

Dymola

Dynamic Modeling Laboratory

User's Manual

Dymola

Dynamic Modeling Laboratory

User's Manual

Version 5.3a

© Copyright 1992-2004 by Dynasim AB. All rights reserved.

Dymola™ is a trademark of Dynasim AB.

Dymola® is a registered trademark of Dynasim AB in Sweden.

Modelica® is a registered trademark of the Modelica Association.

Dynasim AB
Research Park Ideon
SE-223 70 Lund
Sweden

E-mail: Support@Dynasim.com
URL: <http://www.Dynasim.com>
Phone: +46 46 2862500
Fax: +46 46 2862501

Contents

What is Dymola?	13
Features of Dymola	13
Architecture of Dymola	14
Basic Operations	14
Simulating an existing model	15
Building a model	16
Features of Modelica	19
Background	19
Equations and reuse	19
Modelica history	20
Getting started with Dymola	23
Introduction	23
Simulating a model — industrial robot	24
Simulation	32
Other demo examples	33
Solving a non-linear differential equation	33
Simulation	36
Improving the model	40

Using the Modelica Standard Library	42
The Modelica Standard Library	43
Creating a library for components	48
Creating a model for an electric DC motor	49
Testing the model	56
Creating a model for the motor drive	59
Parameter expressions	61
Building a mechanical model	65
Introduction to Modelica	73
Modelica basics	73
Variables	74
Connectors and connections	75
Partial models and inheritance	75
Acausal modeling	76
Background	76
Differential-algebraic equations	78
Advanced modeling features	79
Vectors, matrices and arrays	79
Class parameters	79
Algorithms and functions	81
Hybrid modeling in Modelica	81
Synchronous equations	82
Relation triggered events	85
Variable structure systems	86
Initialization of models	90
Basics	90
Continuous time problems	91
Parameter values	95
Discrete and hybrid problems	96
Example: Initialization of discrete controllers	97
Standard libraries	100
Summary	101
References	101
Developing a model	107
General concepts	107
Window types	107
Class layers	111

Class documentation	112
Coordinate system	113
Model editing	114
Basic operations	114
Components and connectors	115
Connections	121
Creating graphical objects	123
Changing graphical attributes	124
Modelica text	125
Documentation	126
HTML documentation	129
External references	129
HTML options	130
Editor command reference	131
File menu	131
Edit menu	138
Window menu	141
Help menu	142
Special keyboard commands	143
Model editor initialization	143
Simulating a model	147
Basic steps	147
Simulation menu	149
Plot window	156
Variable selector	157
Plot window interaction	159
File menu	160
Plot menu	160
Animation window	166
Defining Graphical Objects	166
File menu	168
Animation menu	169
Scripting language	173
Basic operations	173
Script files	176
Help commands	176
Simulator API	178
Script functions	181

Debugging models	182
Over specified initialization problems	182
Basic steps in debugging models	182
Finding errors in models	183
Improving simulation efficiency	184
Inline integration	190
Inline integration	190
Inline integration in Dymola	192
References	192
Mode handling	192
Collecting modes	192
Using mode information in real-time simulation	193
Known Limitations	193
References	193
Dynamic Model Simulator	197
Overview	197
What is Dymosim?	197
Who wrote Dymosim?	198
Running Dymosim	198
Dymosim as a stand-alone program	199
Dymosim and Matlab	200
Selecting the integration method	201
Integrator properties	201
Dymosim integrators	203
Dymosim reference	205
Model functions for Dymosim	205
Dymosim m-files	206
Dymosim command line arguments	207
Basic file format	208
Dymosim input file “dsin.txt”	209
Simulation result file “dsres.mat”	213
Bibliography	215
Other simulation environments	219
Using the Dymola-Simulink interface	219
Graphical interface between Simulink and Dymola	219
Simulation in Matlab	224
Real-time simulation	225

dSPACE systems	225
xPC and Real-Time Workshop	228
Real-time simulation using RT-LAB	229
DDE communication	233
Dymola DDE commands	233
Explorer file type associations	233
Dymosim Windows application	234
Appendix — Modelica	239
Modelica syntax specification	239
Lexical conventions	239
Grammar	240
Appendix — Advanced Modelica	249
Declaring functions	249
User-defined derivatives	249
How to declare a derivative	250
External functions	253
Including external functions	254
Linking to external library	254
Other languages	256
Means to control the selection of states	257
Motivation	257
The state select attribute	258
Using noEvent	259
Background: How events are generated	260
Guarding expressions against evaluation	260
How to use noEvent to improve performance	261
Combined example for noEvent	261
Mixing noEvent and events in one equation	262
Constructing anti-symmetric expressions	264
Equality comparison of real values	266
Type of variables	266
Trigger events for equality	266
Locking when equal	267
Guarding against division by zero	267
Appendix — Migration	271
Migrating to newer libraries	271

How to migrate	271
Basic commands to specify translation	272
How to build a convert script	275
Appendix — Installation	281
Installation on Windows	281
Installing Dymola	281
Dongle installation	283
Additional setup	284
Installing updates	285
Removing Dymola	285
Installation on UNIX	285
Installing Dymola	285
Additional setup	286
Removing Dymola	287
Dymola License Server	287
Background	287
Installing the license server	288
Installing on client computers	290
Troubleshooting	290
License file	291
Compiler problems	291
Simulink	292
dSPACE systems	293
Other Windows-related problems	293
Index	295

WHAT IS DYMOLA?

What is Dymola?

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 casual connections. Model libraries are available in many engineering domains.

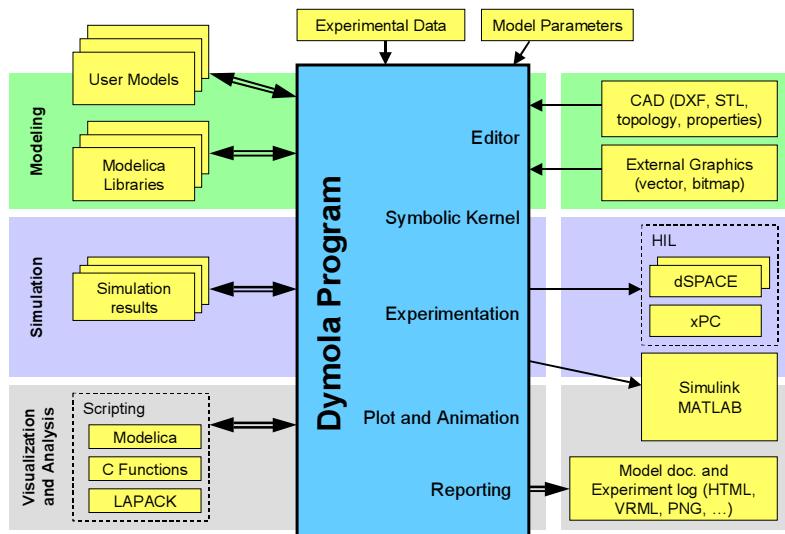
Dymola uses a new modeling methodology based on object orientation and equations. The usual need for manual conversion of equations to a block diagram is removed by the use of automatic formula manipulation. Other highlights of Dymola are:

- 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.

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 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.



Basic Operations

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

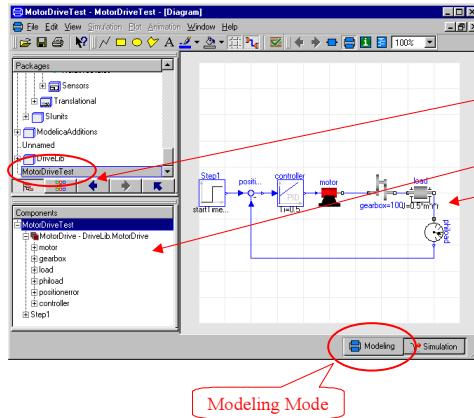
The Modeling mode of the Main window is used to compose models and model components.

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

Simulating an existing model

Find the model

Dymola starts in Modeling mode. The model to simulate is found by using the Open or Demo commands in the File menu. Example models are also found in the libraries: Modelica or ModelicaAdditions or in other libraries opened by the Library command.

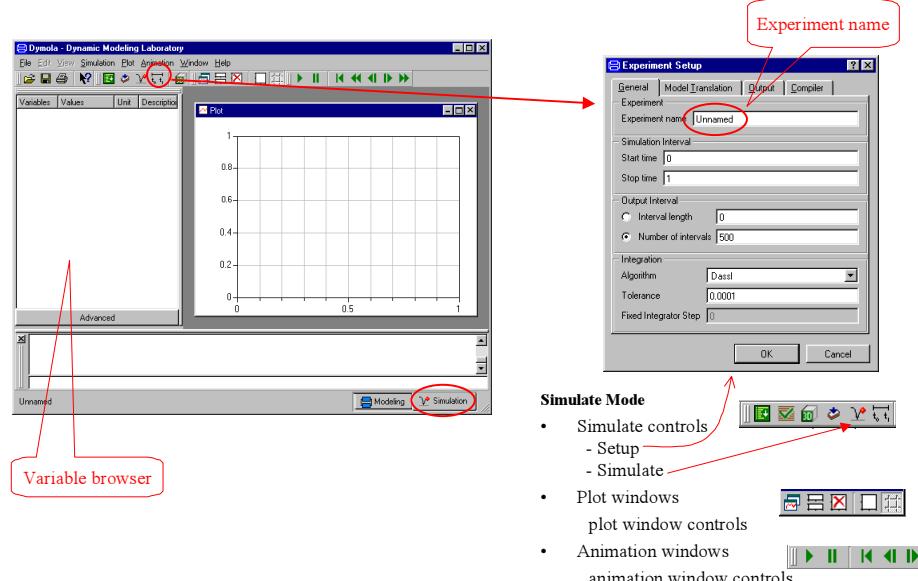


Browsing

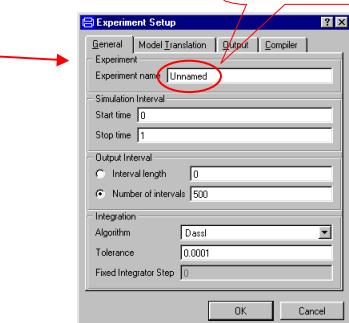
- Tree Browser for model packages and models
- Component hierarchy
- Viewer
 - diagram
 - documentation

Simulation Mode

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



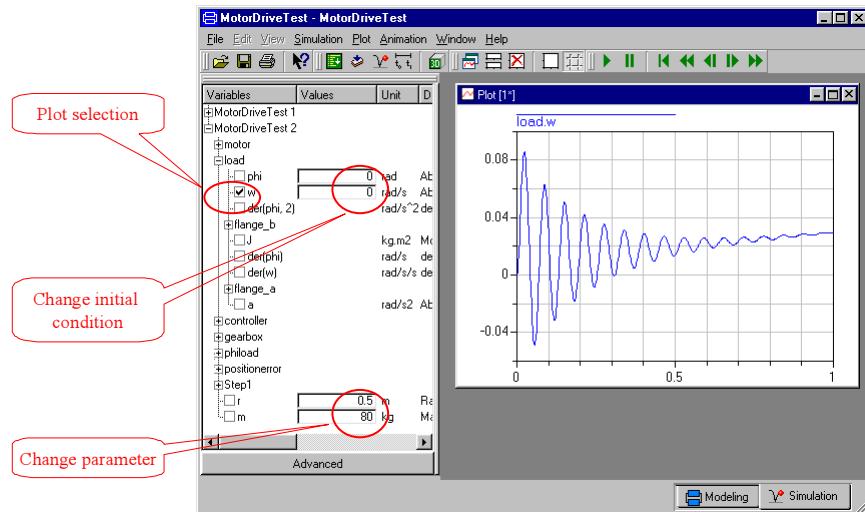
Experiment name



Simulate Mode

- Simulate controls
 - Setup
 - Simulate
- Plot windows
 - plot window controls
- Animation windows
 - animation window controls

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



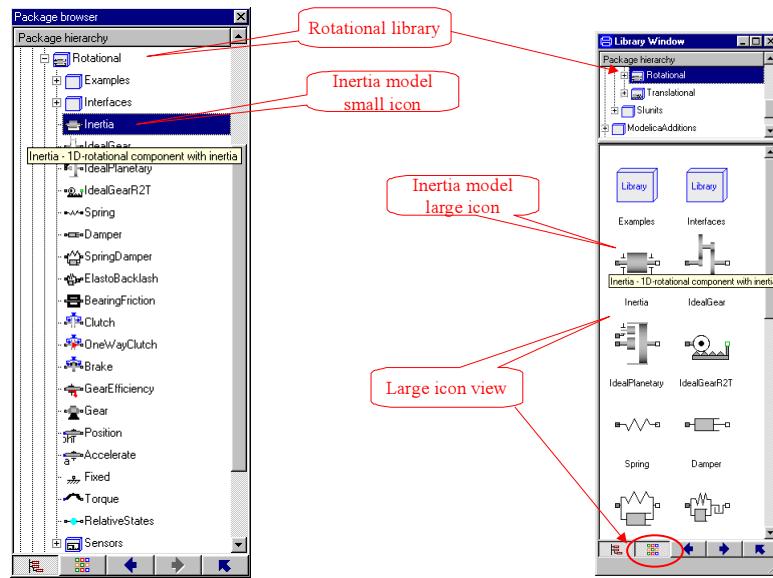
An animation window shows a 3D view of the simulated model. The animation can be run at different speeds, halted, single stepped and run backwards.

Building a model

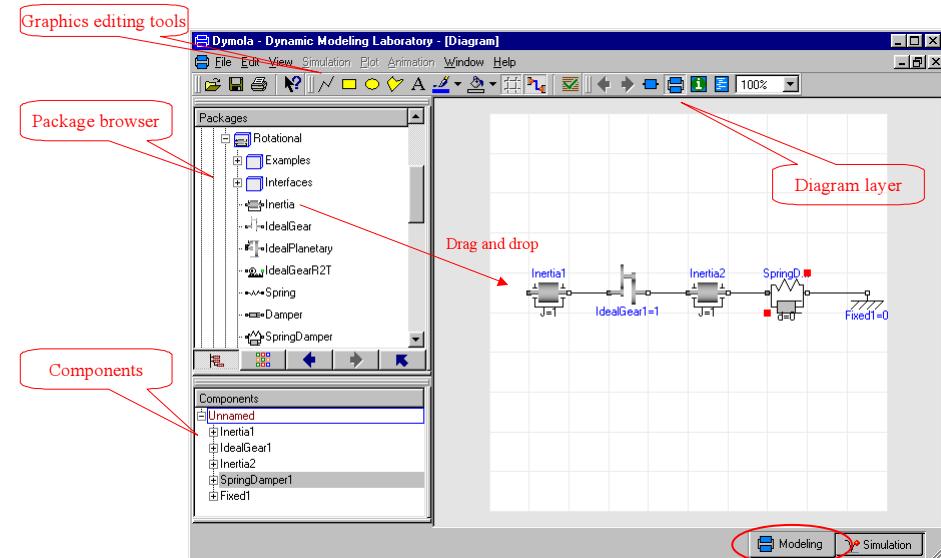
The graphical model editor is used for creating and editing models in Dymola. Structural properties, such as, components, connectors and connections are edited graphically, while equations and declarations are edited with a built-in text editor.

Find a component model

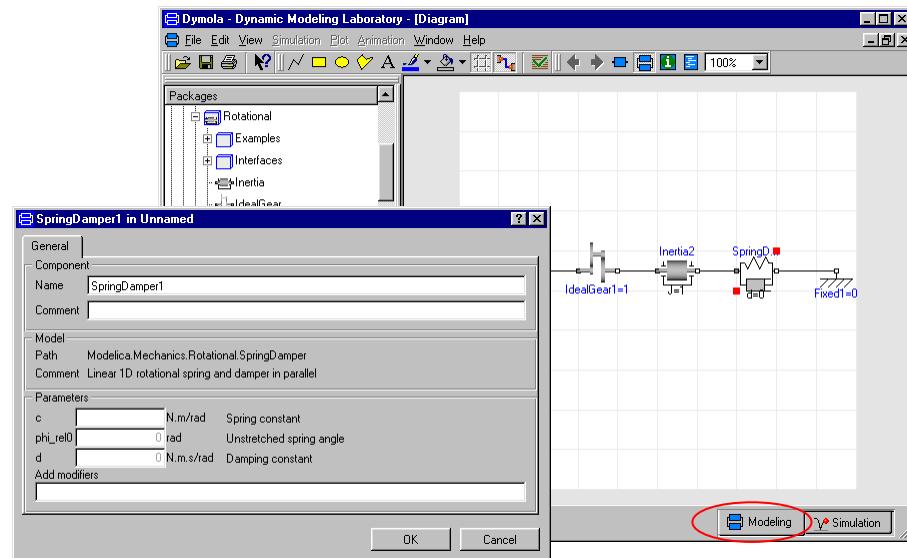
The package browser allows viewing and selecting component models from a list of models with small icons. It is also possible to get a large icon view



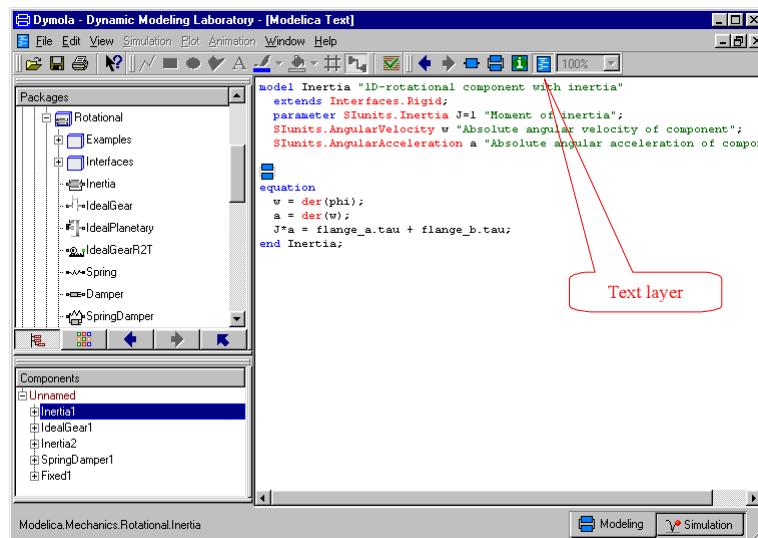
Components are dragged from the package browser to the diagram layer and connected. The component hierarchy is shown in the component browser.



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



The model browser allows opening a component for inspection of the documentation or the model itself, for example, 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.



Features of Modelica

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

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

Background

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

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

Equations and reuse

**Equations facilitate
true model reuse.**

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

The language has been designed to allow tools to generate very efficient code. Modelica models are used, for example, in hardware-in-the-loop 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.

