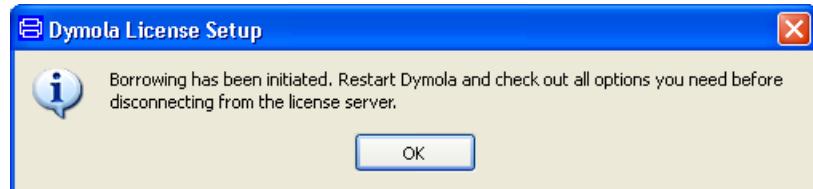


2. Select an end date, either by changing the date in the input field for **Last date borrowed** or by clicking on the arrow to display a calendar for selection of date. Clicking the arrow will display:

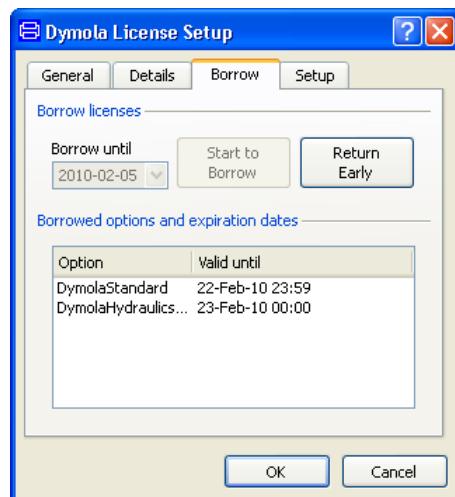


Here the possible selection of dates is clearly visible. Clicking on a date will change the input field to that date.

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.

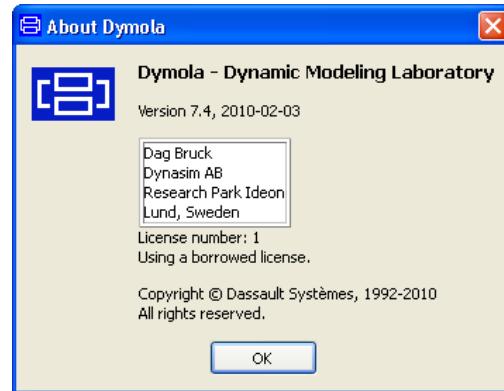
Please be careful not to open libraries/options that might be needed for others unless you really intend to do so. (Borrowing an option only available for one user only might not be appreciated by others.)

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.

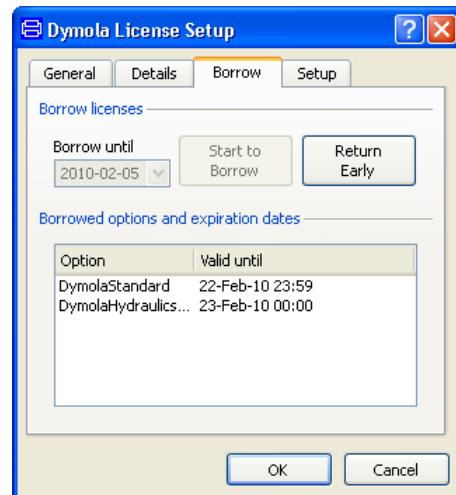
## Running Dymola

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 **Help > About Dymola**:

**Borrowing period in  
About dialog.**



Most information is given using the command **Help > License...**, in the **Borrow** tab.

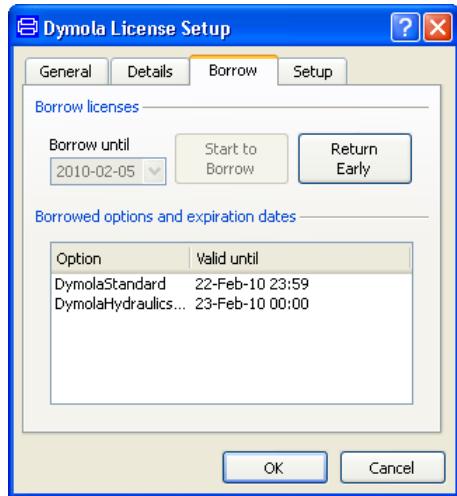


### **Returning a license before expiration of borrowing (early return)**

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 **Help > License...**, 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.

## 6.4 Dymola License Server on Linux

**This section refers only to the Linux version of Dymola.**

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 11.11. 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 auxxf 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.

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”.

Full details of FLEXnet license server installation can be found in the FLEXnet User’s Manual, which can be downloaded from [www.flexera.com](http://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 a 64 bit Dymola 2015 FD01, is:

```
/opt/dymola-2015FD01-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 a 64 bit Dymola 2015 FD01, is:

```
/opt/dymola-2015FD01-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.

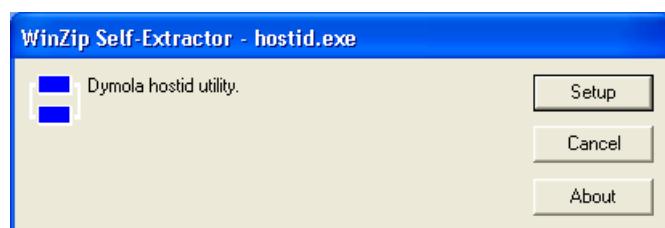
---

## 6.5 Utility programs

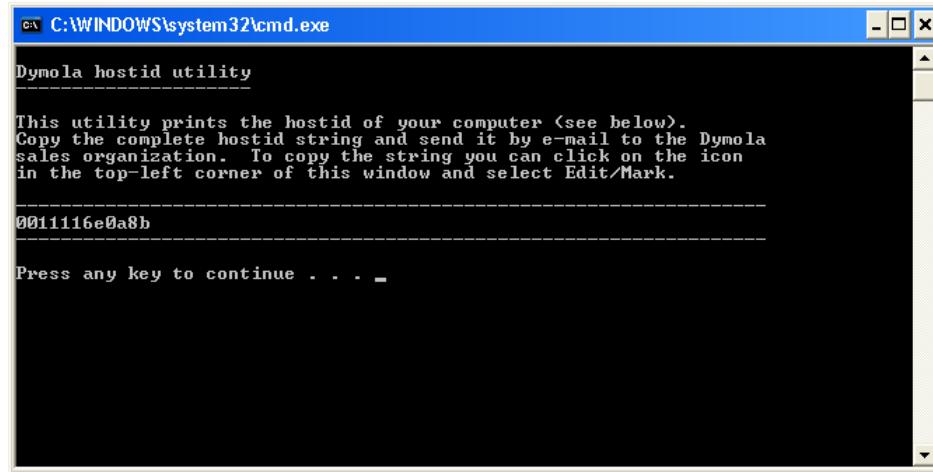
### 6.5.1 Obtaining a host id

To be able to easily 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 demo is installed, the host id can also be found using Dymola, please see section “Obtaining a host id” on page 633.)

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.

---

## 6.6 System requirements

### 6.6.1 Hardware requirements

- At least 1 GB RAM
- At least 400 MB disc space

### 6.6.2 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 2-4 GB of RAM for 32-bit architecture and 3-6 GB of RAM for 64-bit architecture.

## 6.6.3 Software requirements

### Microsoft Windows

Dymola 2015 FD01 is supported, as 32 or 64 bit application, on Microsoft Windows 7 and Windows 8.1. 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.

Concerning Windows XP and Windows Vista, a 32-bit version is available on request. Dymola 2015 FD01 is the last version supporting Windows XP and Windows Vista.