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 25 experts with previous know-how of modeling languages and differential-algebraic equation models. After more than 30 three-days' meetings during a five year period, version 2.0 of the language specification was finished in January, 2002.

GETTING STARTED WITH DYMOLA

Getting started with Dymola

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 Help menu after selecting Documentation. The tool tips and the "What's this" feature are fast and convenient ways to access information.

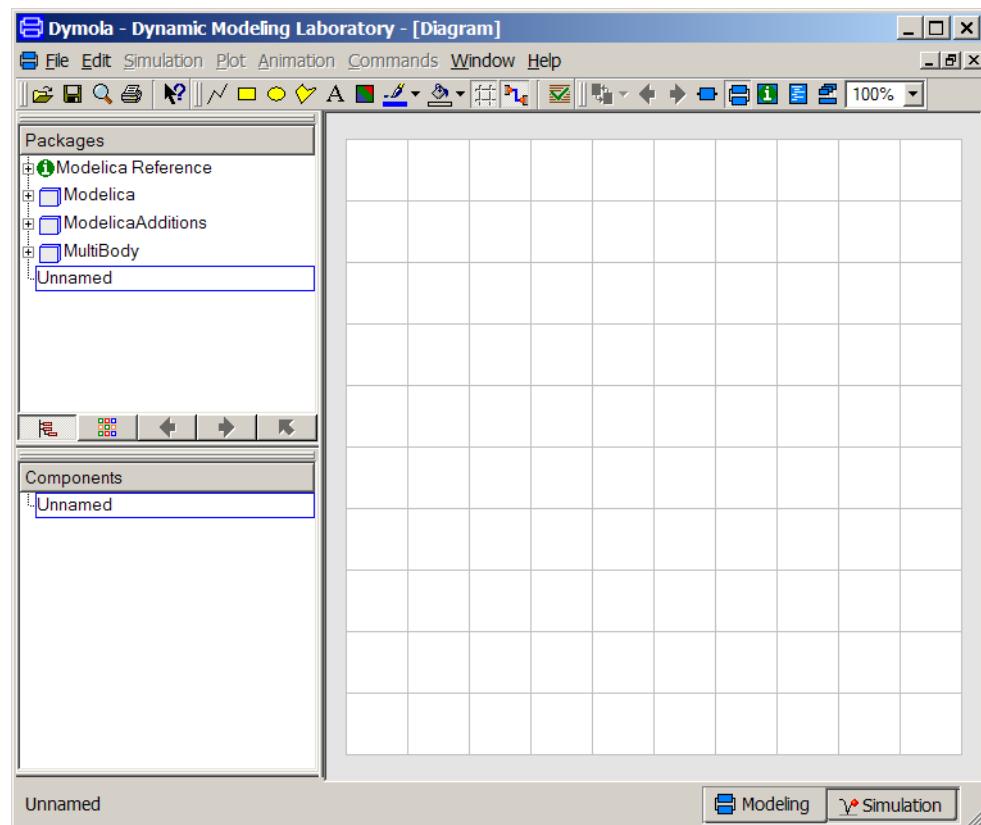
Start Dymola. A Dymola window appears. A Dymola window operates in one of the two modes:

- **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 depends on the mode, and additional ones have been added after this guide was written. Dymola starts with a useful default configuration, but allows customizing.

The Dymola window.

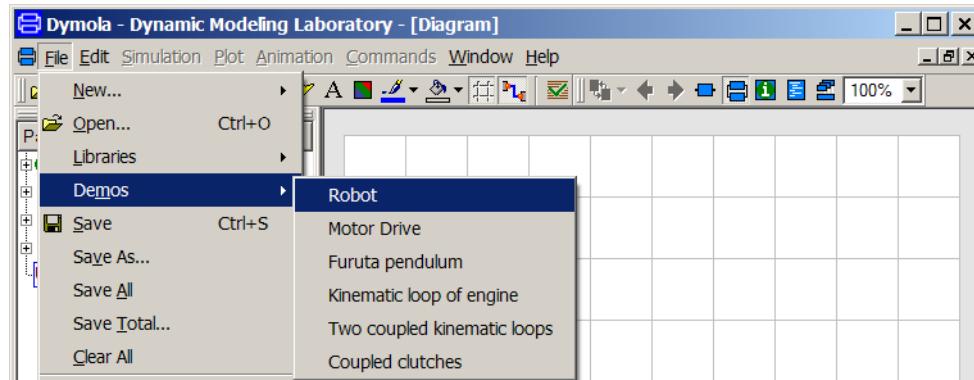


Simulating a model — industrial robot

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 “Solving a non-linear differential equation.”

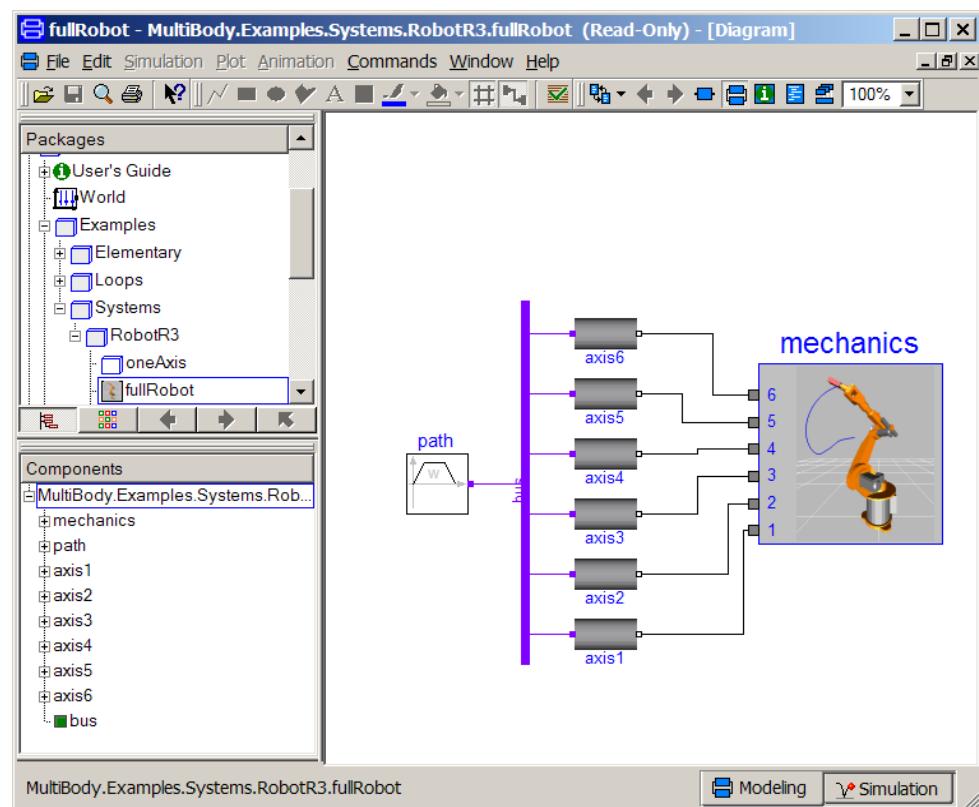
We will study a model of an industrial robot. Start Dymola. The Dymola window appears in Modeling mode as displayed above. To view the industrial robot model, use the File/Demos menu and select Robot.

Opening a demo example.



Dymola starts loading the model libraries needed for the robot model and displays it.

The robot demo.



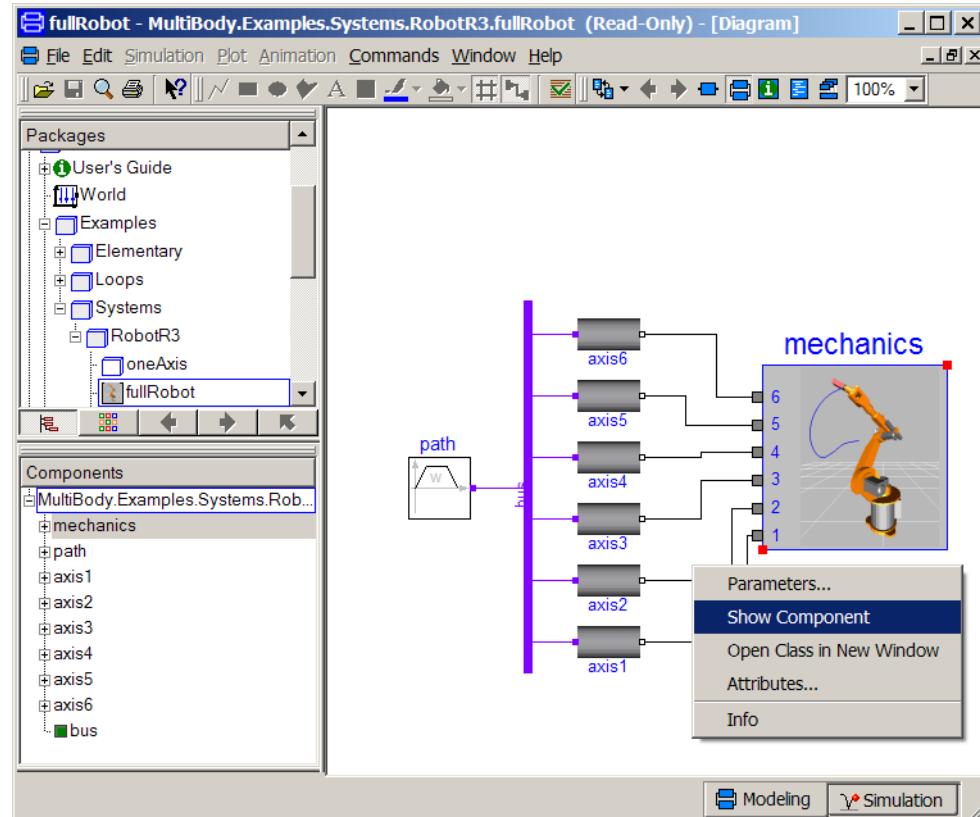
The package browser in the upper left sub-window displays the package hierarchy and it is now opened up with the robot model selected and highlighted. The model diagram in the right sub-window opens up and shows the top-level structure of the model. The model diagram has an icon for the model of the 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 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 Diagram window (red handles appear, see below) and press the right button on the mouse. From the menu, choose Show Component.

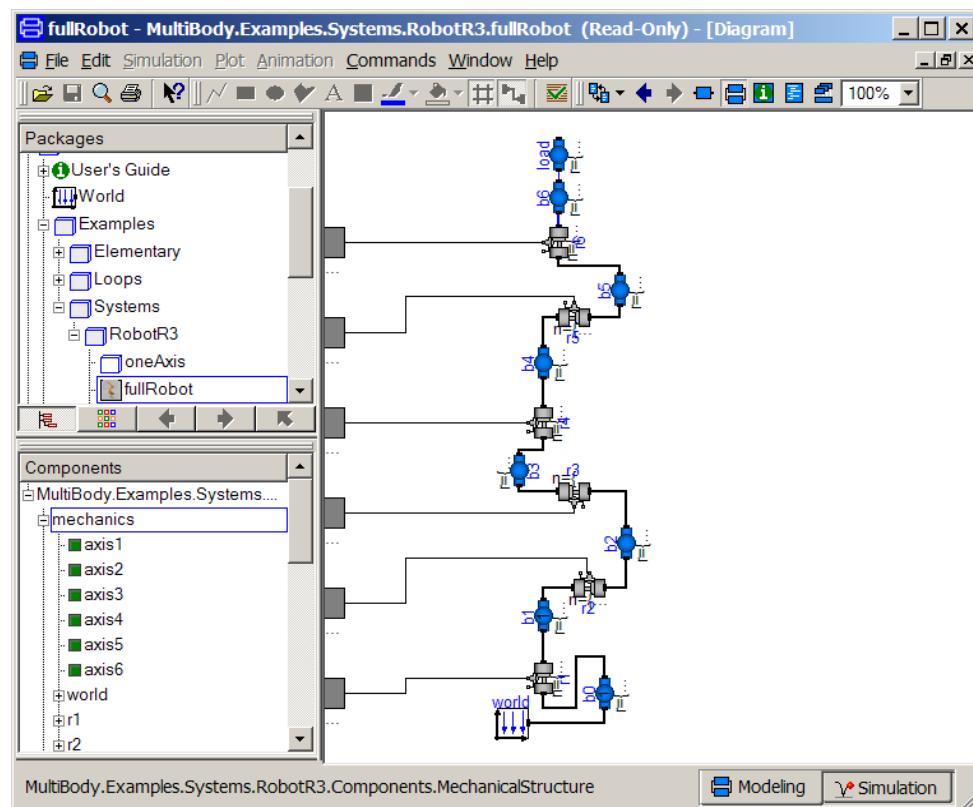
About to view the mechanical structure of the robot.



It is not necessary to select the robot component explicitly by pressing the left button on the mouse to access its menu. It is sufficient to just have the cursor on its icon in the diagram window and press the right button on the mouse. The component browser also gives easy access to the robot component. Just position the cursor over “mechanics”. The component browser provides a tree representation of the component structure. The diagram window and the component browser are synchronized to give a consistent view. When you select a component in the diagram window, it is also highlighted in the component browser and vice versa. The diagram 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.

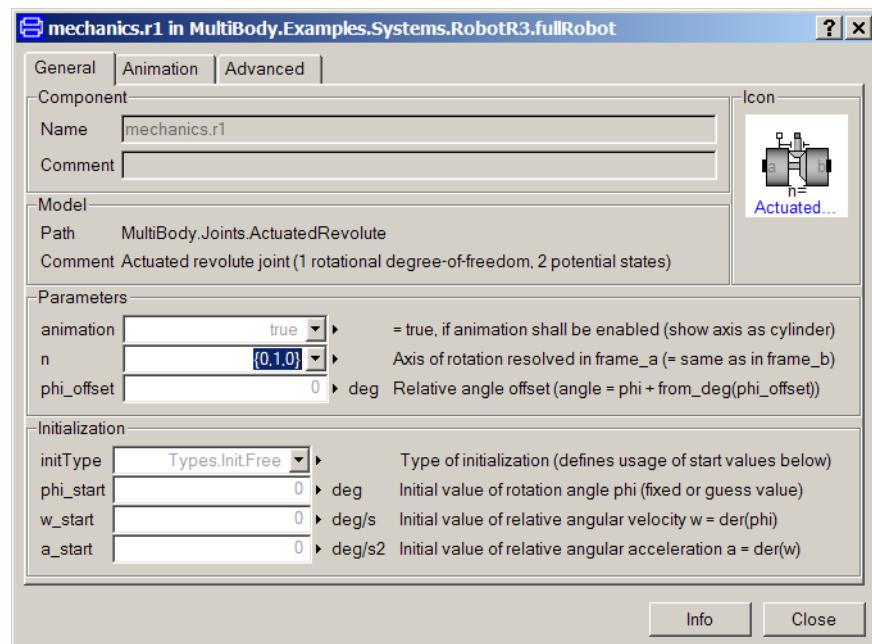
The Diagram window now displays the mechanical structure consisting of connected joints and masses. The component browser is opened up to also show the internals of the mechanics model component.

The mechanical structure of the robot.



Double click on, for example, $r1$ at the bottom of the diagram window. This is a revolute joint. The Parameter dialogue appears. The Parameter dialogue can also be accessed from the right button menu. Double clicking the left button selects the first alternative from the right button menu.

Parameter dialogue.

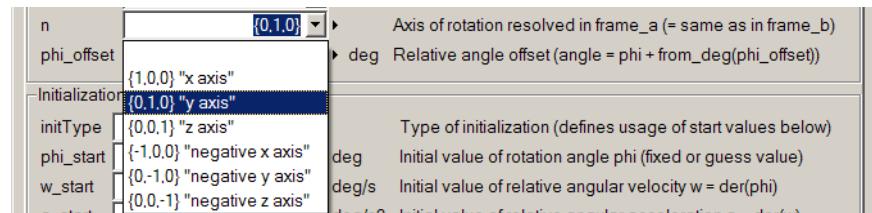


The parameter dialogue allows the user to inspect actual parameter values. In this case the parameter values are write protected to avoid unintentional changes of the demo example. Thus the dialogue just has a Close button (and an Info-button). When the parameter values can be changed there is one OK 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 dialogue may have several tabs. This dialogue has the tabs: General, Animation and Advanced. In a tab the parameters can be further structured into groups as shown. It is easy for a model developer to customize the parameter dialogues. Graphical illustrations can be included to show meaning of parameters.

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.). 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. Press Close.

Choices for n.

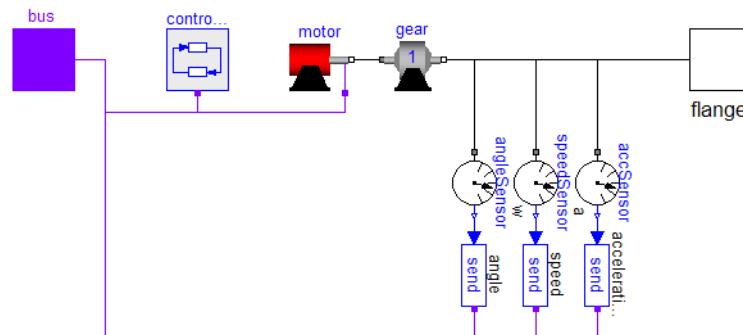


To learn more about this component, select Info. An HTML browser is opened to show the documentation of the Revolute joint.

If you do only want to see the documentation select the component in the diagram, press the right mouse button and select Info. An HTML browser is opened to show the documentation of the Revolute joint. By double clicking on the other components, you can, for example, see what parameters a mass has.

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 use the right button menu on one of the axis icons. Please, note that robot.mechanics also has components axis1, ..., axis6, but those are just connectors. You shall inspect for example robot.axis1. Another convenient way is to use the component browser and use the right button menu and select Show Component. Since this is the first menu option, double clicking will open up the component in the Diagram window. Please, recall that double clicking on a component in the diagram window pops up the parameter dialogue.

The robot drive train.



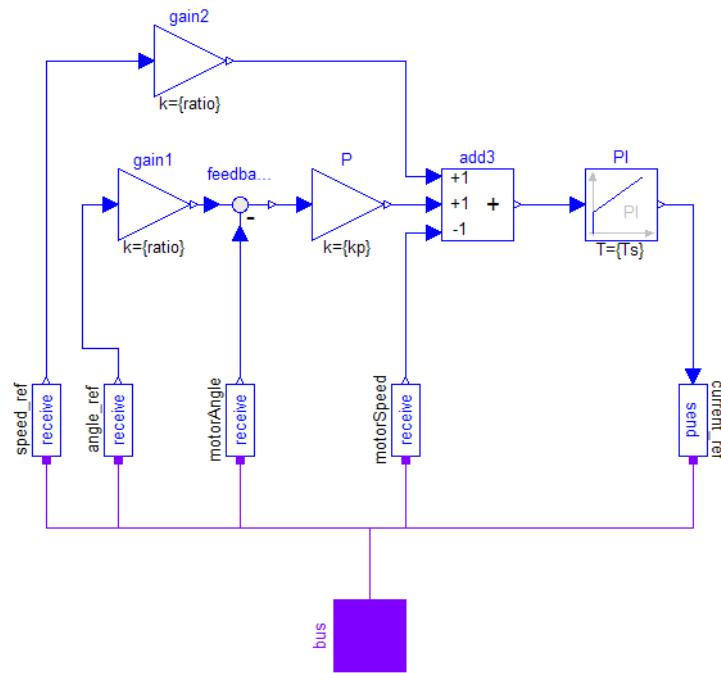
The drive train includes a controller. A data bus is used to send measurements and reference signals to the controller and control signals from the controller to the actuator. The bus for one axis has the following signals:

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

There are send elements to output information to the bus as illustrated in the model diagram above. Similarly there are a read elements to access information from the bus.

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

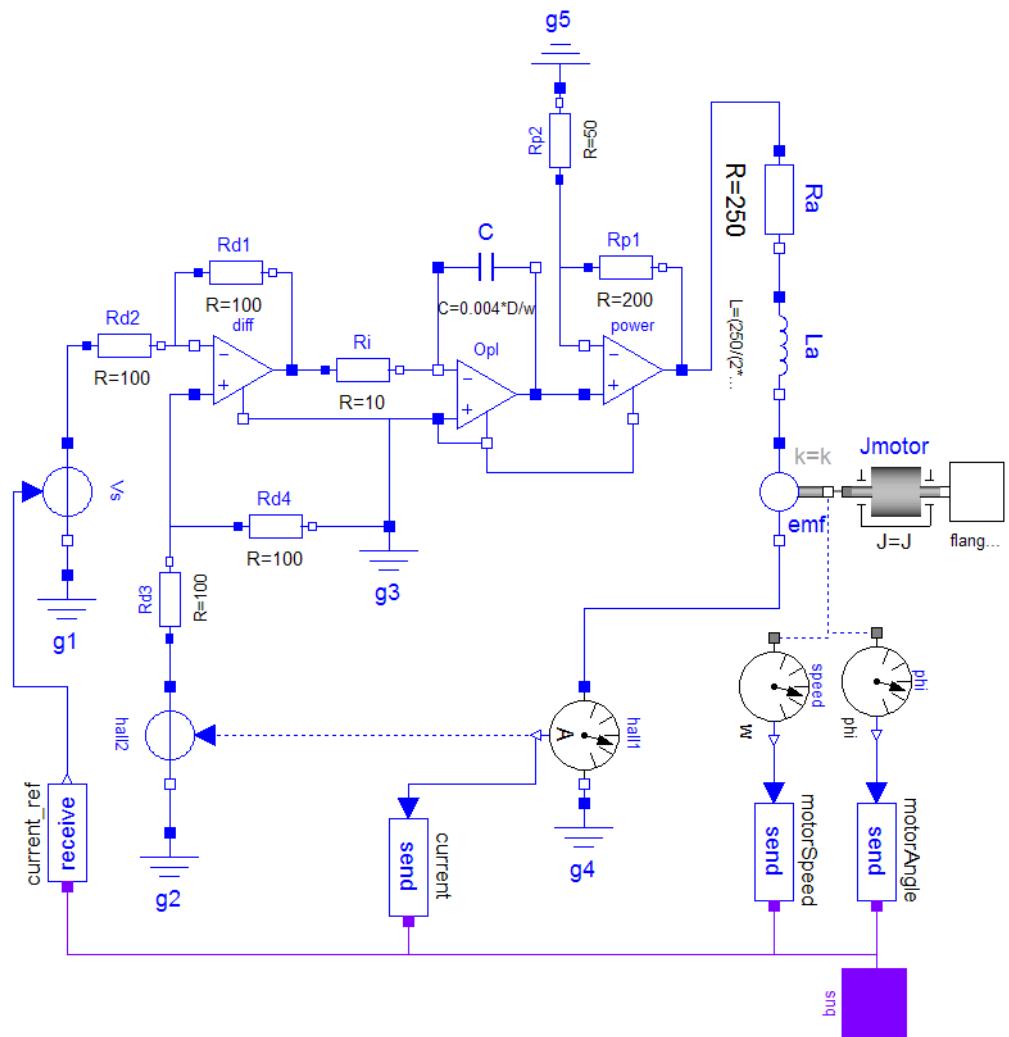
The robot controller



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

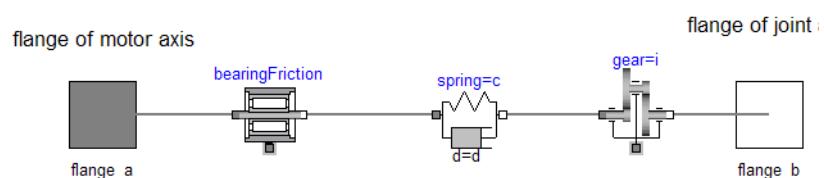
The motor model consists of the electromotorical force, three operational amplifiers, resistors, inductors, and sensors for the feedback loop.

The robot motor.



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

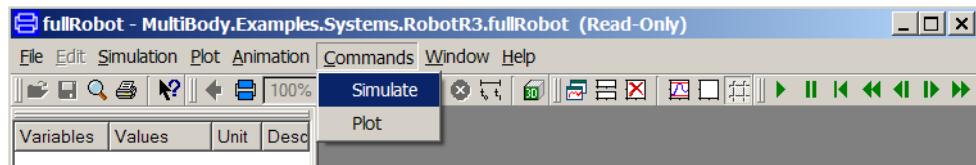
The robot gearbox.



Simulation



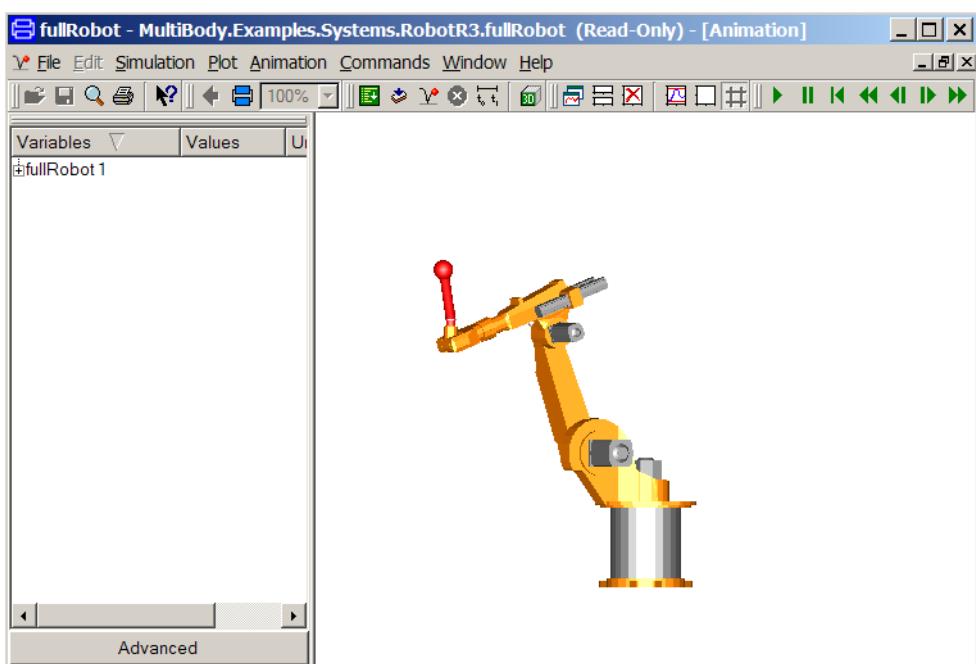
Let us simulate the robot model. To enter the Simulation mode, click on the tab at the bottom right. When you selected the robot the Commands menu became active to indicate that robot contains a Command for simulating it, select this.



In addition the simulation menu includes commands to setup and run simulations. The Set-up menu item opens a dialogue allowing the setting of start and stop times. The icon indicates which toolbar button to use for quick access to the set up dialogue. The Simulate menu item starts a simulation.

In this case, however, a command script has been prepared. To run the script, select Commands select 'Simulate'. Dymola then translates the model and simulates automatically. The animation window is opened automatically, because the simulation result contains animation information.

Animated 3D view of the robot.



Start the animation by selecting Animation/Run or clicking the run button on the tool-bar,

Animation toolbar.

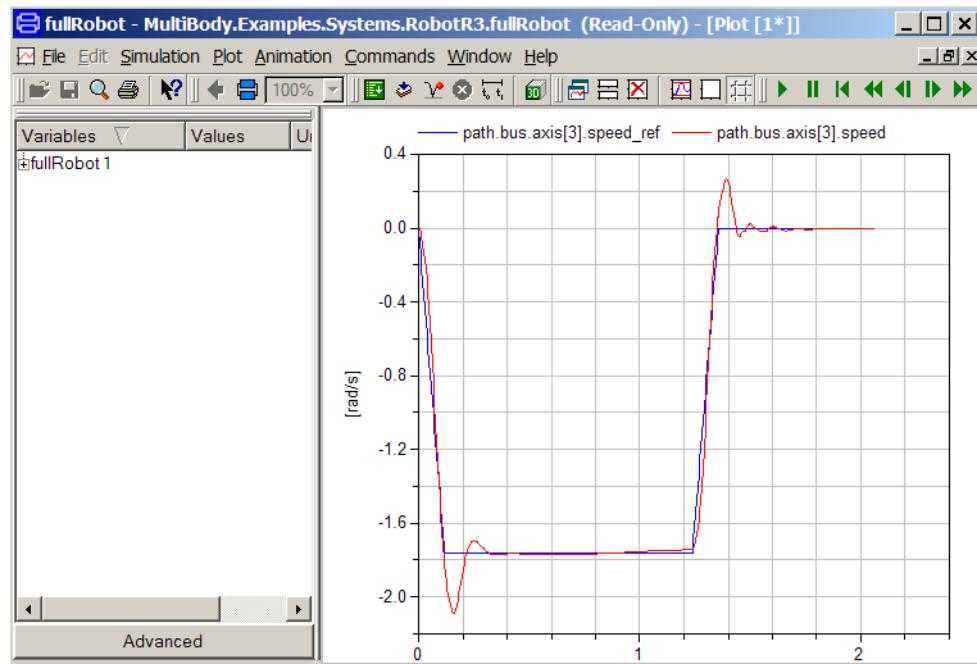


which contains the usual buttons of pausing rewinding, stepping forward and backward, respectively.

It is possible to change the viewing position by opening a view controller, Animation/3D View Control.

The command Plot makes a plot of the speed reference and actual speed of the third joint.

Plotting the speed.



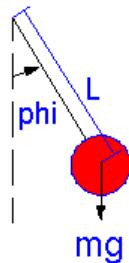
Other demo examples

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.

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

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

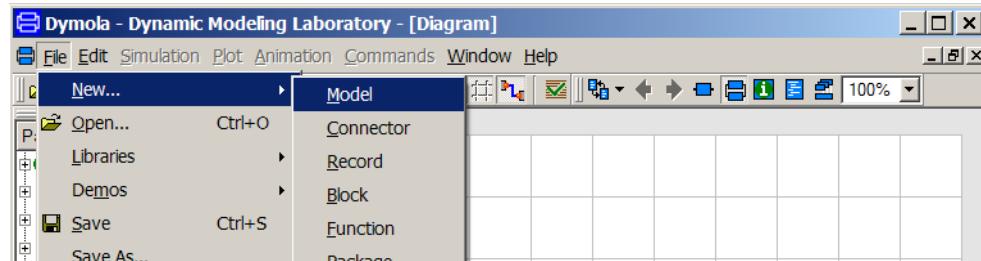
The variable w is the angular velocity and $\text{der}(w)$ denotes the time derivative of w , i.e., the angular acceleration. For the angular position we have

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

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

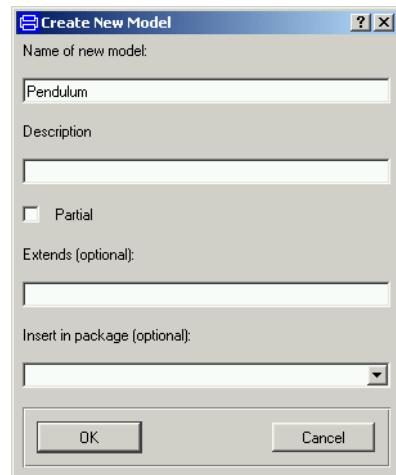
Click on the tab for Modeling at the bottom right. Select File/New Model.

The first step to create a new model.

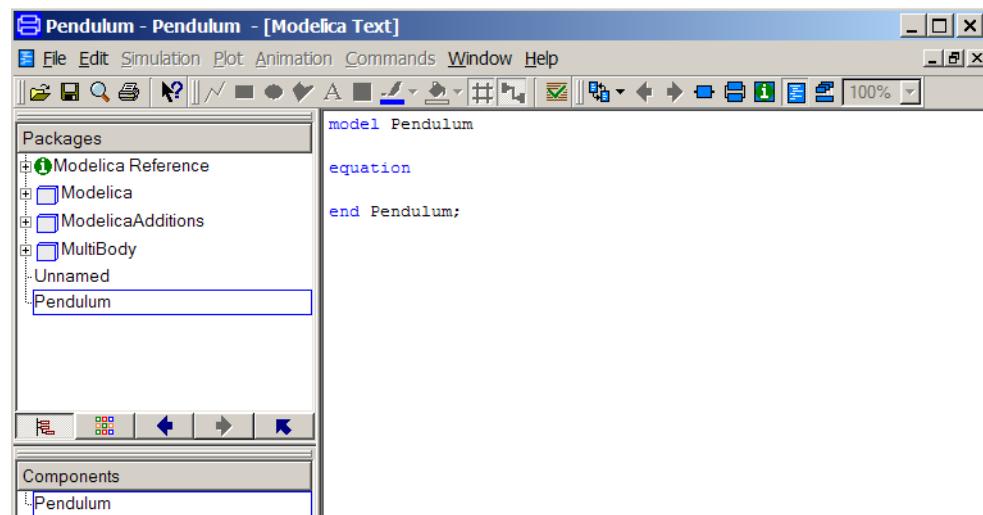


A dialog window opens. Enter “Pendulum” as the name of the model.

The dialogue to name a new model component.



Click OK. You will have 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. Press the Modelica Text toolbar button (the second rightmost tool button). The right subwindow can now be used as a text editor.

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 ϕ and w are declared. A start value is given for ϕ , while w is implicitly given a start value of zero.