### Compilers

*Please note* that for the Windows platform, Microsoft C/C++ compiler, or a GCC compiler, must be installed separately. The following compilers are supported for Dymola 2015 FD01 on Windows:

Free editions:

- Visual Studio 2008 Express Edition (9.0)
- Visual C++ 2010 Express (10.0)
- Visual Studio 2012 Express Edition (11.0)
- Visual Studio 2013 Express Edition (12.0)

Professional editions:

- Visual Studio 2005 (8.0)
- Visual Studio 2008 (9.0)
- Visual Studio 2010 (10.0)
- Visual Studio 2012 (11.0)
- Visual Studio 2013 (12.0)

GCC compilers

- MinGW, with a GCC version compatible with 4.8.

Note – the GCC compiler has some limitations, and a demand for an add-on, please see section “GCC compiler” on page 629.

### Dymola license server

For a Dymola license server on Windows, all files needed to set up and run a Dymola license server on Windows, except the license file, are available in the Dymola distribution. (This includes also the license daemon, where Dymola presently supports FLEXnet Publisher version 11.11. A recent version is part of the Dymola distribution.)

### Linux

Dymola 2015 FD01 is supported on SUSE Linux (Release 11), 32-bit and 64-bit, with gcc version 4.3.4, and compatible systems.

In addition to gcc, the model C code can also be compiled by clang. To change compiler, change the variable CC in

```
/opt/dymola-2015FD01-<architecture>/insert/dsbuild.sh
```

<architecture> is i586 for 32-bit application, x86\_64 for 64 bit-application; for example for the 64-bit application the path is:

```
/opt/dymola-2015FD01-x86_64/insert/dsbuild.sh
```

Dymola 2015 FD01 is supported as a 32-bit and 64 bit application on Linux.

#### Notes

- For rendering of jpg files, libjpg62 must be installed.
- 32-bit compilation might require explicit installation of 32-bit libc. E.g. on Ubuntu:  
sudo apt-get install g++-multilib libc6-dev-386

#### Note on Optimization library

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.

#### Dymola license server

For a Dymola license server on Linux, all files needed to set up and run a Dymola license server on Linux, except the license file, are available in the Dymola distribution. (This includes also the license daemon, where Dymola presently supports FLEXnet Publisher version 11.11. This version is part of the Dymola distribution.)

---

## 6.7 License requirements

### 6.7.1 General

This description covers the license requirements for different features; for description of the features themselves, please see relevant documentation in “Dymola User Manual Volume 2”, chapter “Other 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:

- **Demo** license
- Licenses for academia (**Academic Learn/Innovate, Student Learn/Innovate**)

Some limitations may apply to those; in particular the demo license is limited.

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.

## 6.7.2 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.

## 6.7.3 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.)

### 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.
  - If you want to use other models as well, you need the **Source Code Generation** license.

## 6.7.4 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.)

### **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 “Other simulation environments”.

## **6.7.5        Licenses for executing/importing/using code**

### **Executing a Dymola model on a machine without a Dymola license**

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 **Standard Runtime 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.)

## **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 **Standard Runtime Configuration** license, and the environment variable DYMOLA\_RUNTIME\_LICENSE is set to the path where the license is located.
- 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 `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).

## **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:
  - The machine you import to must have a **Standard Runtime 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.)

**Important!** If you have a Dymola license on the machine, the environment variable must still be set.

- The FMU is from another vendor. For this case, please see the documentation for that vendor; there are no additional requirements from Dymola.

## **6.7.6      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**

- Automotive Demos (in the menu **File > Demos**)
- DataFiles
- Design.Experimentation
- Design.Validation
- DymolaCommands
- Modelica Reference
- Modelica Standard Library
- Modelica\_DeviceDrivers
- Modelica\_LinearSystems2
- Modelica\_StateGraph2
- Modelica\_Synchronous
- Plot 3D
- UserInteraction
- VehicleInterfaces

### **Libraries in Dymola library menu that require additional licenses when for example running more advanced examples**

- Design.Calibration – requires **Model Calibration** license when running more than one tuner.
- Design.Optimization – requires **Design Optimization** license when running more advanced examples. Note that there is a newer library available for optimization.

### **Libraries in Dymola library menu always requiring additional licenses**

Libraries in the Dymola library menu not listed above always require additional licenses.

## **6.7.7 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](http://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.

---

# **6.8 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.

## **6.8.1 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 **Help > License > 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 2015 FD01 64-bit is used on a 64-bit computer):

```
cd \Program Files (x86)\Dymola 2015 FD01\bin64  
dymola.exe /FLEXlmDiag
```

On Linux the command will be:

```
dymola /FLEXlmDiag
```

The additional information will in many cases be helpful in correspondence with support.

### **License server**

Correct operation of the license server should be verified with lmtools.exe, see “Installing the license server” on page 647. 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**

#### **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 **Help > License....**
- 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 advice that you in 64-bit Dymola generate a 64-bit Dymosim.exe using the flag Advanced.CompileWith64=2.

(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); dymosim.exe compiled as 32-bit, dymosim.dll, dymosim with DDE server, and FMUs compiled as 64-bit.
- 1 All applications above compiled as 32-bit.
- 2 All applications above compiled as 64-bit.

Dymosim as OPC server can presently only be compiled as 32-bit, setting Advanced.CompileWith64=2 is not supported when compiling dymosim as OPC server.)

### **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.

## **6.8.2 Compiler problems**

### **Test compiler button and error messages**

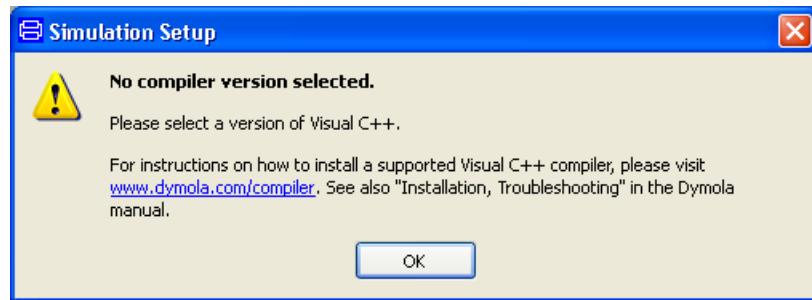
The compiler used to compile the C code generated by Dymola into executable code for simulation is set in the **Compiler** tab using the command **Simulation > Setup...**, see chapter “Simulating a model”, section “Editor command reference – Simulation mode”, sub-section “Main window: Simulation menu”, command “Simulation > Setup...”.

Some potential problems can be found by pressing the **Test compiler** button in the Compiler tab (see above). Any warning messages indicate problems that need to be resolved before translating a model. Pressing the button performs a number of tests:

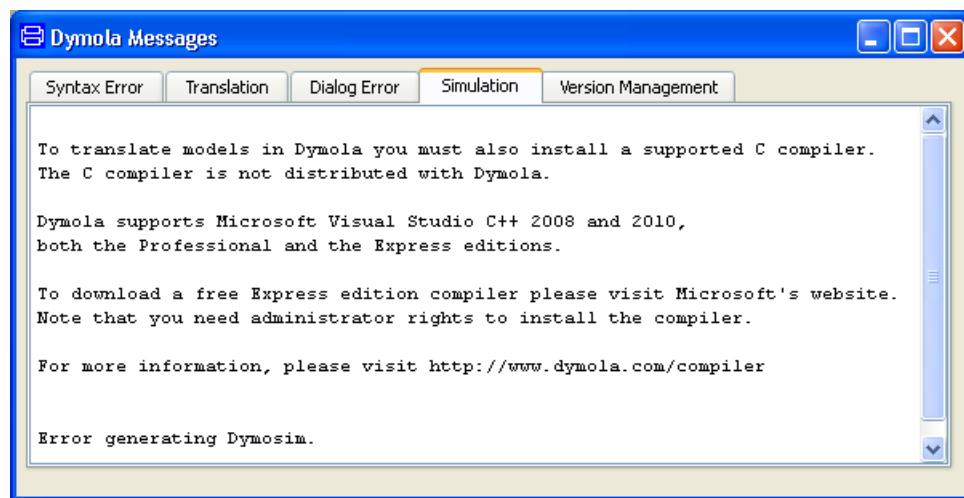
- Validates the DOS environment.
- Check the Dymola installation for run-time libraries.
- Verifies that the selected VC++ directory contains a valid compiler.
- Validates that the compiler can compile by executing a small test model.

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:



### Dymola webpage for compiler issues

A web page is available for Dymola compiler issues: [www.Dymola.com/compiler](http://www.Dymola.com/compiler). This page contains the following information:

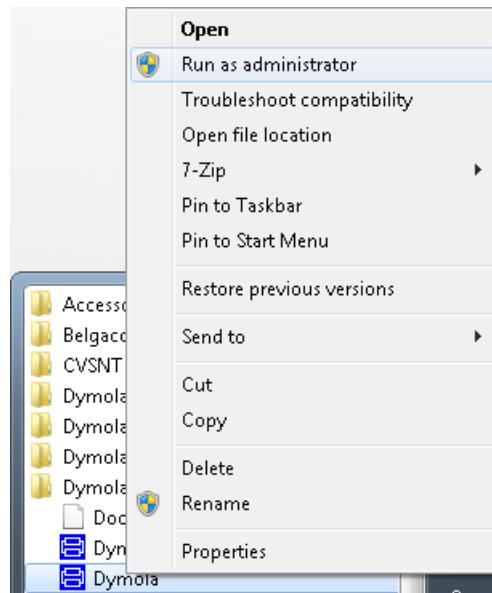
- Links to download compilers.
- Extract from the documentation concerning installation and troubleshooting of compilers.

A link to this page is presented when detecting any error using the **Find compiler** button and in the build log when no compiler is installed/selected.

### Dymola on Windows 7

Note that Visual Studio performs parts of its installation the first time it is run. This must be performed in order to use Visual Studio with Dymola, a step that requires administrator privileges. The rights can be elevated by running Dymola as administrator the first time.

This is done by right-clicking the Dymola icon in the Windows Start menu and selecting **Run as administrator**.



To validate this, any model (e. g. a demo) should be opened and translated in Dymola.

### 6.8.3 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 2015 FD01\mfiles`, `Dymola 2015 FD01\mfiles\traj` and `Dymola 2015 FD01\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 manual "Dymola User Manual Volume 2", chapter

“Other Simulation Environments”, section “Using the Dymola-Simulink interface”, sub-section “Graphical interface between Simulink and Dymola”.

- 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. Un-checking 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  $2\pi$  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.

## 6.8.4 Change of language

Dymola is available in Japanese. Sometimes the user wants to change the language after installation. This is done by changing the components of the installation. Please see section “Changing the setup of Dymola” on page 643.

You must restart Dymola after changing the setup.

## 6.8.5 Other Windows-related problems

### Starting the installation

The installation normally starts automatically when you insert the distribution CD. If auto start has been disabled, please start D:\dymola.msi (assuming your CD drive is labeled D) from Windows Explorer by double-clicking on the file or use the **Start** button in Windows, select **Run**, enter D:\dymola.msi 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.

# 7 Index

## 6

64-bit  
dymosim, DDE, FMU, [520](#)

## A

about, [331](#)

absolute tolerance, [523](#)

acausal modeling, [112](#)

algebraic loops

nonlinear, [618](#)

algorithms, [117](#), [472](#), [483](#), [522](#)

absolute tolerance, [523](#)

Adams, [523](#)

A-stable, [525](#)

BDF, [524](#)

Cerk23, [526](#)

Cerk34, [526](#)

Cerk45, [526](#)

DASSL, [526](#)

dense output, [524](#)

Dopri45, [526](#)

Dopri853, [526](#)

Esdirk23a, [526](#)

Esdirk34a, [526](#)

Esdirk45a, [526](#)

Euler, [526](#)

Explicit Euler (inline integration), [616](#)

fixed order, [524](#)

global error, [523](#)

Godess, [525](#)

Implicit Euler (inline integration), [616](#)

Implicit Runge Kutta (inline integration), [616](#)

LSODAR, [526](#)

Mixed explicit/implicit Euler (inline integration), [616](#)

Radau IIa, [526](#)

relative tolerance, [523](#)

Rkfix2, [526](#)

Rkfix3, [526](#)

Rkfix4, [526](#)

Sdirk34hw, [526](#)

stability boundary, [525](#)

stiff, [525](#)

Trapezoidal method (inline integration), [616](#)

variable order, [524](#)

variable step size, [524](#)

align

objects, [312](#)

to gridlines, [320](#)

angle for elliptical arc, [186](#)

**animate**, [505](#)  
 together, [505](#)  
**animated GIF**, [299](#), [463](#)  
**animation**, [35](#), [376](#)

- backstep, [498](#)
- continuous run, [499](#)
- menu, [497](#)
- pause, [498](#)
- reverse, [498](#)
- rewind, [498](#)
- run, [498](#)
- setup, [499](#)
- step forward, [499](#)
- time reduced, [300](#)
- visual shapes, [377](#)

 window. *See* animation window

**animation window**, [34](#), [376](#)

- moving view, [35](#), [445](#)
- pan view, [35](#), [445](#)
- roll view, [35](#), [445](#)
- rotate view, [35](#), [445](#)
- scroll view, [35](#), [445](#)
- tilt view, [35](#), [445](#)
- zooming, [35](#), [445](#)

**annotations**, [195](#)

- display of, [337](#)
- expanding/collapsing, [195](#)
- graphical, [313](#)
- parameter, [267](#)
- variable, [267](#)

**array**

- defining an array, [52](#)
- of connectors, [255](#)

**arrow keys**, [163](#)

**assignment**, [117](#)

- interactive, [534](#)

**A-stable**, [525](#), [526](#)

**auto-format**, [203](#), [338](#)

**automatic manhattanize**, [182](#)

**AVI (Audio-Video Interleaved)**, [299](#), [463](#)

**B**

**base class**, [153](#), [216](#), [345](#)

**batch execution**, [553](#)

**bitmap images**, [189](#)

**BMP**

- graphical object, [189](#)

**borrowing**

- general, [651](#)
- on Linux, [656](#)

**on Windows**, [651](#)

**bound parameter**, [85](#)

**boundary**, [152](#)

**bounds checking**, [600](#)

**bracket handling**, [204](#)

**bring to front**, [312](#)

**browser**

- component, [26](#), [154](#)
- information, [159](#), [388](#)
- package, [25](#), [154](#), [320](#)
- variable, [372](#), [504](#), [506](#)

**built-in commands**, [557](#)

**built-in functions**, [557](#)

**bus signal tracing**, [384](#)

**C**

**C compiler**, [629](#)

**Call Function**, [344](#)

**cascading style sheet**, [300](#)

**change**

- attributes, [316](#)
- class, [247](#)

**characters**

- national, [233](#)

**chattering**, [603](#), [607](#)

**check**, [314](#)

- model - checkModel (built-in function), [560](#)
- model - normal, [314](#)
- model - with simulation, [314](#)
- Modelica text - syntax, [205](#)
- of built-in functions, [315](#)