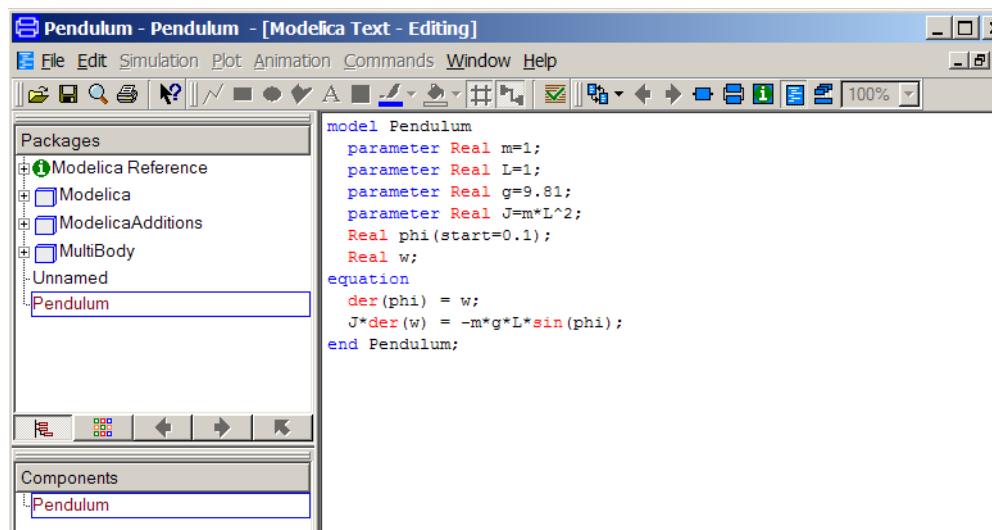
```

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 not automatically be color coded. To get the color coding of keywords and types, press the right mouse button and select Highlight Syntax.

Declaration of parameters, variables and equations.



The model is ready to be saved. Select File/Save. Call the file pendulum.mo and place it in a working directory.

Simulation

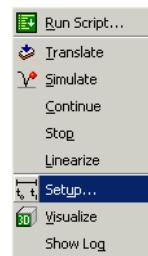


Now it is time to simulate. To enter the Simulation mode, click on the tab at the bottom right. The simulation menu is now activated and new tool bar buttons appear.

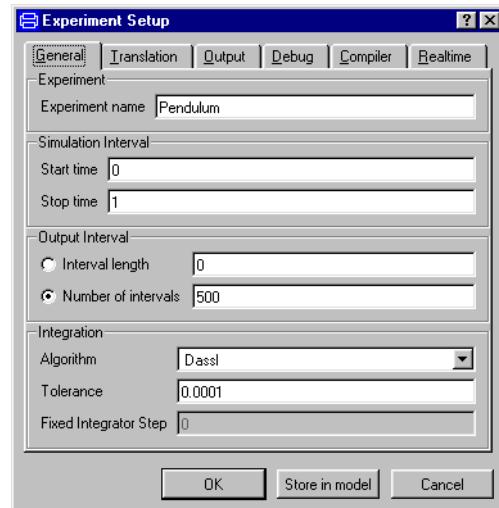


To set up the simulation select Simulation/Setup or click directly on the Setup toolbar button.

Selecting Setup in the Simulation menu.



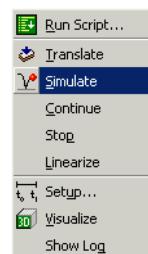
The Simulation Setup menu.



Set the stop time to 10 seconds. Click either OK, or Store in model (additionally this stores some information in the model).

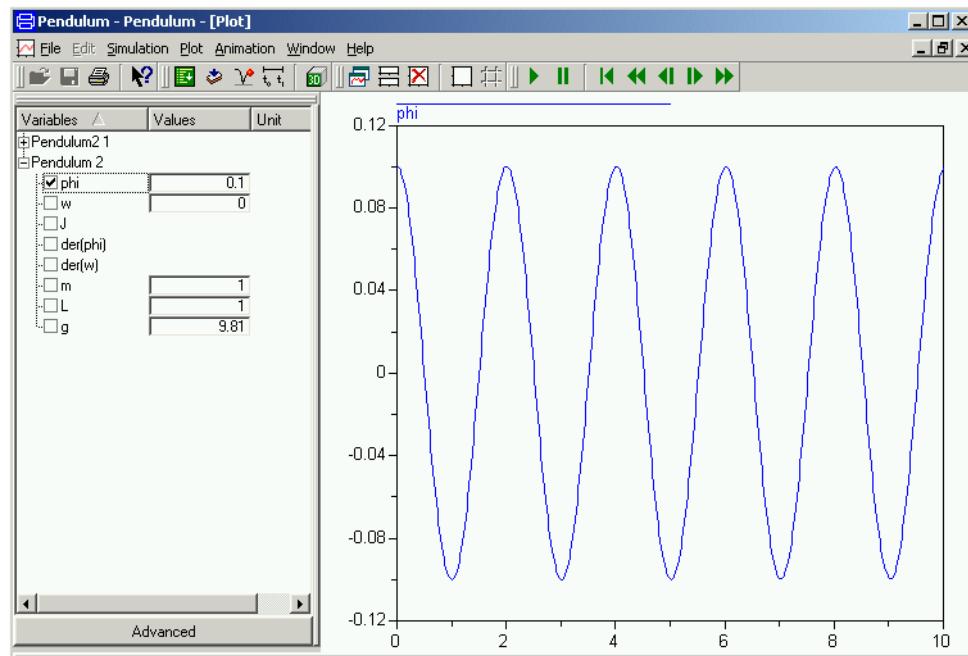


Selecting Simulate in the Simulation menu.



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.

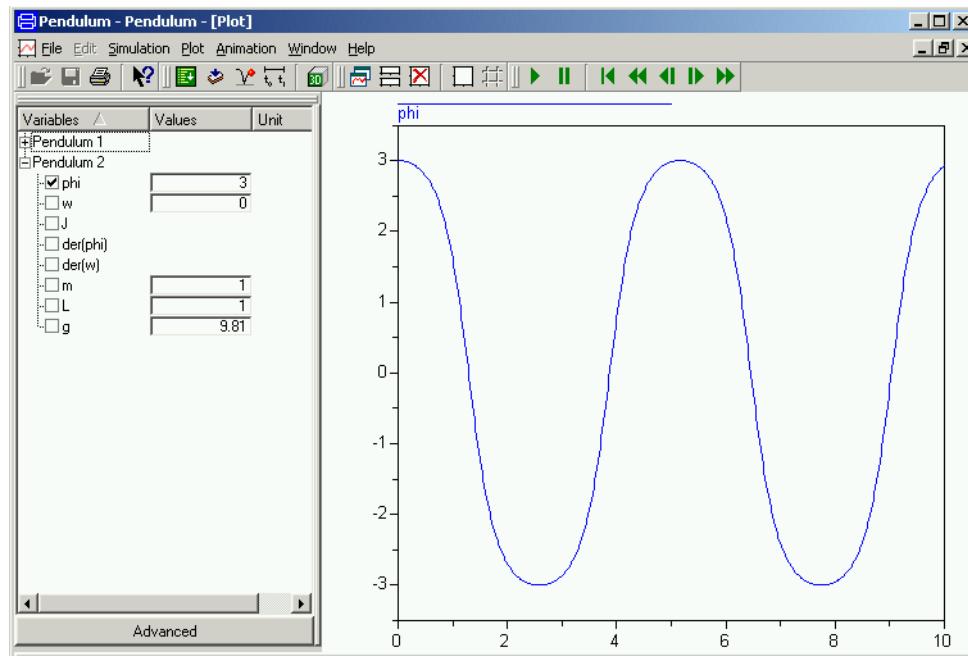
Plotting the angle.



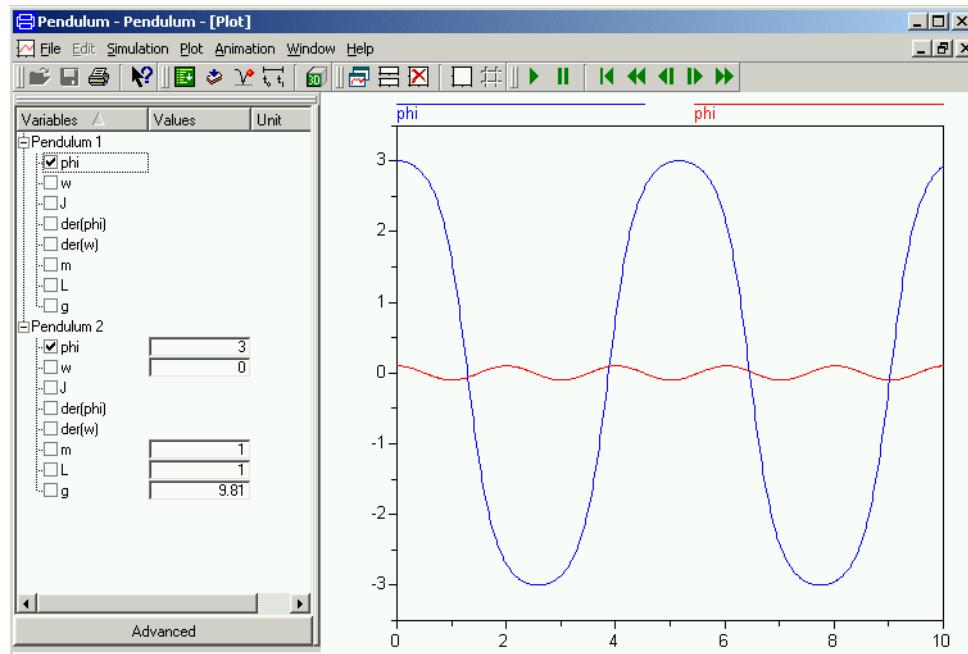
When the simulation is finished, the component browser displays variables to plot. Click in the square box in front of phi to get the angle plotted as shown above.

Let us study a swing pendulum with larger amplitude and let it start in almost the top position with $\phi = 3$. It is easy to change initial conditions. Just enter 3 in the value box for phi. Click on the Simulate tool button.

Pendulum angle when starting in almost the top position.



The results of previous simulations are available as the experiment Pendulum 1 and we can open it up and have phi of the two runs plotted in the same diagram.



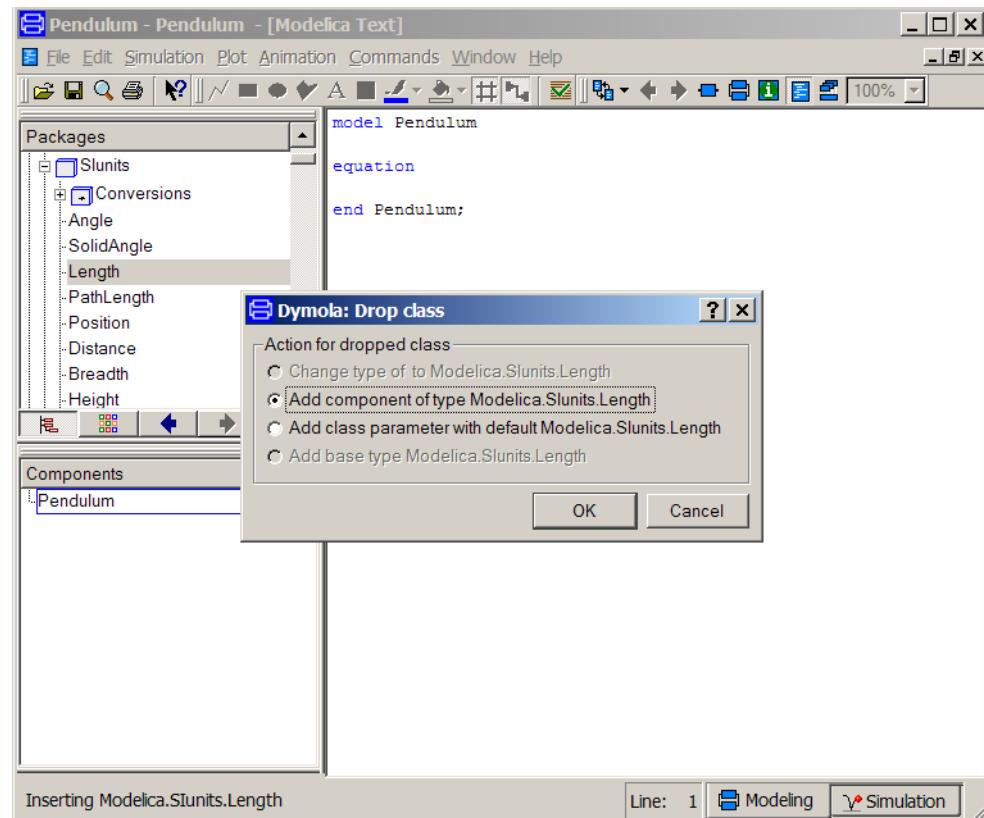
Values of parameters are changed in a similar way. To change the length of the pendulum, just enter a new value for L.

Improving the model

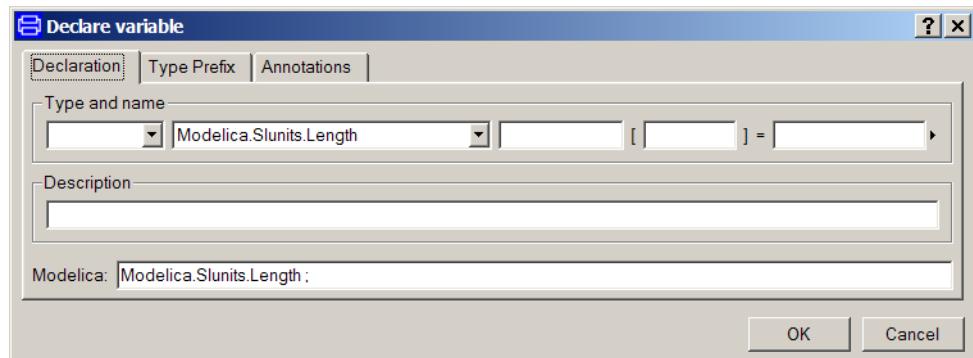
The parameters and variables are more than real numbers. They are physical quantities. The Modelica standard library provides type declarations for many physical quantities.

Let us redo the model. Start Dymola or if it is already started then give the command File/ Clear All in the Dymola window. As previously, click on the tab for Modeling at the bottom right. Select File/New Model. Name it pendulum. Click OK. Press the Modelica Text toolbar button (the second rightmost tool button).

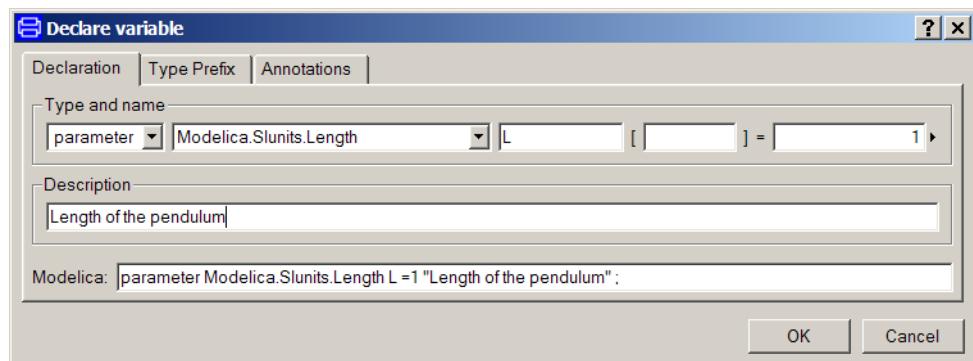
Open Modelica.SIunits in the package browser. To define the parameter length, drag Length to the component browser. A menu pops up:



The choice to add a component is preselected. Click OK. A menu to declare a variable pops up:



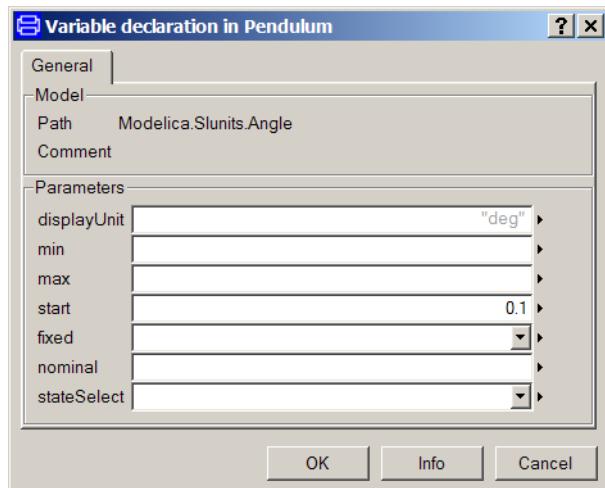
Complete the description:



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

The other quantities are defined in analogue ways. You have now Length selected in the package browser, to find the quantity Mass in Modelica.SIunits, enter M and the browser goes to the first component starting with M. If it is not the desired one, press M once again and so on to find it.

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 left of the value field and selecting Edit. A submenu pops up:



Enter 0.1 for start.

After entering the equations textually as previously the Modelica Text windows displays

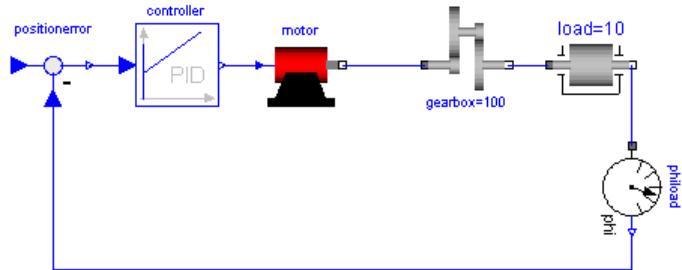
```
model Pendulum
  Modelica.SIunits.Mass m=1 "Mass of the pendulum";
  Modelica.SIunits.Length L=1 "Length of the pendulum";
  Modelica.SIunits.Acceleration g=9.81 "Gravity of acceleration";
  Modelica.SIunits.MomentOfInertia J=m*L^2 "Moment of inertia";
  Modelica.SIunits.Angle phi(start=0.1) "Pendulum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

The icon on the second line indicates graphical information or annotations. It may be displayed. Put the cursor on in the window, click the right mouse button to get the context menu. Select Expand/Show entire text, which reveals that it 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.

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.

The Modelica Standard Library

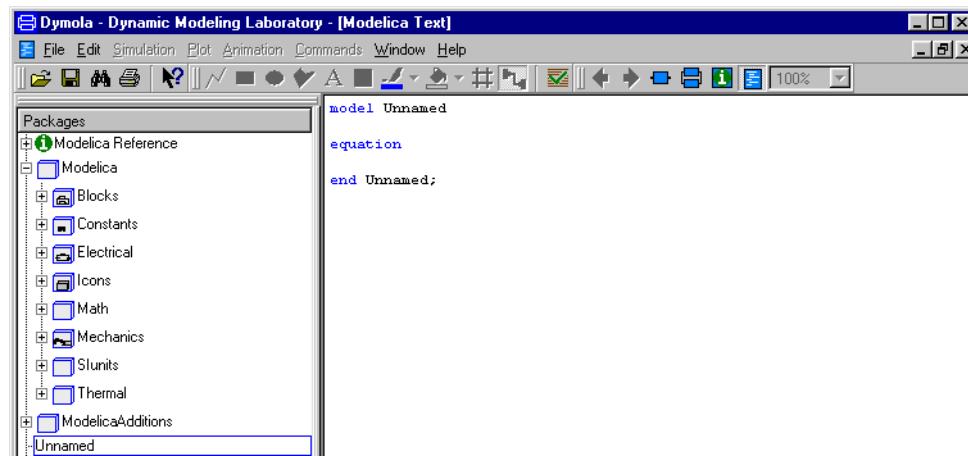
We will now have a look at the Modelica Standard Library to see what is available and how we access the model components and their documentation. To open the library, double click on Modelica in the Package browser.

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

1. **Blocks** with continuous and discrete input/output blocks such as transfer functions, filters, and sources.
2. **Constants** provides constants from mathematics, machine dependent constants and constants from nature.
3. **Electrical** provides electric and electronic components such as resistor, diode, MOS and BJT transistor.
4. **Icons** provides common graphical layouts (used in the Modelica Standard Library).
5. **Math** gives access to mathematical functions such as sin, cos and log.
6. **Mechanics** includes a one-dimensional translational and rotational components such as inertia, gearbox, planetary gear, bearing friction and clutch.
7. **SIunits** with about 450 type definitions with units, such as Angle, Voltage, and Inertia.
8. **Thermal** provides models for heat-transfer.

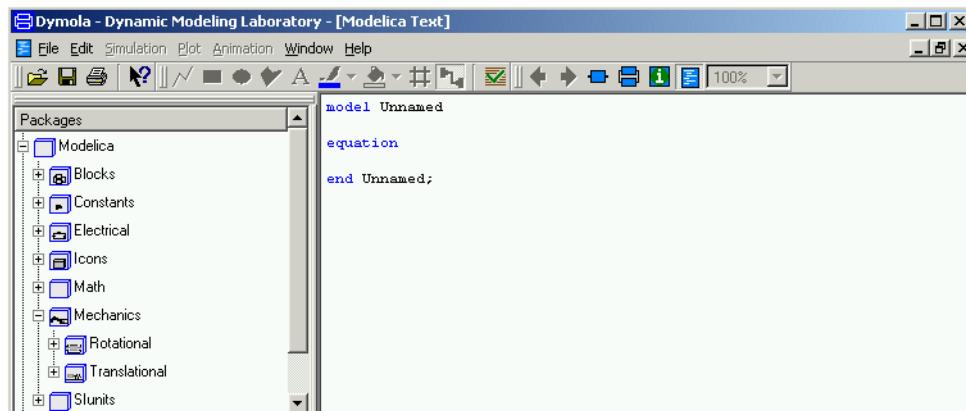
To get documentation for the entire Modelica Standard Library, place the cursor on Modelica, press the right mouse button and select Info. Your web browser is directed to an html file containing documentation for Modelica. This documentation has been generated automatically from the Modelica description 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. To have a list select 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, click on Mechanics (double-click shows its text).