**checkModel**, [560](#)

**circles**, [186](#)

**class**

- add base class, [247](#)
- change, [247](#)
- change class of component, [246](#)
- constraining, [348](#)
- description, [112](#), [217](#), [223](#), [316](#)
- documentation, [70](#), [215](#)
- extend, [82](#), [345](#)
- parameters. *See* class - replaceable classes (local)
- replaceable classes (local), [116](#), [247](#), [268](#)
- set component scale factor, [317](#)

**classDirectory**, [568](#)

**classes**

- creating, [167](#)
- used, [327](#)

**clear all**, [295](#)

clear log, [307](#), [465](#)  
clearPlot, [570](#)  
clipboard, [299](#), [463](#)  
code completion, [202](#)  
color, [193](#), [354](#)  
  coding, [201](#)  
  defining custom colors, [354](#)  
  of plotted variables, [490](#)  
  palette, [353](#)  
  selecting, [353](#)  
  true RGB, [354](#)

Comma Separated Values. *See* file extensions : .csv command  
  command line arguments for Dymosim, [529](#)  
  input line, [378](#), [380](#), [450](#)  
  log, [378](#), [379](#), [448](#), [477](#)  
  log file, [379](#), [449](#)  
  log pane, [378](#), [379](#), [448](#)  
  window, [378](#), [477](#)

command window, [13](#). *See* command - window commands  
  built-in, [557](#)  
  edit, [325](#)

comment  
  attribute. *See* class description  
comment out, [205](#)  
comment selection, [338](#), [517](#)  
comments, [197](#), [212](#)  
compiler, [629](#)  
  GCC (on Linux), [644](#)  
  GCC (on Windows), [629](#)  
  installation, [629](#)  
  Microsoft Visual C++, [629](#)  
  on Linux, [480](#)  
  setup, [479](#)  
  troubleshooting, [667](#)

component, [17](#)  
  **attributes**, [347](#)  
  browser, [26](#), [154](#)  
  conditional, [256](#)  
  library, [62](#)  
  reference, [88](#), [212](#)  
  replaceable, [247](#)  
  size, [152](#), [168](#), [317](#)

computer algebra, [114](#)

Concurrent Version System. *See* model management : version control

concurrent zooming. *See* plot window zooming

conditional  
  components, [256](#)  
  connectors, [256](#)

connection, [109](#), [163](#), [309](#)  
  creating using Smart Connect, [178](#)  
  creating without using Smart Connect, [182](#)  
  hiding graphical connection, [183](#)

connections, [177](#)

connector, [109](#)  
  array, [255](#)  
  conditional, [256](#)  
  expandable, [254](#)  
  nested, [251](#)  
  overlapping, [252](#)  
  protected, [148](#)  
  stacked, [252](#)  
  stream, [114](#)  
  visibility, [148](#)

constraining  
  class, [348](#)  
  clause, [348](#)

context menu, [26](#), [163](#), [169](#), [182](#)  
  animation window, [511](#)  
  command window - command input line, [515](#)  
  command window - command log pane, [513](#)  
  component browser, [343](#)  
  components, [346](#)  
  coordinate system boundary, [350](#)  
  edit window Diagram layer, [332](#)  
  edit window Documentation layer, [333](#)  
  edit window Icon layer, [332](#)  
  edit window Modelica Text layer, [335](#)  
  edit window Used Classes layer, [341](#)  
  graphical objects, [351](#)  
  message window, [519](#)  
  plot window, [507](#)  
  plot window – curve and legend, [508](#)  
  plot window - signal operators, [509](#)  
  plot window - text objects, [510](#)  
  script editor, [516](#)  
  table window, [511](#)  
  variable browser - signals, [506](#)  
  variable browser nodes, [504](#)  
  visualizer window, [512](#)  
    while connecting, [350](#)

continue line, [534](#)  
continue simulation, [468](#)  
continuous run of animation, [499](#)

conversion  
  offset, [281](#)

coordinate system, [152](#), [317](#)  
  boundary, [152](#)  
  specification, [152](#)

copy, [308](#)  
Copy Call, [345](#)

copy to clipboard, [299](#), [409](#), [463](#)  
corner radius, [185](#)  
create  
    link, [225](#), [457](#)  
    plot window using built-in function, [570](#)  
    script, [464](#)  
createPlot, [570](#)  
creating a model, [64](#)  
crossing function, [121](#)  
CSS, [300](#)  
current selection, [162](#)  
cursor  
    position - display, [320](#)  
cut, [308](#)  
CSVs. *See* model management : version control

## D

DAE, [113](#)  
    high index, [114](#)  
daemon  
    license, [647](#)  
    vendor, [647](#)  
data flow modeling, [113](#)  
DataFiles package, [278](#)  
DC motor, [54](#)  
DDE server, [480](#), [520](#)  
debug monitor, [608](#)  
debugging models, [477](#), [583](#)  
declaration  
    conditional, [256](#)  
    parameter, [262](#)  
    variable, [262](#), [318](#)  
declarations, [39](#)  
default graphics, [190](#)  
delete, [309](#)  
delete key, [164](#)  
deleting objects, [164](#)  
demo, [24](#), [294](#)  
dense output, [524](#), [526](#)  
der(...) operator, [38](#)  
description of class, [112](#), [217](#), [223](#), [316](#)  
diagram layer, [35](#), [148](#), [326](#), [392](#)  
    in simulation, [373](#), [402](#)  
    moving view, [161](#), [401](#)  
    window, [373](#), [402](#)  
    zoom factor, [326](#)  
    zooming, [161](#), [401](#)  
differential-algebraic equations. *See* DAE

discrete equations, [119](#)  
displayUnit, [174](#), [279](#), [283](#), [420](#)  
displayunit.mos, [279](#), [421](#)  
displayunit\_us.mos, [279](#), [421](#)  
document, [558](#)  
documentation  
    editor. *See* editor - documentation  
    for web publication, [232](#)  
    generating HTML documentation, [231](#)  
    HTML (of classes), [215](#)  
    manuals, [103](#)  
    of class, [70](#), [215](#)  
    of model, [70](#), [215](#)  
    of simulation, [90](#), [452](#)  
    printable, [232](#)  
    templates, [232](#)  
documentation layer, [149](#), [215](#), [326](#)  
    content, [216](#)  
    edit, [223](#)  
    HTML source code, [219](#)  
double-clicking, [163](#), [182](#)  
dragging, [168](#)  
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  
dsmodel, [589](#)  
dsu.txt. *See* Dymosim files : dsu.txt  
duplicate  
    class, [345](#)  
    objects, [309](#)  
DXF files, [443](#)  
Dymola  
    32-bit application on Linux, [644](#)  
    32-bit application on Windows, [624](#)  
    64-bit application on Windows, [624](#)  
Dymola license server  
    on Linux, [656](#)  
    on Windows, [647](#)  
Dymola Main window, [144](#)  
Dymosim, [520](#)  
    command line arguments, [529](#)  
    DDE server, [520](#)  
    OPC server, [520](#)  
Dymosim files, [520](#)  
    dsfinal.txt, [561](#)  
    dsin.txt, [521](#), [529](#), [561](#)  
    dslog.txt, [520](#)  
    dsres.mat, [521](#), [522](#)  
    dsu.txt, [522](#)

dymtools, [522](#)  
dynamic state selection. *See* state selection  
dynamic typing, [267](#)

## E

each, [365](#)  
edit  
    angle for elliptical arc, [186](#)  
    documentation (of class), [71](#), [223](#)  
    documentation (of simulation), [90](#), [452](#)  
    documentation layer, [223](#)  
    equation or expression (in command log), [459](#)  
    equation or expression (in documentation layer), [334](#)  
    function call, [212](#)  
    graphical annotations, [313](#)  
    graphical objects, [184](#)  
    HTML source code, [230](#)  
    Modelica text layer, [193](#)  
    text object, [186](#)  
edit annotation  
    transformation extent, [349](#)  
Edit combined, [360](#)  
edit menu  
    align, [312](#)  
    copy, [308](#)  
    cut, [308](#)  
    delete, [309](#)  
    draw shapes, [314](#)  
    duplicate, [309](#)  
    edit annotation, [313](#)  
    edit attributes, [316](#)  
    find - components in diagram layer, [309](#)  
    find and replace, [310](#)  
    flip horizontal, [313](#)  
    flip vertical, [313](#)  
    go to, [311](#)  
    manhattanize, [312](#)  
    options, [319](#)  
    order, [312](#)  
    paste, [308](#)  
    redo, [308](#)  
    rotate, [312](#)  
    select all, [309](#)  
    smooth, [312](#)  
    Split Model, [313](#)  
    undo, [308](#)  
    variables, [318](#)  
edit window, [25](#), [146](#), [163](#)  
editing commands, [325](#)  
editor  
    documentation (of class), [71](#), [223](#)  
    documentation (of simulation), [90](#), [452](#)  
    Modelica text, [193](#)  
efficiency  
    improving simulation, [604](#)  
ellipses, [185](#)  
encapsulated, [316](#)  
Enhanced Metafile Format, [299](#), [463](#)  
environment variables  
    DYMOLA, [645](#)  
    DYMOLA\_RUNTIME\_LICENSE, [635](#)  
    DYMOLAPATH, [645](#)  
    DYMOLAWORK, [626](#), [645](#)  
    LM\_BORROW, [656](#)  
    MODELICAPATH, [637](#), [645](#)  
equations  
    manipulated, [589](#)  
    synchronous, [118](#)  
eraseClasses, [569](#)  
error messages, [382](#), [585](#)  
    report function call in, [478](#)  
errors, [583](#)  
    rules for avoiding, [583](#)  
evaluation  
    of parameters, [460](#)  
evaluation of parameters, [131](#)  
event, [121](#)  
    events and chattering, [607](#)  
    logging, [588](#)  
examples  
    bounds checking, [600](#)  
    DC motor, [54](#)  
    Furuta pendulum, [94](#)  
    ideal diode, [122](#)  
    initialization of discrete controllers, [133](#)  
    model template using replaceable components, [259](#)  
    motor drive, [108](#), [113](#)  
    motor model, [109](#)  
    pendulum, [38](#), [129](#)  
    plot - createPlot, [572](#)  
    plot - plot (built-in function), [574](#)  
    plot - plotArray, [575](#)  
    plot - plotText, [579](#)  
    polynomial multiplication, [117](#)  
    rectifier circuit, [123](#)  
    result files in variable browser - keep between  
        simulations, [400](#)  
    results of translating Modelica models, [590](#)  
    sampled data system, [118](#)  
    start values, [619](#)  
    stuck simulation, [603](#)

exit  
     Dymola, [307](#)  
 expandable connectors, [254](#)  
     signal tracing, [384](#)  
 expanding/collapsing  
     annotations, [195](#)  
     components, [195](#)  
     connections, [195](#)  
 experiment setup. *See* simulation setup  
 export  
     animation, [299](#), [463](#)  
     Encrypted File, [307](#)  
     Encrypted Total Model, [307](#)  
     HTML, [300](#)  
     image, [299](#), [463](#)  
     model to other simulation environments, [467](#)  
     plot as image, [573](#)  
     to clipboard, [299](#), [463](#)  
     visualizer image, [513](#)  
 export options  
     need for, [529](#)  
 exportAnimation, [581](#)  
 exportDiagram, [582](#)  
 exportDocumentation, [582](#)  
 exportEquations, [582](#)  
 exportIcon, [582](#)  
 exportInitial, [560](#)  
 expression  
     interactive, [533](#)  
 extend, [82](#), [345](#)

**F**

fast sampling, [603](#)  
 fault-finding, [583](#), [604](#), [608](#)  
 favorite package, [164](#)  
 file  
     import (alternatives), [277](#)  
 file data  
     use of, [277](#)  
 file extensions  
     .avi, [581](#)  
     .AVI, [299](#), [463](#)  
     .bmp, [189](#)  
     .cab, [636](#)  
     .csv, [277](#), [358](#), [505](#)  
     .emf, [299](#), [463](#)  
     .gif, [189](#), [463](#)  
     .jpeg, [189](#)  
     .jpg, [513](#)  
     .mat, [277](#), [357](#), [505](#)  
     .mo, [292](#), [294](#)  
     .mof, [589](#)  
     .mos, [324](#), [421](#), [540](#)  
     .msi, [636](#)  
     .png, [189](#), [299](#), [463](#), [513](#)  
     .svg, [299](#)  
     .txt, [277](#), [358](#), [505](#)  
     .wrl, [299](#), [463](#), [581](#)  
     .x3d, [581](#)  
     .xpm, [513](#)  
     .svg, [463](#)  
 file formats  
     .x3d, [299](#), [463](#)  
     .xhtml, [300](#), [463](#)  
 file menu  
     change directory, [297](#)  
     check, [314](#)  
     clear all, [295](#)  
     clear log, [307](#), [465](#)  
     copy to clipboard, [299](#), [463](#)  
     demos, [294](#)  
     duplicate class, [293](#)  
     exit, [307](#)  
     export animation, [299](#), [463](#)  
     export Encrypted File, [307](#)  
     export Encrypted Total Model, [307](#)  
     export HTML..., [300](#)  
     export image, [299](#), [463](#)  
     export to clipboard, [299](#), [463](#)  
     generate script, [464](#)  
     import FMU, [462](#)  
     libraries, [294](#)  
     load, [293](#)  
     new, [291](#), [292](#)  
     open, [293](#), [462](#)  
     print, [298](#), [462](#)  
     print preview, [297](#)  
     recent files, [307](#)  
     save, [294](#), [462](#)  
     save all, [294](#), [462](#)  
     save as, [294](#)  
     save log, [307](#), [464](#)  
     save total, [294](#)  
     search, [295](#)  
     version, [294](#)  
 fill  
     color, [193](#)  
     gradient, [355](#)  
     pattern, [355](#)  
     style, [355](#)  
 final, [365](#)

find  
     class or component, 295  
     components in diagram layer, 309  
     text (inside a class), 310  
 fixed attribute of start value, 53, 126, 175  
 fixed order, 524  
 flat Modelica text, 151  
 FLEXnet, 647  
     license server options file, 655  
 flip  
     horizontal, 313  
     vertical, 313  
 flow prefix, 110  
 FMI (Functional Mock-up Interface)  
     see FMU, 566  
 FMU  
     export model to FMU, 467, 566  
     import FMU, 462, 561  
 font  
     size, 40, 204  
 font size, 188, 224, 456  
     base, 321  
     restrict minimum font size, 320  
     save base font size, 322  
 formatting  
     Modelica text, 203  
 frame rate, 300  
 function, 117  
 function call  
     edit, 212  
     insert, 210, 338, 361, 366, 451, 515, 518, 530  
 functions  
     built-in, 557  
 Furuta pendulum, 94

## G

GCC (on Linux), 644  
 GCC (on Windows), 629  
 generate  
     script, 464  
 GenerateBlockTimers, 611  
 GenerateTimers, 613  
 getClassText, 582  
 getExperiment, 560  
 getLastError, 569  
 GIF  
     animated, 299, 463  
     graphical object, 189  
 global error, 523