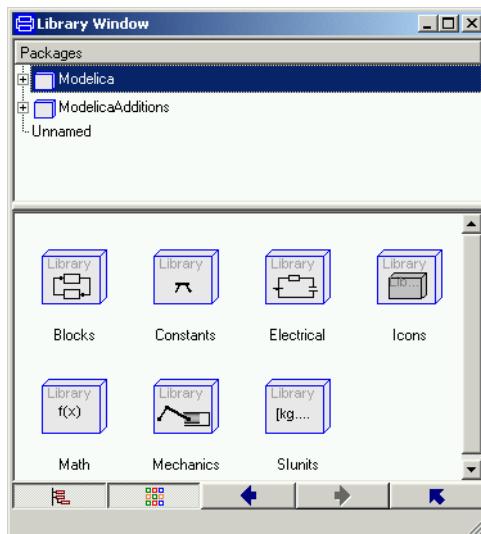
Opening Modelica.Mechanics.



To get documentation on Modelica.Mechanics, as previously demonstrated, place the cursor on Mechanics, press the right mouse button and select Info.

Besides using the package browser of the Dymola window, it is also possible to open a library browser. It is done by either selecting Window/New Library Browser or using the right mouse button and selecting Show Library Window.

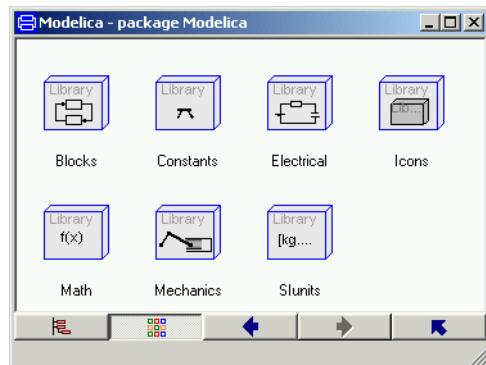
A library window displaying the Modelica Standard Library.



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

It is possible to close the package browser by toggling the button to the bottom left.

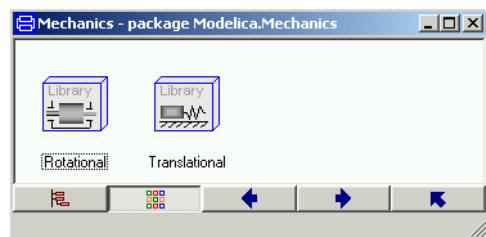
A library window displaying the components of the Modelica Standard Library.



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.

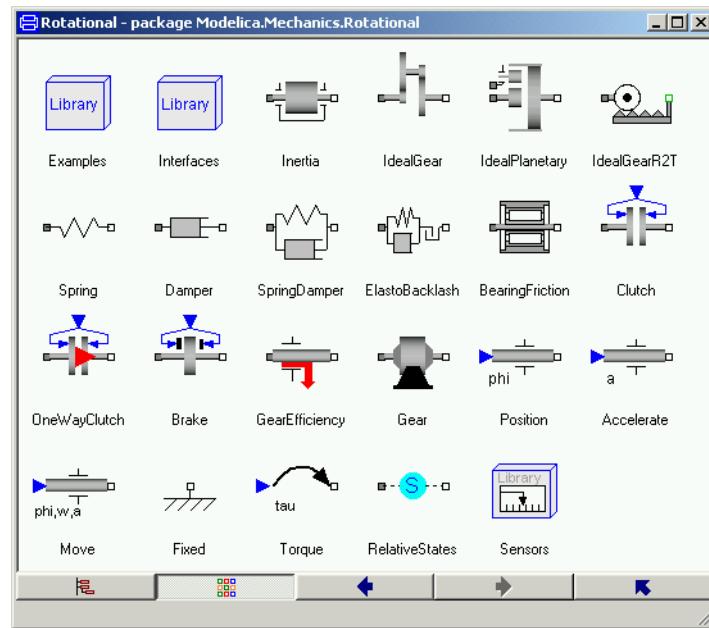
Open Modelica.Mechanics.Rotational in the library window by first double clicking on the icon for Mechanics

Modelica.Mechanics.



and then on the icon for Rotational. It contains components for rotating elements and gearboxes, which are useful for our modeling of the electrical motor drive.

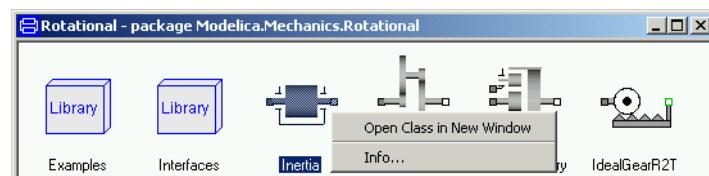
The rotational mechanics library window.



The Info for Modelica.Mechanics.Rotational contains important information on design principles behind the library and a list of components. A quick scan of the list indicates that the model Inertia may be of interest for us.

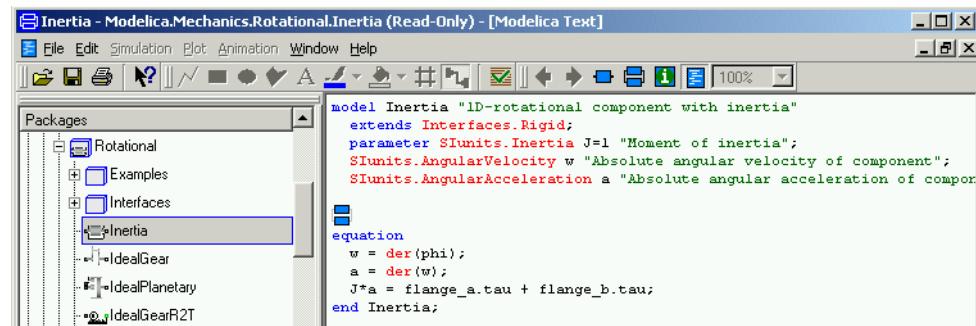
Select Inertia and press the right mouse button for a context menu. Select Info to get documentation for the model.

The context menu for a component.



To get a model window for Inertia, select its icon in the library window and once again press the right mouse button, select Show in New Window and a window for the model Inertia is created.

Mathematical definition of a rotational inertia.



The screenshot shows the Dymola software interface with the title bar "Inertia - Modelica.Mechanics.Rotational.Inertia (Read-Only) - [Modelica Text]". The menu bar includes File, Edit, Simulation, Plot, Animation, Window, and Help. The toolbar has icons for Open, Save, New, Copy, Paste, and others. The left pane is a "Packages" browser with nodes: Rotational, Examples, Interfaces, and Inertia (which is selected). The right pane displays the Modelica code for the Inertia component:

```
model Inertia "1D-rotational component with inertia"
  extends Interfaces.Rigid;
  parameter SIunits.Inertia J=1 "Moment of inertia";
  SIunits.AngularVelocity w "Absolute angular velocity of component";
  SIunits.AngularAcceleration a "Absolute angular acceleration of component";
  equation
    w = der(phi);
    a = der(w);
    J*a = flange_a.tau + flange_b.tau;
end Inertia;
```

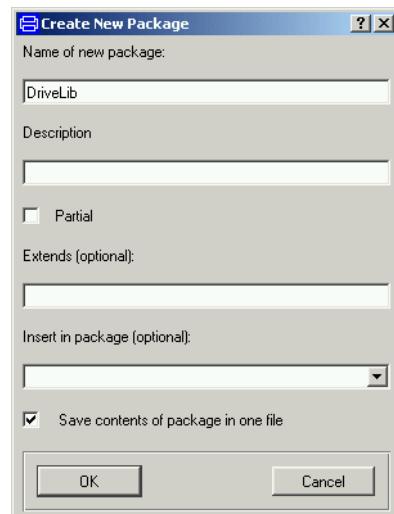
Switch to the Modelica Text representation, where you find Euler's equation as the last equation.

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.

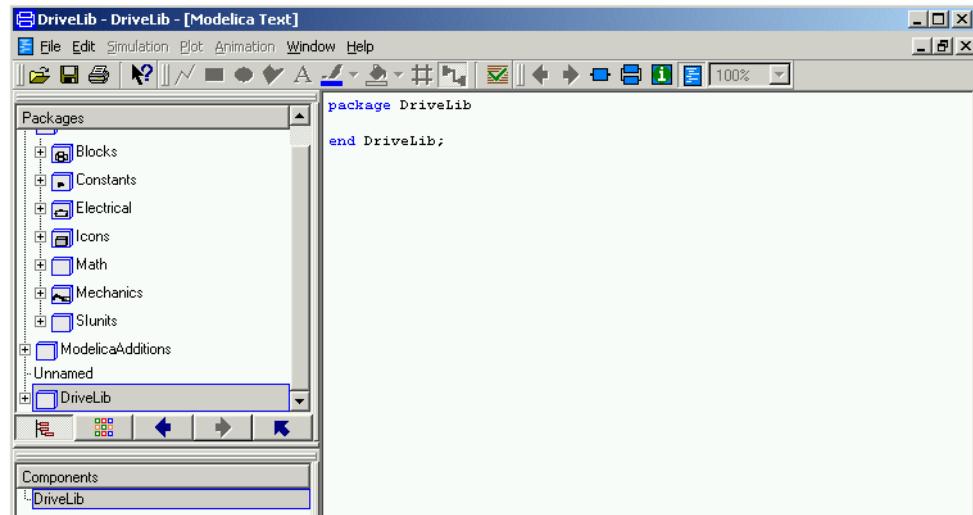
Creating a library for components

It is a good idea to insert developed components into a library. It is a good way to keep track of components and it supports also the drag and drop feature when you will use the models as components. Let us collect all developed components in a library called DriveLib. To create it, go to the Dymola window, and select File/New/Package and a dialog is opened.

Creating a new Modelica package.



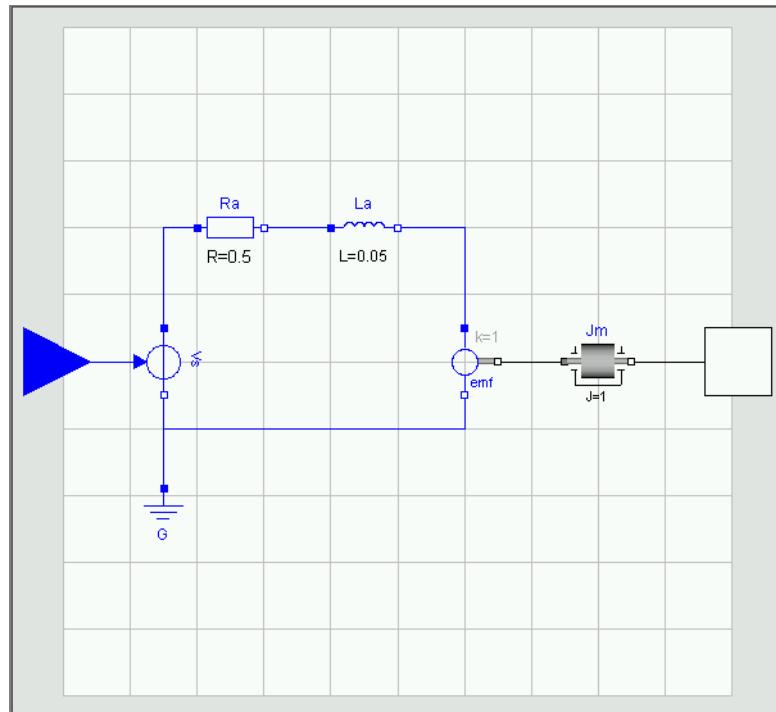
Enter DriveLib as the new name of the package and click OK, and 'Accept' in the information-window.



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

Creating a model for an electric DC motor

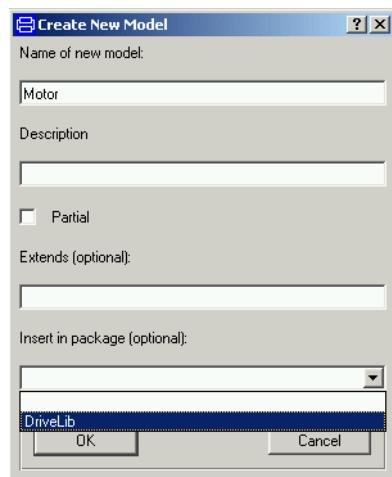
An electrical DC motor.



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 such as Modelica 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, release the button and the text DriveLib will appear in the slot.

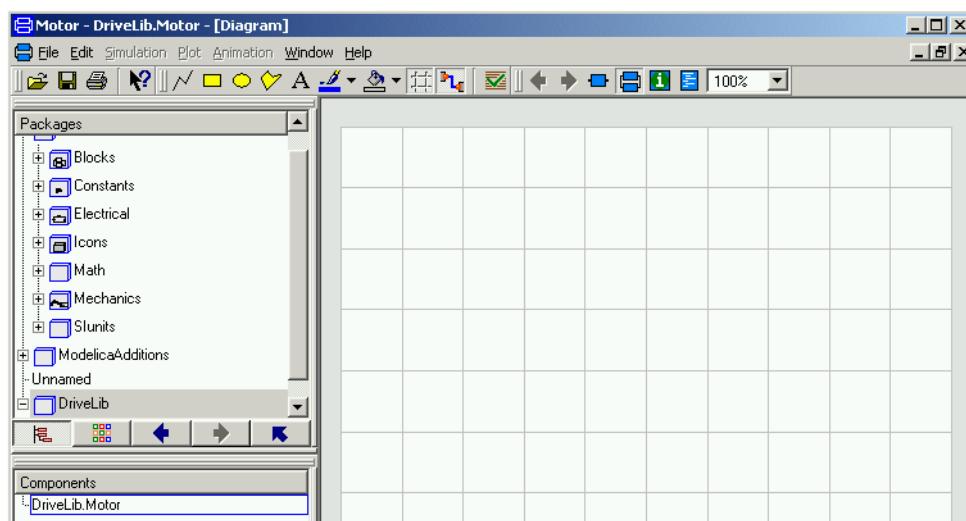
Inserting Motor in DriveLib.



Click on OK.

Opening DriveLib in the package browser shows that it has a component Motor as desired.

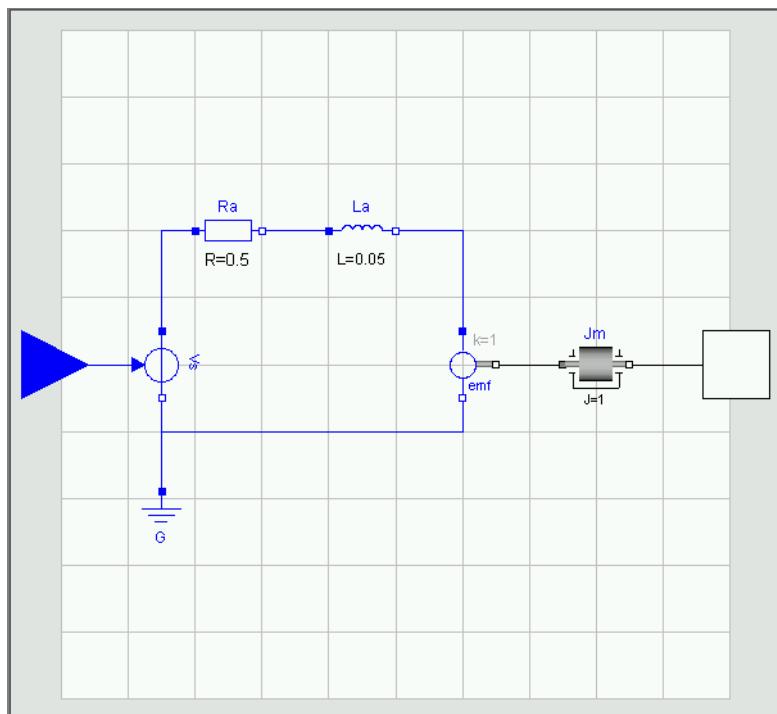
An empty Motor model.



The model window now contains an empty Motor model. The diagram 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.

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

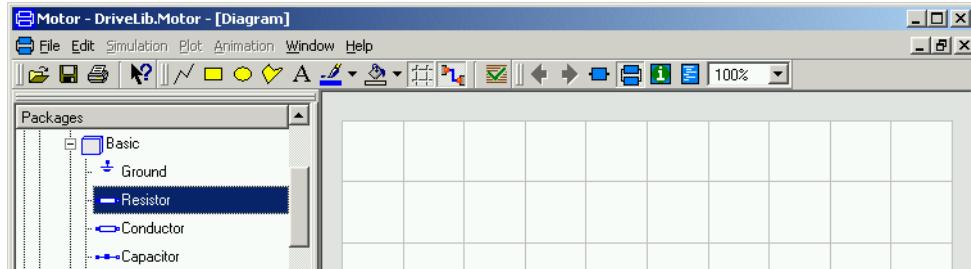
The finished motor model with all components.



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.

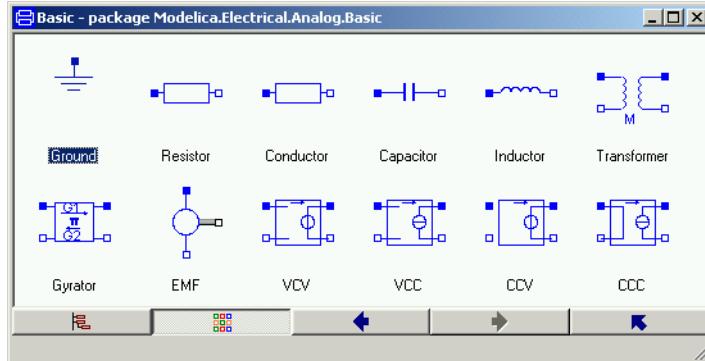
To drag from package browser, open in turn Modelica, Electrical, Analog and Basic. Note that title of the Dymola window is still DriveLib.Motor and also the component browser has DriveLib.Motor as top level to indicate that we are editing the motor model.

About to drag a resistor from the package browser.



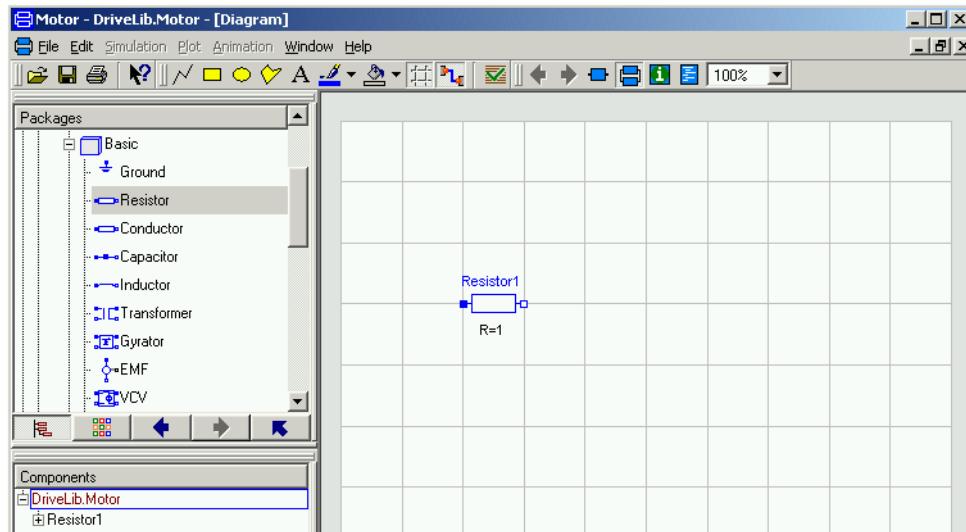
You can also drag from a library window. If you do not have a library window for it then open one. Go to Modelica.Electrical.Analog.Basic.

The basic analog components library window.



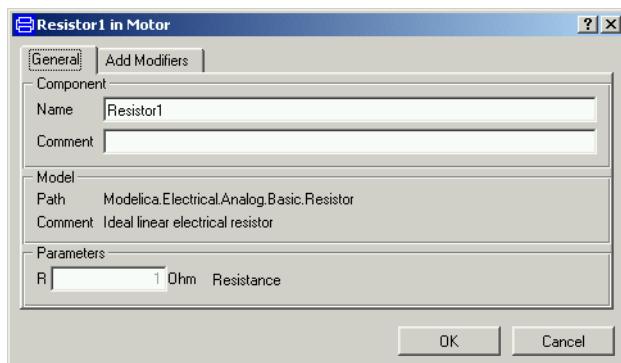
Drag a resistor from Basic to the Motor window and place it as shown above. The component browser displays that Motor has a component Resistor1.

Inserting a resistor component.



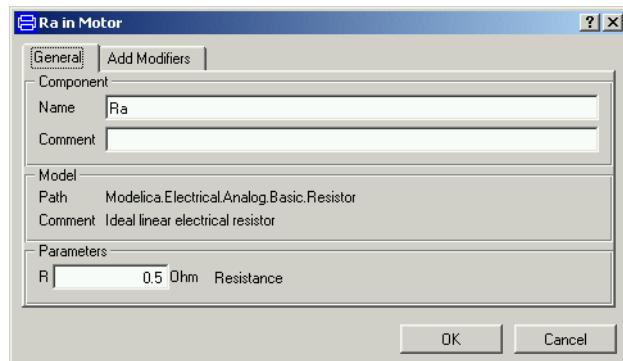
When inserting a component is given an automatically generated name. The name may be changed in the parameter window. Double click on the component, to get its parameter window. The parameter window can also be reached by pointing on the component and pressing the right mouse button and select Parameters.

The parameter window of a resistor with default settings.



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

The parameter window 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. Name it emf.

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. Get the Info for SignalVoltage displayed in your web browser (point on the icon and use the right mouse button menu). The documentation shows that SignalVoltage extends the model Modelica.Electrical.Analog.Interfaces.OnePort. Click on that link. The documentation shows that the pin p is a filled blue square. To get the proper sign we would like to have pin p in the top position. To flip the component, use Edit/Flip Vertical.

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

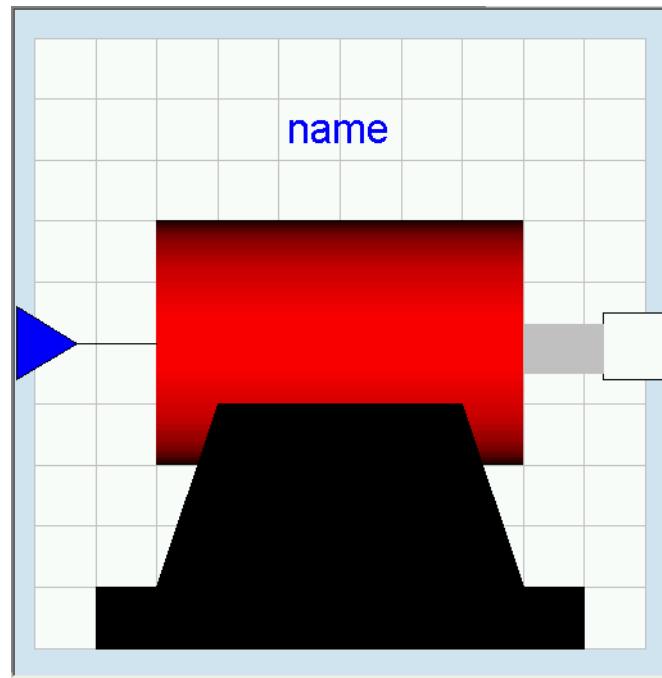
Now all model components are in place. Components are connected by drawing lines between connectors. To connect the resistor to the inductor, point at the right connector of the resistor (the small white square) and drag to the left connector of the inductor. The resistor and the inductor are now connected 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. Select the connection, 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.

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 environ-

ment. 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 automatic 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 Connect Mode (to the right of the Toggle Grid button).

Click on the toolbar button (to left of the button for activating the Diagram view) to find that you also there can see icons for the connectors. Let us draw an icon for the motor model. One design is shown below.

The icon of the electrical DC motor.



To draw it, we will use the toolbar for editing graphics.

Toolbar for editing.

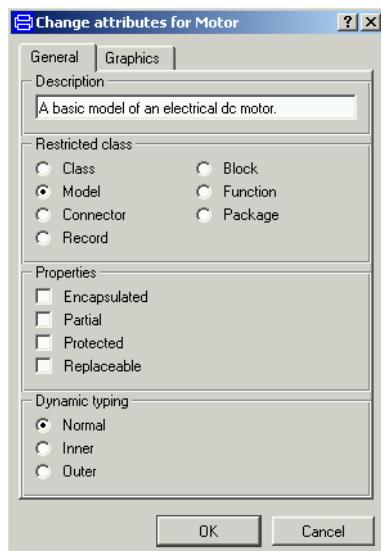


Start by drawing the big red cylinder (shaded rectangle); Click the Draw rectangle button (yellow rectangle) and lay out a rectangle. Let it be selected. Click on the arrow to the right of the Fill Color button. Select Colors... and then select a red color. Click OK. To select the gradient, click once again on the arrow to the right of the Fill Color button. Select Gradient/Horizontal. Draw the rest of the parts using draw rectangle or draw closed 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. The %-sign has the magic function that when the model is used, the actual component name will be displayed.

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 dialogue.

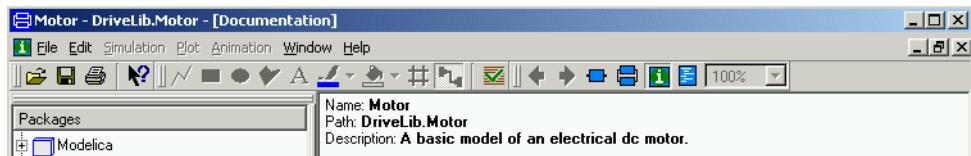
Model attributes.



Enter a description and click OK.

A longer documentation can be provided in the following way. Click on the toolbar button for Documentation (the button between the Diagram button and the Modelica Text button).

Documentation View.



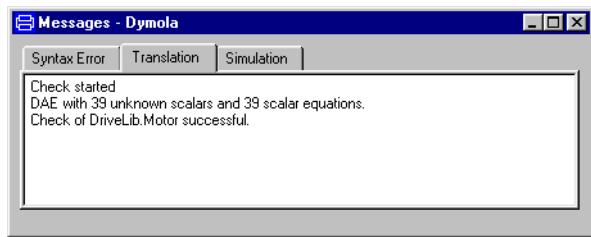
To enter a description, put the cursor in the window. Click right mouse button and select Edit Source. The description is given as html code. When done, click right mouse button and deactivate Edit Source.

We have now created the model. Save it.

Testing the model

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

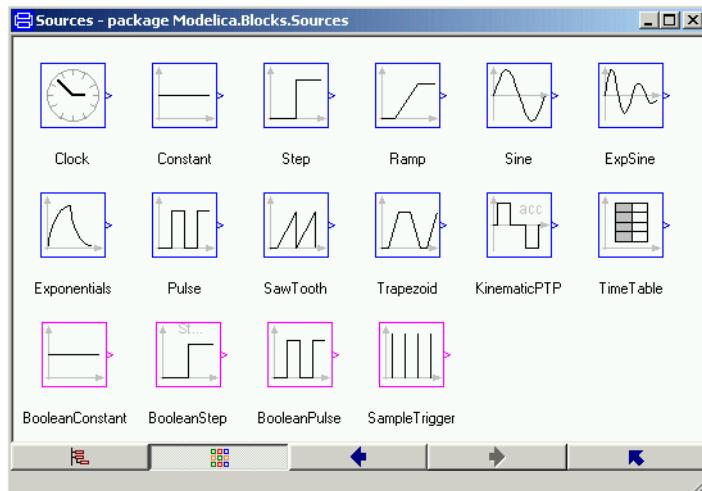
Checking the model.



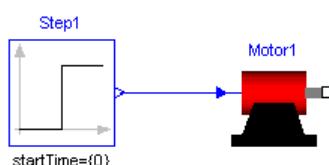
The connector inPort defines the voltage reference, and should be defined for the complete model, but is viewed as a known input to the model.

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 (select File/New/Model or the Edit/New Model in the context-menu for DriveLib) and insert it into DriveLib. It is good practice to keep test models. Use the package browser to drag over a Motor component from DriveLib to TestMotor. We need a source for the signal input to the motor. Signal sources are to be found in Modelica.Blocks.Sources.

Signal sources.



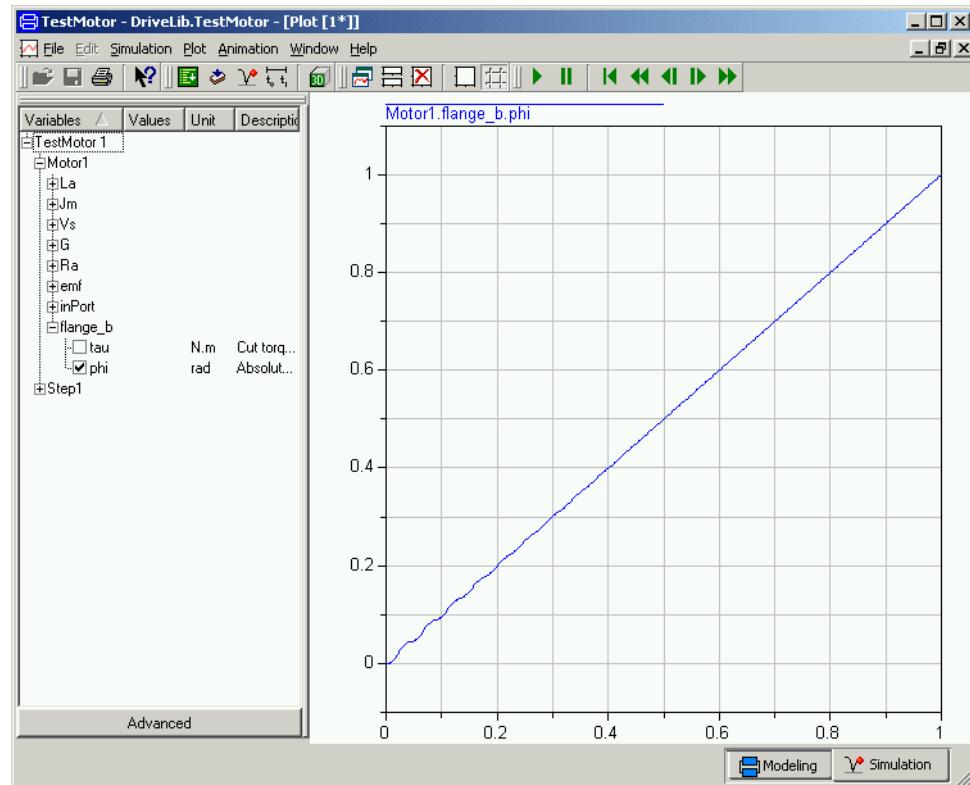
Drag, for example, over Step to the model window and connect it to the motor.



Now it is time to simulate. Click on the tab for Simulation. Click on the toolbar button Simulate to start a simulation.

To inspect the result, we will first look at the angular position of the motor, Motor1.flange_b.phi. Open Motor in the plot selector by clicking on the + sign. Open the flange_b and tick phi.

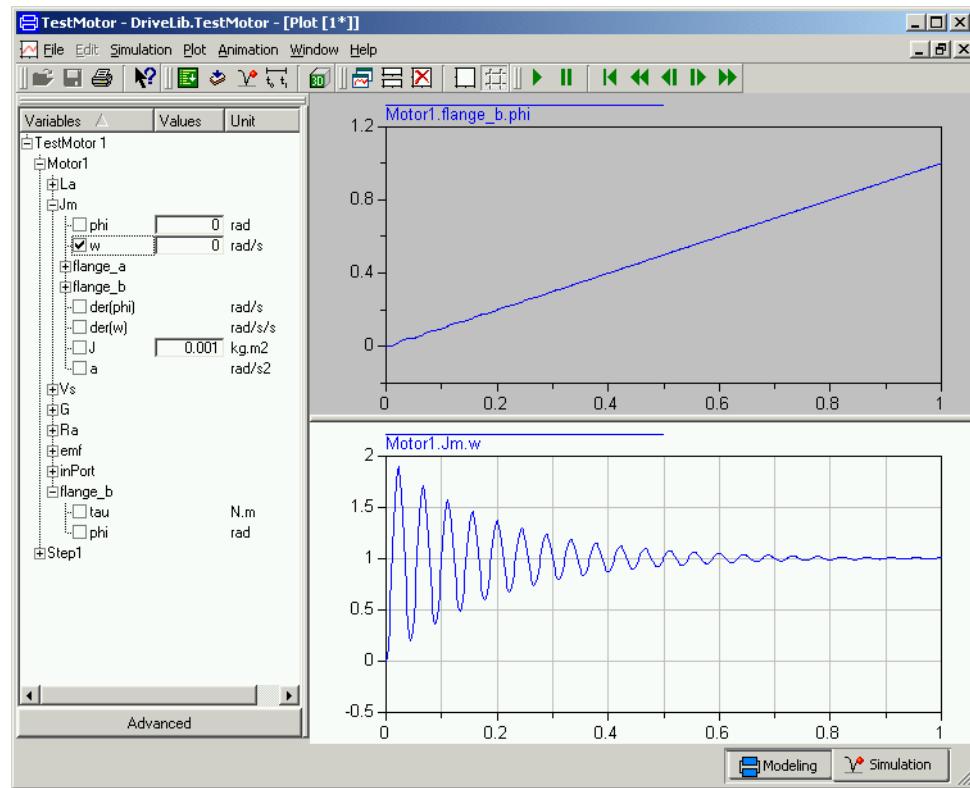
Angular position.



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 Motor1.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. It is just to 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. Tick Motor1.Jm.w and the result shown below is obtained. Selecting a diagram makes it active. Selecting Plot/Delete Diagram removes the diagram.

Angular velocity.



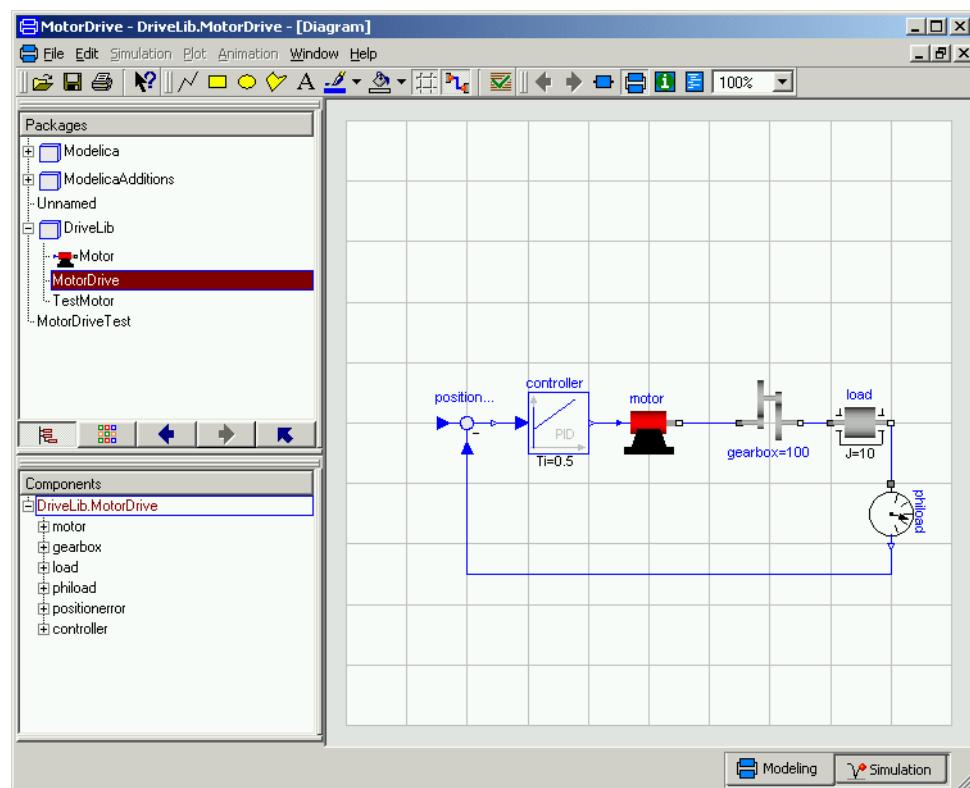
Creating a model for the motor drive

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 name of the components are given in the component browser at the lower left.

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

For the meaning of ratio for the gearbox model, please, consult Info for the model. Set ratio to 100 as indicated. It means that the motor load rotates 100 times slower than the DC motor. The library Modelica.Mechanics.Rotational.Sensors contains a sensor for angles.