## H

Godess solvers, 525  
 graphical annotations, 313  
 graphical attributes, 193  
 graphical objects, 184, 376  
     default, 190  
     insert point, 352  
     remove point, 352  
 grid, 152, 153, 163, 182, 185, 317  
 gridlines, 163, 314  
 guidelines  
     for model development, 583

## I

handles, 162  
 hardware-in-the-loop simulation, 20  
 help, 558  
 help menu  
     about, 331  
     documentation, 329  
     Dymola website, 329  
     what's this, 329  
 high index DAE, 114  
 high resolution images, 299, 463  
 highlight syntax, 40, 338  
 Highlight syntax, 201, 205  
 HIL, 592, 596. *See* hardware-in-the-loop simulation  
 host id, 633  
 HTML, 300  
     documentation of classes, 215  
     export, 300  
     external references, 306  
     generating HTML documentation, 231  
     hyperlinks, 215, 225, 457  
     Microsoft Word template, 302  
     online documentation, 301, 306  
     options, 300  
     setup, 300  
 hybrid model, 117  
 hybrid modeling, 117  
 hybrid system, 588  
 hyperlink, 225, 457

## Icon

icon layer, 147, 326  
     moving view, 161  
     zooming, 161  
 if-expression, 121  
 images

bitmap, 189  
high resolution, 299, 463  
inserting images in parameter dialog, 269  
inserting in command log, 458  
inserting in diagram or icon layer, 189  
inserting in documentation layer, 226

import  
  file data (alternatives), 277  
  FMU - importFMU (built-in function), 561  
  initial, 468  
  start values - importInitial (built-in function), 561  
  start values - importInitialResult (built-in function), 561  
  statement, 534

importFMU, 561

importInitial, 561

importInitialResult, 561

incorrect license file format, 635

indenting, 203

Info  
  button, 220  
  menu entry, 220

info button, 29

information  
  browser, 159, 388

initialization of models, 125  
  over-specified, 617

initialized, 561

inline integration, 483, 614  
  algorithms, 616. *See* algorithms

insert  
  equation or expression (in command log), 458  
  equation or expression (in documentation layer), 227  
  function call, 210, 451, 515, 530  
  image in command log, 458  
  image in diagram or icon layer, 189  
  image in documentation layer, 226  
  link, 225, 457  
  Modelica text into MS PowerPoint, 233  
  statement, 210

Instability, 588

installation  
  environment variables, 645  
  license daemon, 647  
  license server on Linux, 656  
  license server on Windows, 647  
  Linux, 644  
  remote on Windows, 636  
  troubleshooting, 665  
  windows, 624

integration algorithms. *See* algorithms

integration methods. *See* algorithms

## J

Jacobian, 126, 129, 386, 589  
JPEG  
  graphical object, 189

## K

keyword  
  redeclare, 116

## L

layer, 146  
  diagram, 148, 326, 392  
  documentation, 149  
  icon, 147  
  Modelica text, 150  
  used classes, 151

library, 62  
  creating, 164  
  documentation, 215  
  libraries used, 529  
  list of libraries, 137  
  menu, 294  
  Modelica 2-compatible libraries, 636  
  shortcut to, 294  
  window, 158

library window, 13

libraryinfo.mos, 637

license  
  borrowing - general, 651  
  borrowing - on Linux, 656  
  borrowing - on Windows, 651  
  daemon, 647  
  demands, 660  
  early return - on Linux, 656  
  early return – on Windows, 654  
  error message, 665  
  expiration - settings, 639  
  export options, need for, 529  
  incorrect file format, 635  
  node-locked, 633  
  not found or invalid - settings, 639  
  preventing checking out - in Linux, 646  
  preventing checking out – in Windows, 639  
  requirements, 660  
  runtime, 635  
  runtime, need for, 529  
  server  
    on Linux, 656

on Windows, 647  
 server options file, 655  
 sharable, 632  
**line**, 185  
 arrow style, 354  
 color, 193  
 style, 353  
 style of plotted variables, 491  
 thickness, 354  
 thickness of plotted variables, 491  
**linear analysis menu**  
 bode plot, 332  
 full linear analysis, 332  
 linearize, 331  
 poles, 331  
 poles and zeros, 331  
 root locus, 332  
**linearize**, 469, 562  
**linearizeModel**, 562  
**link**  
 create, 225, 457  
 in error log to variables in model window, 587  
 insert, 225, 457  
 local, 226, 457  
**Linux**  
 compiler, 480  
 Dymola as 32-bit application, 644  
 Dymola license server, 656  
 installation, 644  
**list**, 562  
**listfunctions**, 562  
**load** (data files). *See* import  
**local**  
 link, 226, 457  
**local class reference**, 212  
**local host id**, 633  
**log**  
 command, 378, 379, 448, 477  
 command log file, 449  
**logging events**, 588

## M

**main window**, 13  
**manhattanize**, 69, 182, 312, 320  
**manipulated equations**, 589  
**marker style**  
 of plotted variables, 491  
**matching**  
 of variables, 270

**mathematical notation**  
 Modelica text layer, 198  
**Matlab**, 12, 521  
 Dymosim m files, 527  
 importing Dymola result files to Matlab, 521  
 importing Matlab files to Dymola, 358  
 loading Matlab files to Dymola, 277  
 saving result in Matlab format, 277, 357, 505  
**matrix**  
 defining a matrix, 52  
 editor, 357  
 equations, 115  
 handling of a large matrix, 361  
**menu**  
 animation, 497  
 diagram, 487  
**message window**, 382  
**methods**. *See* algorithms  
**m-file**, 527  
 dymbrowse, 522, 528  
 dymget, 521, 528  
 dymload, 521, 528  
 tcomp, 528  
 tcut, 528  
 tder, 528  
 tdiff, 528  
 tfigure, 528  
 tinteg, 528  
 tload, 522, 528  
 tloadlin, 528  
 tnhead, 528  
 tmindex, 528  
 tnlist, 522, 528  
 tplot, 522, 528  
 tplotm, 528  
 trajectory, 527  
 trange, 528  
 tsame, 528  
 tsave, 522, 528  
 tzoom, 528  
**Microsoft**  
 Excel, 505  
 PowerPoint, 233  
 Visual C++, 629  
 Windows installation, 624  
 Word 2003, 299, 463  
 Word 97, 299, 463  
 Word HTML template, 302  
**mixed-mode integration**, 614  
**mode**  
 modeling, 13  
 Modeling, 23, 144

pedantic compiler mode, [206](#), [474](#)  
simulation, [13](#)  
Simulation, [23](#), [145](#)

model, [107](#)  
    creating, [166](#)  
    debugging, [583](#)  
    documentation, [70](#), [215](#)  
    extending, [167](#)  
    hybrid, [117](#)  
    instability, [588](#)  
    partial, [111](#)  
    replaceable, [117](#)  
    split model, [239](#)  
    submodel, [239](#)  
    templates, [259](#)

model editor, [107](#)

model management, [322](#)  
    version control, [322](#)

Modelica, [19](#), [107](#), [193](#)  
    modelica URI scheme, [189](#), [226](#)

Modelica 2-compatible libraries, [636](#)

Modelica Reference, [137](#)

Modelica Standard Library, [54](#), [109](#), [136](#), [321](#)

Modelica text, [150](#), [193](#), [194](#), [327](#)  
    editor, [193](#)  
    flat. *See* flat Modelica text