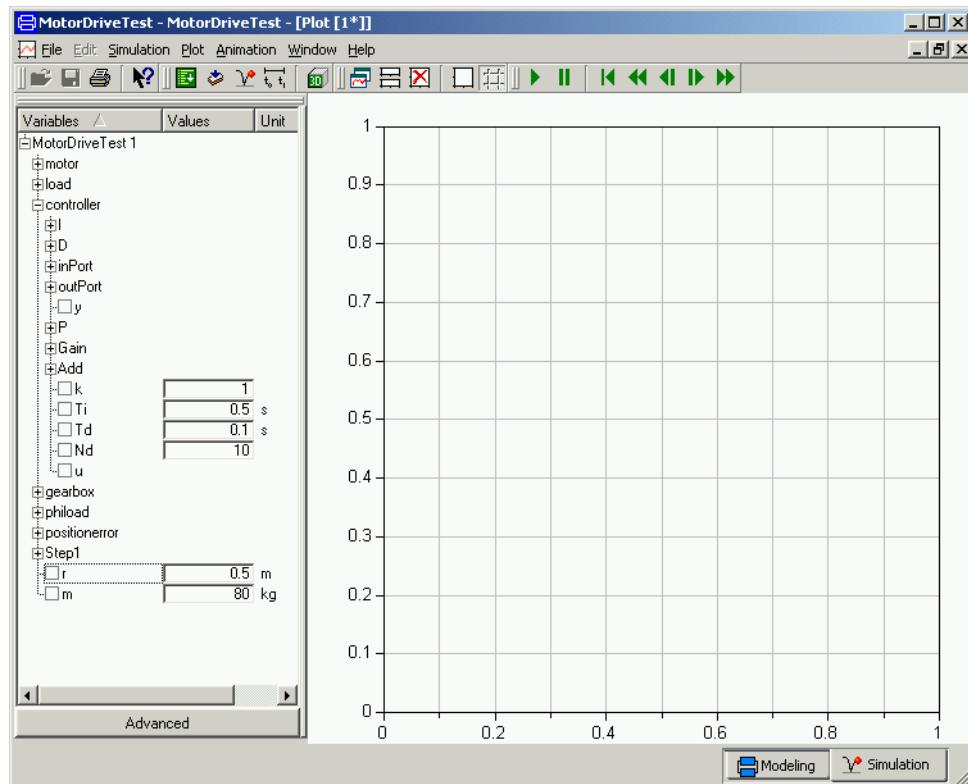
The completed motor drive.



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 dialogue 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 over a component Step from Modelica.Blocks.Sources and connect it. Save the model.

A model can be used for different tasks. One is tuning the controller manually. Click on the tab for Simulate. Translate the model MotorDriveTest. The PID controller has four parameters: k , T_i , T_d and N_d .

Tuning the controller.



There are many ways to tune a PID controller. One is to disable the integrator part by setting a large value for T_i , say 1000 here. Disable also the derivative part by setting T_d small, say 0.001. Simulate for 10 seconds. The step response for $k = 1$ is very slow. Increase it to find out what happens. We leave the problem of tuning the controller to the interested reader.

Parameter expressions

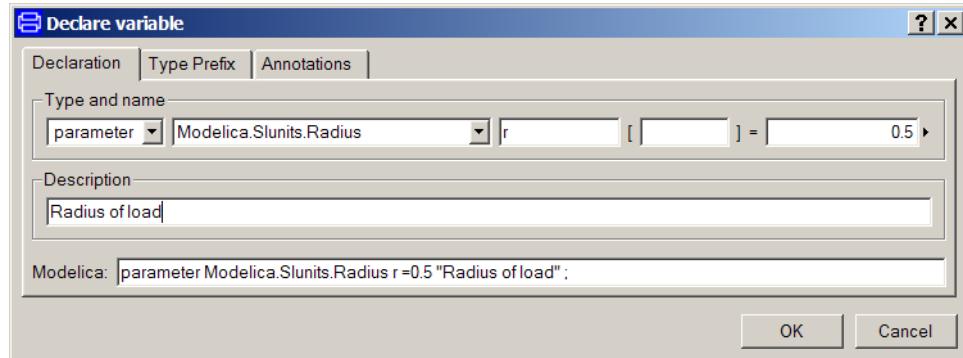
Modelica supports parameter propagation and parameter expressions, which means that a parameter can be expressed in terms of others. Assume that the load is a homogeneous cylinder rotating long its axis and we would like to have its mass, m , and radius, r , as primary parameters on the top level. The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

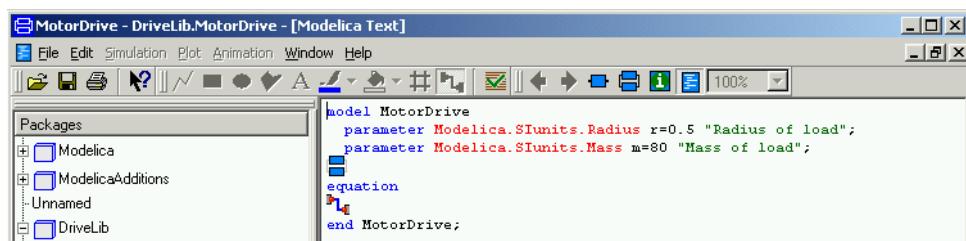
The parameters and variables are more than real numbers. They are physical quantities. The Modelica standard library provides type declarations for many physical quantities. Open Modelica.SIunits in the package browser. For the parameter r , which is a radius, it is natural to declare it as being of type Radius. To find it enter R and the browser goes to the first component starting with R. 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 preselected. Click OK. A menu to declare a variable pops up. Complete the declaration:

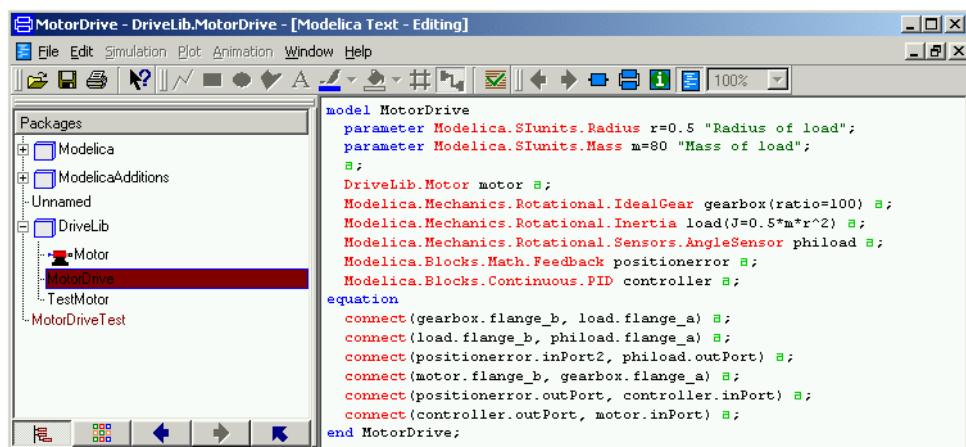


Click OK and the text appearing in the bottom row is inserted into the Modelica text window. The parameter m is defined in an analogue way.

Parameter declarations added to motor drive.



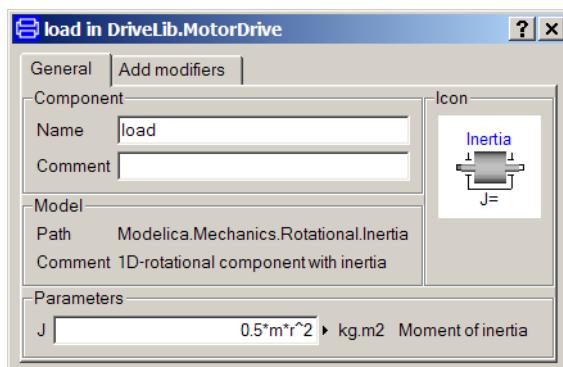
In Modelica Text representation above, the components and connections are indicated by icons. It is possible have them expanded textually. Click the right mouse button and select Expand/Show components and connect.



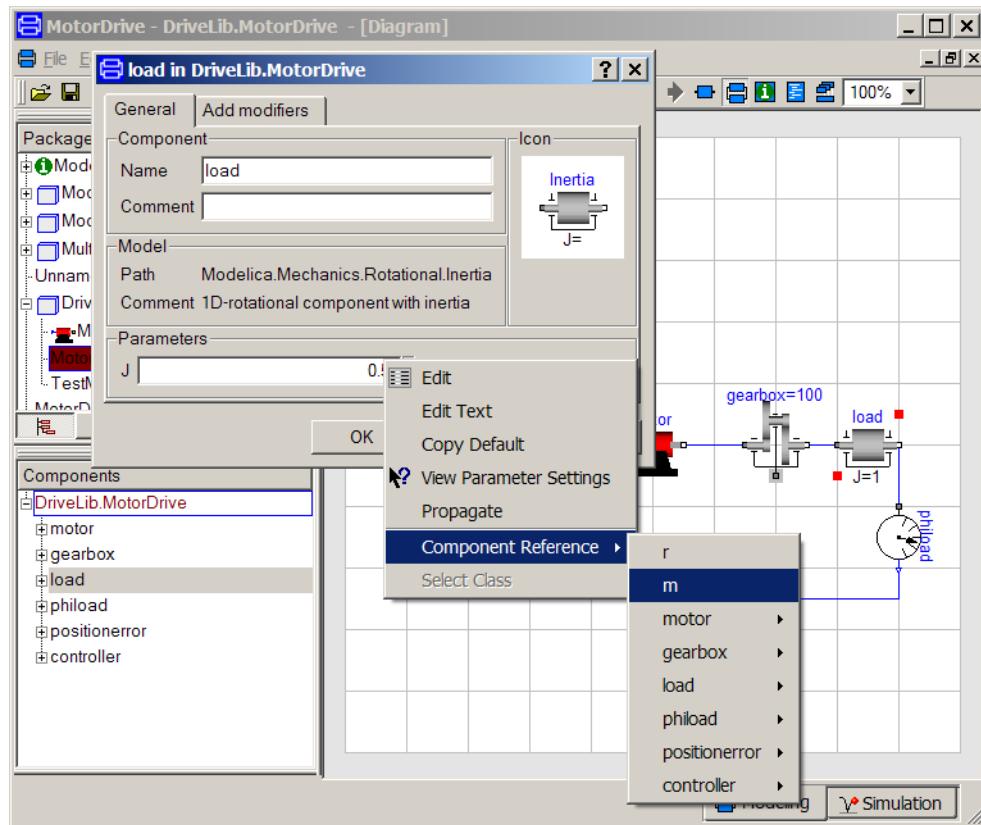
It is also possible to expand the annotations such as the graphics for the icon of the model, the positions and sizes of the icons of the components, the path of the connections etc., by clicking the right mouse button and select Expand/Show entire text. However, we refrain from showing it in this document.

Activate the diagram representation. Double click on the load icon to open the parameter window.

Binding a parameter to an expression.

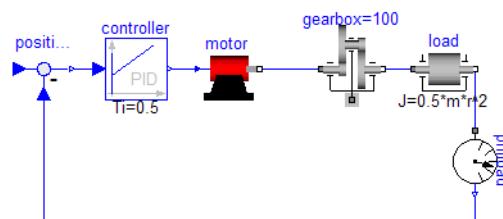


Click in the value field of J and enter the expression for J. Click OK. The model window now displays the expression for the load inertia. When entering the expression, you are sometimes not sure about the exact name of the variables names, for example is the radius called r, r0 or r1? The problem is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



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.

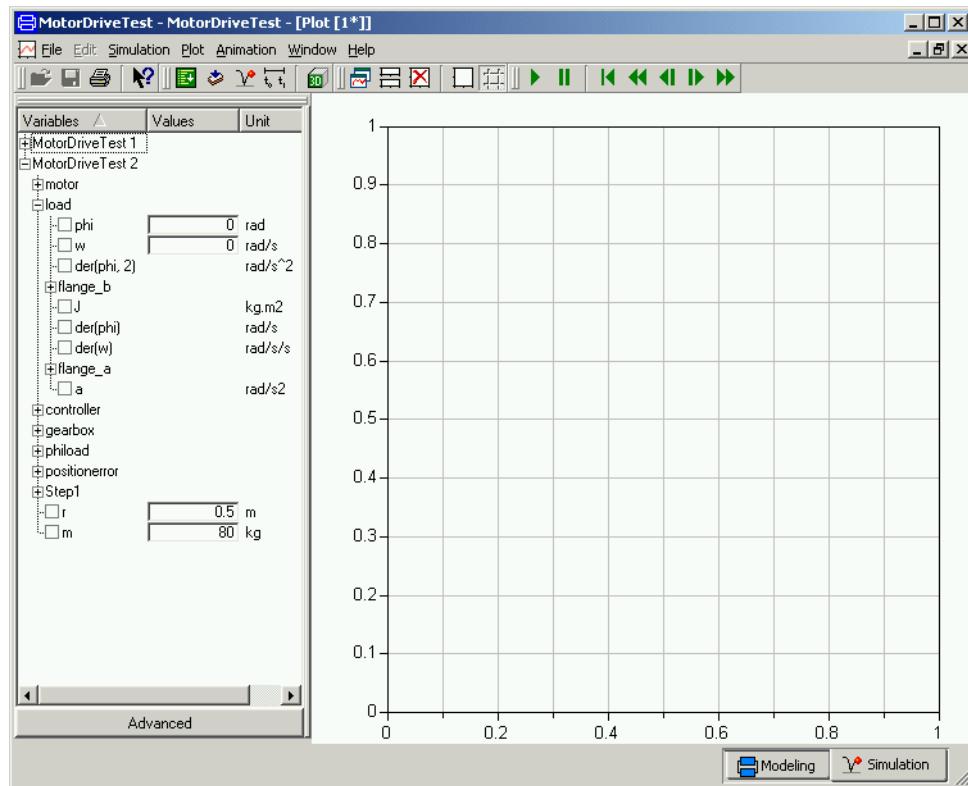


Switch to Simulation mode.

Translate MotorDriveTest.



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

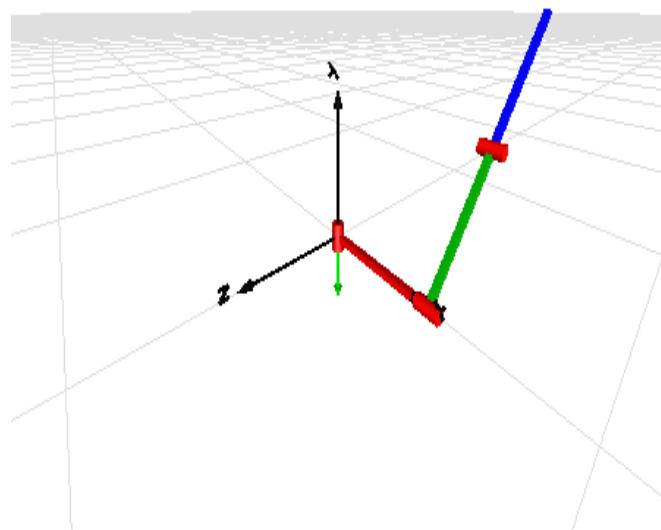


The parameters r and m can be set interactively between simulation runs, but not J , because it is no longer a free parameter, because there is an expression binding it to r and m .

Building a mechanical model

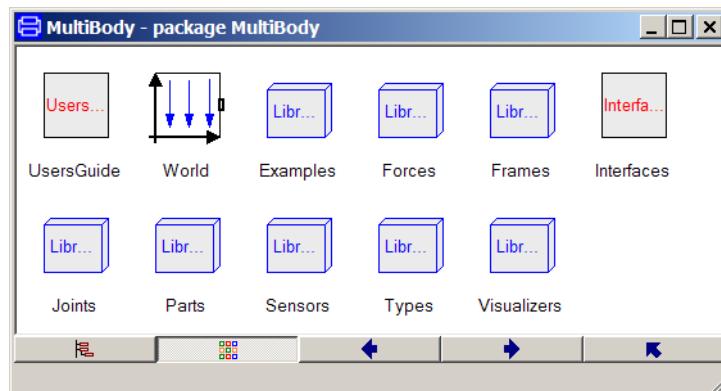
We will now develop a more complex model, a 3D mechanical model of a pendulum called a Furuta pendulum. It has an arm rotating in a horizontal plane with a single or double pendulum attached to it, see below.

The Furuta pendulum.



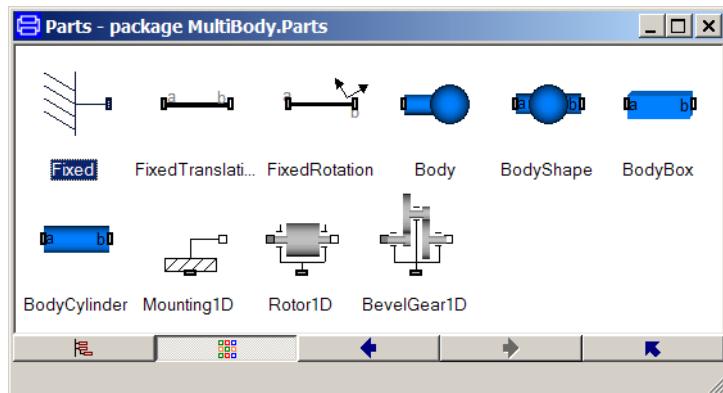
Start Dymola. The package browser shows the library MultiBody. This library includes 3D mechanical components such as joints and bodies, which can be used to build a model of the Furuta pendulum. Use the package browser to open MultiBody. You may also create a new library window. Select MultiBody in the package browser and click the right mouse button and select Open Library Window.

The MBS library window.

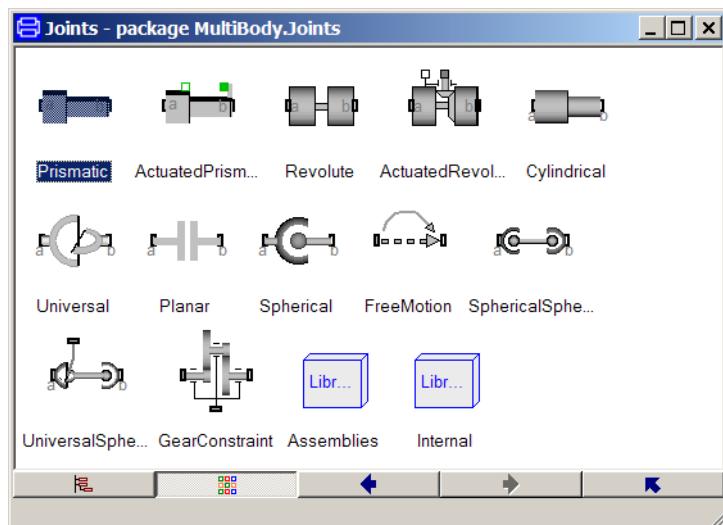


To build the Furuta pendulum, you will need to use the Parts and Joints sub-libraries. Open them by double-clicking.

The MBS parts library window.