Modelica text layer, [194](#)  
    bracket handling, [204](#)  
    code completion, [202](#)  
    color coding, [201](#)  
    comment out, [205](#)  
    comments, [212](#)  
    component reference, [212](#)

context menu, [201](#)

features, [200](#)

font size, [40](#), [204](#)

formatting, [203](#)

function calls, [210](#)

indenting, automatic, [203](#)

inserting statements, [210](#)

local class reference, [212](#)

mathematical notation, [198](#)

navigation, [198](#), [205](#)

syntax check, [205](#)

variable declaration, [207](#)

MODELICAPATH, [293](#)

modeling, [107](#)  
    acausal, [112](#)  
    hybrid, [117](#)

modeling mode, [13](#)

Modeling mode, [23](#), [144](#)

modifiers, [171](#), [176](#)

mouse button  
    left, [162](#), [163](#)  
    right, [163](#)

moving  
    animation view, [35](#), [445](#)  
    class, [346](#)  
    diagram layer view, [161](#), [401](#)  
    icon layer view, [161](#)  
    objects, [163](#)  
    plot view, [411](#)  
    plot window, [76](#)

multi-core support, [605](#)

My Documents, [626](#)

## N

national characters, [233](#)

nested connectors, [251](#)

new  
    model, connector etc., [291](#)  
    package, [292](#)

new script, [466](#)

New variable, [318](#)

node-locked license, [633](#)

noEvent(...) operator, [122](#), [614](#)

nonlinear, [618](#)

non-linear systems, [601](#)

nowindow, [554](#)

## O

ODE, [114](#)

OPC server, [480](#), [520](#)

open, [462](#)

open all, [504](#)

Open result, [498](#)

open script, [466](#)

openModel, [563](#)

order, [524](#)

ordering of objects, [312](#)

ordinary differential equations. *See* ODE

output interval, [471](#)

overlapping connectors, [252](#)

over-specified initialization problem, [617](#)

## P

package

browser, 25, 154, 320  
creating, 164  
favorite, 164  
hierarchy, 25  
local, 197  
storing of, 292

pan  
    animation view, 35, 445

parameter, 130, 170  
    annotations, 267  
    class parameters. *See* class - repaceable classes (local)  
    declaration, 112, 262  
    dialog, 28, 171  
    error, 261  
    evaluation, 131, 460  
    propagation, 85  
    structural, 460  
    type prefix. *See* type prefix  
values, 130  
values - save in model from simulation, 213, 395  
warning, 261

parameter dialog  
    activation of dialog entry for start values, 269  
    advanced use, 268  
    editing - advanced, 269  
    possible modifiers, 268  
    presentation of enumeration values, 269

partial models, 111

paste, 308  
    special, 233

pedantic compiler mode, 206, 474

pendulum, 38, 129

picking objects, 162

plot  
    (built-in function), 573  
    an overview of functionality, 404  
    Boolean signals, 408  
    create plot window using built-in function, 570  
    displaying a table instead of curves, 433  
    displaying signal operators, 421  
    enumeration signals, 408  
    export as image, 573  
    from large result files, 404  
    menu. *See* plot menu  
    parametric curves, 434  
    plot expressions, 429  
    plot window interaction, 405  
    save plot as image, 404  
    save settings between sessions, 404  
    save settings in script, 405  
    scripting, 405  
    selecting multiple curves, 432

setup menu. *See* plot setup menu  
window, 34. *See* plot window  
window options, 75

plot menu  
    delete diagram, 488  
    draw - text, 488  
    erase content, 487  
    new diagram, 487  
    new window - new plot window, 487  
    new window - new table window, 487  
    open result, 487  
    plot expression, 488  
    rescale all, 487  
    rescale diagram, 487  
    reset window layout, 487  
    setup. *See* setup menu  
    toggle grid, 488

plot setup menu, 406. *See* plot setup menu

plot window, 374  
    changing displayed unit of signals, 419  
    changing time unit of x-axis, 419  
    create tables from curves, 434  
    displaying a table instead of curves, 433  
    displaying signal operators, 421  
    duplicate diagram to new plot window, 408  
    insert plot and corresponding command in command  
        window, 438  
    inserting texts, 437  
    interaction, 405  
    moving view, 411  
    parametric curves, 434  
    plot expressions, 429  
    rescale. *See* plot window zooming  
    selecting multiple curves, 432  
    time line, 437  
    zooming. *See* plot window zooming

plot window zooming  
    horizontal (time) zooming, 415  
    rescale a diagram, 418  
    rescale a signal. *See* plot window zooming  
    rescale all diagrams in a plot window, 418  
    using the Range tab to zoom, 417  
    vertical zooming, 415  
    zooming in by dragging, 412  
    zooming in/out using Ctrl, 412  
    zooming out using context menus, 418  
    zooming with maximum y scaling, 413

plotArray, 575

plotArrays, 576

plotExpression, 576

plotSignalOperator, 577

plotSignalOperatorHarmonic, 578

plotText, 579  
png  
  export of image, 463  
PNG  
  graphical object, 189  
polygon, 185  
polynomial multiplication, 117  
pre(...) operator, 119  
pretty print, 203, 338  
print, 298, 462  
print preview, 297  
printable documentation, 232  
printPlot, 580  
profiling, 611  
Program Files, 626  
propagate, 364  
propagation of parameters, 85  
protected, 316  
  variables - storage of, 461

## Q

quit  
  Dymola, 307

## R

real-time simulation, 614  
  options, 482  
recent  
  files, 307  
  models, 326  
Recent Windows, 488  
rectangle  
  rounded corners, 185  
rectangles, 185  
redeclare keyword, 116  
redo, 308  
reformat selection, 338  
regular expressions, 296, 396  
relative tic mark, 397  
relative tolerance, 523  
rename, 346  
replace  
  text, 310  
replaceable  
  classes (local). *See* class - replaceable classes (local)  
  components, 247  
  variable, 365

replaceable model, 117  
rescale  
  plot. *See* plot window zooming  
reshaping objects, 152, 163  
result files in variable browser  
  closing result files, 398  
  default handling, 397  
  exporting selected simulation results, 401  
  exporting simulation results, 401  
  opening additional result files, 398  
  selecting what result files to keep, 398  
RGB, 354  
robot demo, 24  
roll  
  animation view, 35, 445  
root finder, 526  
root model, 155  
rotate, 312  
  animation view, 35, 445  
rounded corners, 185  
run script, 34, 307, 466  
RunScript, 548, 563  
runtime  
  license, need for, 529  
runtime license, 635

## S

save, 462  
  animation setup, 464  
  base font size, 322  
  default Modelica version, 322  
  graphical editor settings, 322  
  Modelica editor settings, 322  
  plot settings, 322  
  plot setup, 464  
  position/geometry of Dymola window, 322  
  settings, 464  
  variables, 464  
save all, 294, 462  
save as, 294, 462  
  results, 522  
save log, 307, 464  
save model, 307  
save total, 294  
savelog, 569  
saveSettings, 569  
script, 34, 92  
  create, 464  
  editor, 549

files, 529  
functions, 533  
generating a, 464  
new, 466  
open, 466  
run, 466  
script editor, 549  
scroll  
    animation view, 35, 445  
search  
    bus signals (expandable connectors), 384  
    class or component, 295  
    text (inside a class), 310  
selecting objects, 162, 309  
selections  
    of variables, 270  
send to back, 312  
set corner radius, 185  
setClassText, 564  
setting parameters, 46  
setup  
    animation, 499  
    compiler, 479  
    plot. *See* plot setup menu  
    simulation, 470  
sharable licenses, 632  
shell command  
    dymosim, 529  
shift key, 162, 182  
shortcut to library, 294  
Show Variables, 392  
SI units, 283  
signal operators, 421  
    definitions, 427  
signalOperatorValue, 580  
simulate, 468  
    model - simulateExtendedModel (built-in function), 563  
    model - simulateModel (built-in function), 564  
    model - simulateMultiExtendedModel (built-in function), 565  
simulate model, 43  
simulateExtendedModel, 563  
simulateModel, 564  
simulateMultiExtendedModel, 565  
simulation, 33, 389  
    documentation, 90, 452  
    efficiency, 604  
    interval, 471  
    results. *See* result files in variable browser  
    setup, 41, 470

simulation menu  
    continue, 468  
    import initial, 468  
    linearize, 469  
    new script, 466  
    open script, 466  
    run script, 466  
    setup, 470  
    show log, 486  
    simulate, 468  
    stop, 468  
    translate, 467  
    visualize, 486  
simulation mode, 13  
Simulation mode, 23, 145  
Simulink, 12. *See also* Matlab  
    troubleshooting, 669  
singularity  
    structural, 599  
size (built-in function), 573  
smooth, 355  
split model, 239  
stability boundary, 525  
stacked connectors, 252  
start values  
    discriminating, 618  
    fixed, 53, 126, 175  
    save to model, 84  
        values - save in model from simulation, 213, 395  
starting Dymola, 23  
state event, 121  
state machines, 235  
state selection, 356  
    attribute in parameter dialog, 269  
    dynamic, 382, 478  
    logging, 478  
statements  
    inserting, 210  
stateSelect, 356  
step size, 524  
stiff, 525, 526  
stop  
    Dymola, 307  
    simulation, 468  
store  
    as one file, 316  
    in model, 470, 472, 476  
        of protected variables, 461  
stream connector, 114  
string manipulation, 536

structural  
     parameters, [460](#)  
     singularities, [599](#)  
 structurally incomplete, [600](#)  
 submodel, [239](#)  
*Subversion.* *See* model management : version control  
*SVN.* *See* model management : version control  
 symbols  
     =, [534](#)  
 synchronous equations, [118](#)  
 syntax  
     check, [205](#)  
     errors, [338](#)  
 syntax check, [40](#)  
 system requirements  
     hardware, minimum, [658](#)  
     hardware, recommended, [658](#)  
     software, Linux, [659](#)  
     software, Windows, [659](#)

## T

table handling  
     import/export files, [277](#)  
     interpolation, [533](#)  
     n-dimensional, [278](#)  
     plot, [359](#)  
     script, [533](#)  
 TableND block, [278](#)  
 temperature  
     absolute, [281](#)  
     difference, [281](#)  
 templates, [259](#)  
 terminate  
     Dymola, [307](#)  
 text  
     "null-extent" text object, [186](#)  
     alignment, [225](#), [457](#)  
     bold, [225](#), [457](#)  
     bullet list, [225](#), [457](#)  
     italic, [225](#), [457](#)  
     minimize font size, [188](#), [320](#)  
     numbered list, [225](#), [457](#)  
     size. *See* font size  
     strings, [186](#)  
     texts in plots, [437](#)  
     underline, [225](#), [457](#)  
 tilt  
     animation view, [445](#)  
 time line in plot, [437](#)  
 tolerance, [472](#)

toolbar, [184](#), [185](#)  
 tools  
     window handling, [327](#)  
 tooltip  
     for x- and y-axis in plots, [411](#)  
 tooltips  
     copy to clipboard, [409](#)  
     dynamic, [162](#)  
     dynamic tooltips for signals in plot, [408](#)  
     for legends in plot, [409](#)  
     plot window, [408](#)  
 tracing  
     bus signals (expandable connectors), [384](#)  
     in a script, [553](#)  
 translate  
     model – export to other simulation environments, [467](#)  
     model - translateModel (built-in function), [566](#)  
     model to export - translateModelExport (built-in function), [566](#)  
     model to FMU - translateModelFMU (built-in function), [566](#)  
     translate model, [43](#), [389](#)  
         FMU, [467](#)  
         normal, [467](#)  
     translateModel, [566](#)  
     translateModelExport, [566](#)  
     translateModelFMU, [566](#)  
 troubleshooting  
     Dymola on Windows 7, [668](#)  
 troubleshooting  
     compiler, [667](#)  
     language, [670](#)  
     license borrowing, [666](#)  
     license file, [665](#)  
     license server, [666](#)  
     other windows problems, [670](#)  
     Simulink, [669](#)  
 type prefix, [264](#)  
     causality, [266](#)  
     connector members, [266](#)  
     dynamic typing, [267](#)  
     final, [266](#)  
     protected, [266](#)  
     replaceable, [266](#)

## U

UNC, [626](#)  
 undo, [308](#)  
 undock, [154](#), [448](#)  
 unit

checking, 283, 286  
conversion. *See* displayUnit  
deduction, 283, 287  
display. *See* displayUnit  
upgrading Dymola, 635  
URI, 225  
    modelica URI scheme, 189, 226  
URL, 457  
used classes, 327  
    layer, 151  
UTF-8, 233

## V

variability type. *See* type prefix  
variable  
    annotations, 267  
    attribute fixed, 175  
    browser, 372, 504, 506  
    declaration, 207, 262, 318  
    declaration dialog, 264  
    in Modelica, 109  
    interactive, 534  
    matching, 270  
    predefined, 535  
    replaceable, 365  
    selections, 270  
    store of protected, 461  
    type prefix. *See* type prefix  
variable browser context menu  
    animate, 505  
    animate together, 505  
    open all, 504  
variable order (algorithms), 524  
variable step size, 524  
variable structure systems, 122  
variables, 568  
version, 318  
version control. *See* model management : version control  
view menu  
    diagram layer, 326  
    documentation layer, 326  
    equation layer, 327  
    icon layer, 326  
visual  
    modeling, 377  
    reference, 500  
visualize simulation, 486  
visualizer window, 377

VRML, 299, 463

## W

warnings, 52, 77, 85. *See also* errors  
what's this, 329  
widgets  
    adding GUI widgets in parameter dialog, 269  
window  
    animation, 34, 376  
    command, 13, 378, 477  
    diagram layer, 35, 373, 402  
    Dymola Main, 24, 144  
    edit, 25, 146  
    grid, 152  
    information browser, 159, 388  
    library, 13, 158  
    main, 13  
    message, 382  
    plot, 34, 374  
    tools, 327  
    types, 144, 370  
    undock, 154, 448  
    variable browser, 372  
    visualizer, 377  
window menu  
    cascade windows, 328  
    close all windows, 328  
    diagram layer, 326  
    documentation layer, 326  
    icon layer, 326  
    message window, 328  
    Modelica text, 327  
    modeling, 326  
    new command window, 328  
    new Dymola window, 328  
    new library window, 328  
    next, 326  
    previous, 326  
    simulation, 326  
    tile windows, 328  
    tools, 327  
    used classes, 327  
Windows  
    Dymola as 32-bit application, 624  
    Dymola as 64-bit application, 624  
    Dymola license server, 647  
Windows 7  
    Dymola on Windows 7, 668  
working directory  
    Dymola, 450, 626  
    set default for Dymola, 635

workspace, [450](#)

## X

X3D

as HTML/JS, [300](#), [463](#)  
as pure XML, [299](#), [463](#)

## Z

zoom factor  
diagram layer, [326](#)

zooming  
animation window, [35](#), [445](#)  
concurrent. *See* plot window zooming  
diagram layer, [161](#), [326](#), [401](#)  
icon layer, [161](#)  
plot window, [76](#). *See* plot window zooming