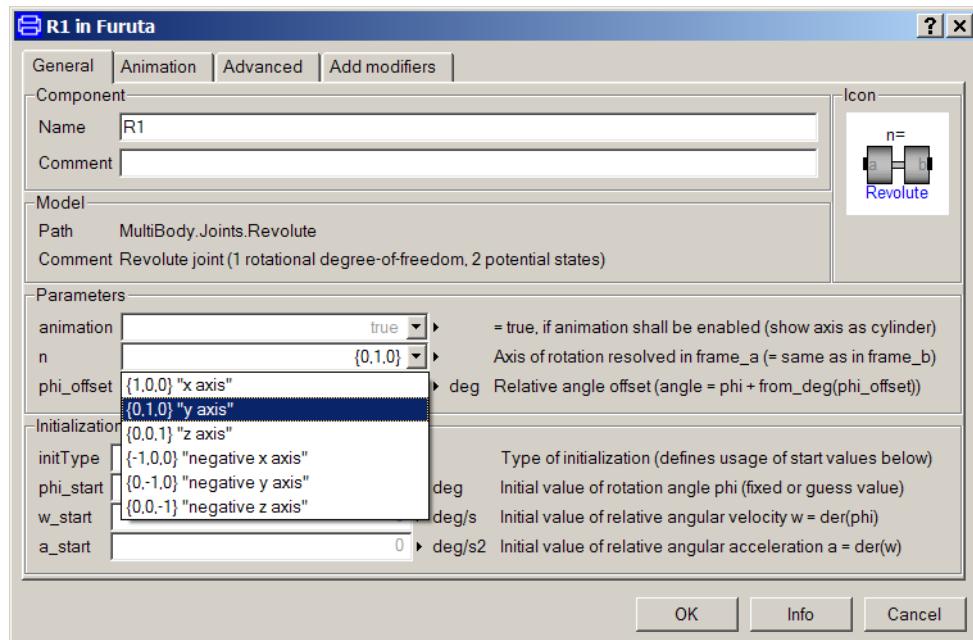
The MBS joints library window.



Select File/New/Model and give the name Furuta.

The first step in building an MBS model is to define an inertial system. Drag the World icon onto the Furuta 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 library window, but it is also convenient to drag from the package browser. 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”. When simulating, it is natural to try different initial configurations by setting angles and possibly angular velocities for the joints. Use the list of choices of initType to select the alternative “initialize generalized position and velocity variables”. The initial value of the angle is specified by setting the parameter phi_start and the initial velocity is specified by setting w_start. These two values are easy to change between simulation runs. 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 over a component. 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 left of the value field of r and an vector editor pops up. Enter 0.5 in the first field. Click OK. Since you did not set all value explicitly, you will get question if default values should be used. Click “Use defaults”. The width and height will be 0.1 by default.

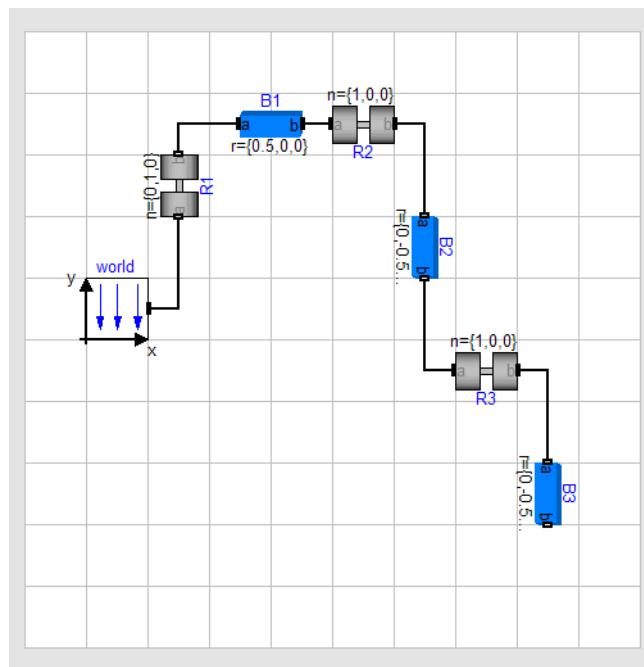
To get nicer animation, you can set different colors for the bars. For example, use the list of choices of color to make the bar red.

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 $r=\{0, -0.5, 0\}$.

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

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

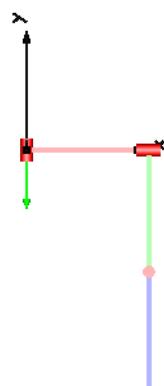
The diagram of the Furuta pendulum.



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.

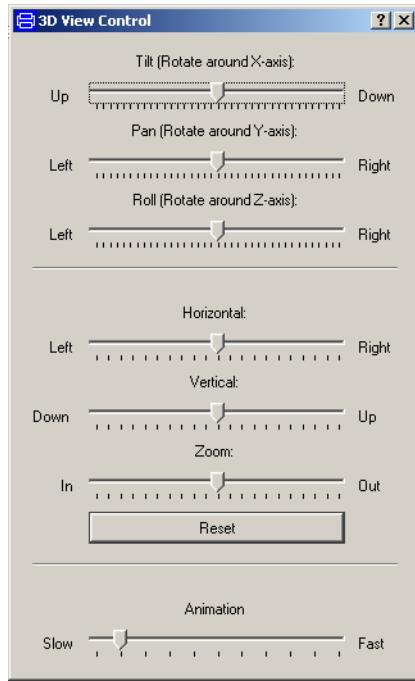


When building 3D mechanical models, it is possible to get visual feedback to check that the mechanical system is put together as intended. Click on the toolbar button Visualize. The animation window shows the initial configuration of the system.



Translate the model. In the plot dialogue, open R2 and enter a value for phi_start, say 170 degrees, and simulate for 5 seconds. View the pendulum in the animation window; you may want to adjust the perspective by using Setup/3D View Control.

3D View Control.



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.)

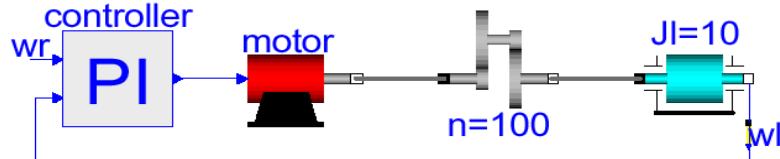
INTRODUCTION TO MODELICA

Introduction to Modelica

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).

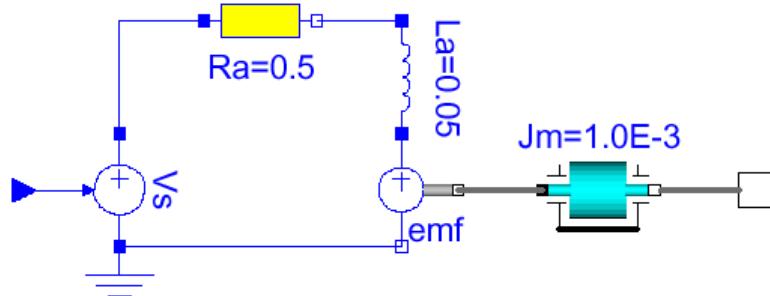
A Modelica model of the motor drive.

```
model MotorDrive
  PI controller;
  Motor motor;
  Gearbox gearbox(n=100);
  Shaft J1(J=10);
  Tachometer wl;

equation
  connect(controller.out, motor.inp);
  connect(motor.flange, gearbox.a);
  connect(gearbox.b, J1.a);
  connect(J1.b, wl.a);
  connect(wl.w, controller.inp);
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 `Gearbox gearbox(n=100);` declares a component `gearbox` of class `Gearbox` and sets the default value of the gear ratio, `n`, to 100.

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.

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.

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 $\text{Pin1.v} = \text{Pin2.v}$ and $\text{Pin1.i} + \text{Pin2.i} = 0$. 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.

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`, that has two pins, p and n, and a quantity, v, that 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.

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.

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 submodels on explicit state-space form

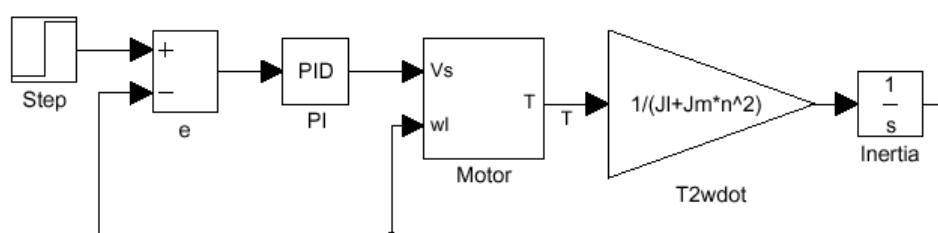
$$\frac{dx}{dt} = f(x, u)$$

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

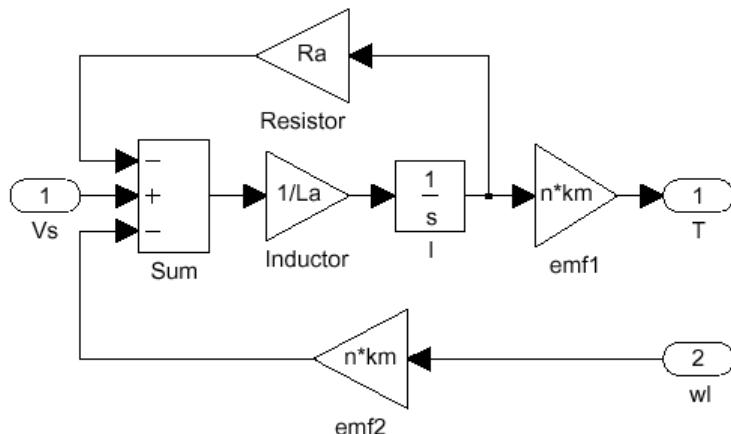
where \mathbf{u} is input, \mathbf{y} is output and \mathbf{x} is the state. It is rare that a natural decomposition into subsystems leads to such a model. Often a significant effort in terms of analysis and analytical transformations is needed to obtain a problem in this form. It requires a lot of engineering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

To illustrate the difficulties, a Simulink model for the simple motor drive (see page 73) 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.

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 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 73.

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

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

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

Advanced modeling features

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

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. The use of component arrays for spatial discretization when modeling heat exchangers is illustrated in Mattsson *et al.* (1998).

Class parameters

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

As an example assume that we would like to replace the PI controller in the motor drive model on page 73 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 74 a model MotorDrive2 with redeclared controller is described as

```
model MotorDrive2 =
  MotorDrive(redeclare AutoTuningPI 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 PI, i.e., have compatible connectors and parameters. The type system of Modelica is greatly influenced by type theory, Abadi and Cardelli (1996), in particular the notion of subtyping (the structural relationship that determines type compatibility) which is different from subclassing (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 PI 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 PI 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
  input Real ref;
  input Real inp;
  output Real 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);

```

The use of model class parameters to support machine-medium decomposition is illustrated in Mattsson *et al.* (1998), Ernst *et al.* (1997) and Tummescheit and Eborn (1998).

Algorithms and functions

Algorithms and functions are supported in Modelica for modeling parts of a system in procedural programming style. Modelica functions have a 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;

```

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.

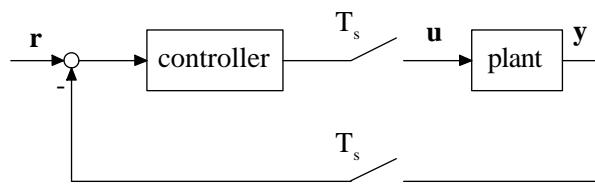
A hybrid Modelica model is described by a set of synchronous differential, algebraic and discrete equations leading to deterministic behaviour and automatic synchronization of the continuous and discrete parts of a model. The consequences of this view are discussed and demonstrated at hand of a new method 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 does not exist a single widely accepted description form. Instead, many formalisms are available, e.g., finite automata, Petri nets, statecharts, 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 realtime systems and for verification purposes.

Synchronous equations

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

Sampled data system.



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

$$\frac{dx_p}{dt} = f(x_p, u)$$

$$y = g(x_p)$$

is controlled by a digital linear controller

$$x_c(t_i) = Ax_c(t_i - T_s) + B(r(t_i) - y(t_i))$$

$$u(t_i) = Cx_c(t_i - T_s) + D(r(t_i) - y(t_i))$$

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

```

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

```

```

Real xp[nx] "plant state";
equation
  der(xp) = f(xp, u); // plant
  y = g(xp);
  when sample(0,Ts) then // controller
    xc =A*pre(xc) + B*(r-y);
    u =C*pre(xc) + D*(r-y);
  end when;
end SampledSystem;

```

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

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

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

```

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

```

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

The synchronous principle has several consequences: First, the evaluation of the discrete equations is performed in zero (simulated) time. In other words, time is abstracted from the computations and communications, see also Gautier *et al.* (1994). Second, in order that the unknown variables can be uniquely computed it is necessary that the number of active equations and the number of unknown variables in the active equations at every time instant are identical. This requirement is violated in the following example:

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

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

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

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

In this case the two when clauses are evaluated in the order of appearance and the second one gets 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.

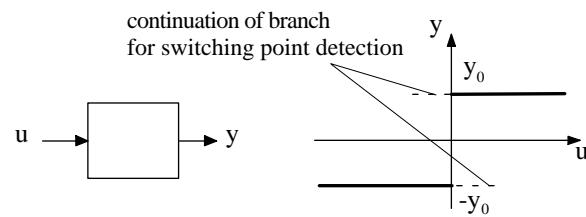
Handling hybrid systems in this way has the advantage that the synchronization between the continuous time and discrete event parts is automatic and leads to a deterministic behaviour without conflicts. Furthermore, some difficult to detect errors of other approaches, such as deadlock, can often be determined during translation already. 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).

The disadvantage is that the types of systems which can be modeled is restricted. For example, general Petri nets cannot be described because such systems have non-deterministic behaviour. For some applications another type of view, such as a process oriented type of view or CSP, may be more appropriate or more convenient.

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. This technique was developed by Cellier (1979). For details see also Eich-Soellner and Führer (1998).

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

In some situations, relations do not introduce discontinuities or non-differentiable points. Even if such points are present, their effect may be small, and it may not affect the integration by just integrating over these points. Finally, there may be situations where a literal

evaluation of a relation is required, since otherwise an “outside domain” error occurs, such as in the following example, where the argument of function `sqrt` to compute the square root of its argument is not allowed to be negative:

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

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

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

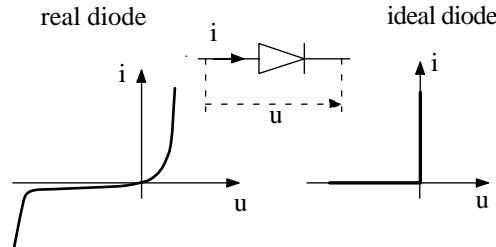
Modelica has a set of additional operators, such as `initial()` and `terminal()` to detect the initial and final call of the model equations, and `reinit(...)` to reinitialize a continuous state with a new value at an event instant. For space reasons, these language elements are not discussed. Instead, in the next section some non-trivial applications of the discussed language elements are explained.

Variable structure systems

Parametrized curve descriptions

If a physical component is modelled 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.

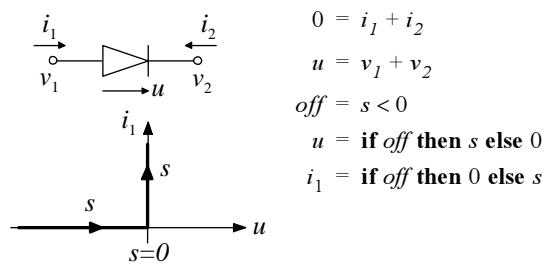


As a typical example, a diode is shown in the figure above, where i is the current through the diode and u is the voltage drop between the pins of the diode. The diode characteristic is shown in the left part of the figure. If the detailed switching behaviour is negligible with regards to other modeling effects, it is often sufficient to use the ideal diode characteristic shown in the right part of the figure, which typically give 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 has 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

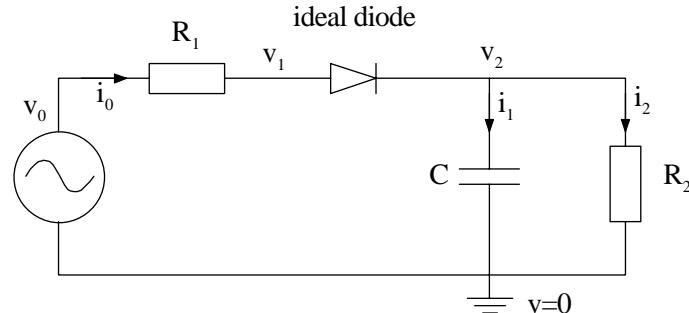
in the form $i = i(u)$ is no longer possible. This problem can be solved by recognizing that a curve can also be described in a parameterized form $i = i(s)$, $u = u(s)$ by introducing a curve parameter s . This description form is more general and allows us to describe an ideal diode uniquely in a declarative way as shown in the figure below.

Ideal diode model.



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

$$\begin{aligned}
off &= s < 0 \\
u &= v_1 - v_2 \\
u &= \text{if } off \text{ then } s \text{ else } 0 \\
i_0 &= \text{if } off \text{ then } 0 \text{ else } s \\
R_I \cdot i_0 &= v_0(t) - v_I
\end{aligned}$$

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

$$i_1 = i_0 - i_2$$

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

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.

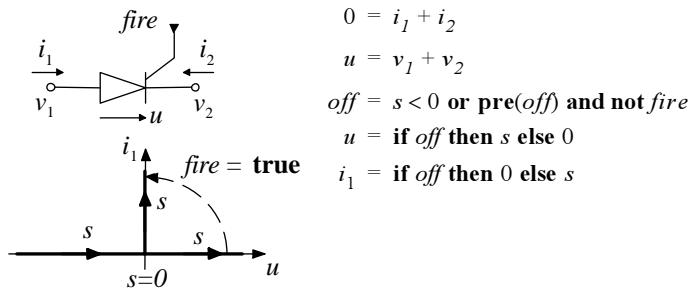
At an event instant, the first 5 equations are a mixed system of discrete and continuous equations which cannot be solved by, say, Gaussian elimination, since there are Real and Boolean unknowns. However, appropriate algorithms can be constructed: (1) Make an assumption about the values of the relations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

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

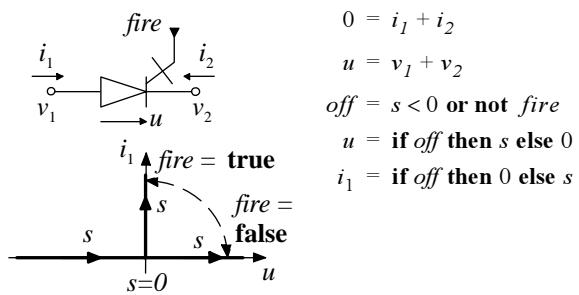
In the above example, both approaches can be simply applied, because there are only two possible values ($s < 0$ is false or true). However, if n switches are coupled, there are n relations and therefore 2^n possible combinations which have to be checked in the worst case.

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.



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 behaviour, see e.g., Barton (1992), Elmquist *et al.* (1993), and Mosterman *et al.* (1996), or by (b) using a complementarity 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 behaviour. 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).

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, $dx/dt = f(x, t)$, 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 $dx/dt = 0$ 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 subproblems 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.

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 utilised during the simulation.

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

1. Start values for variables
2. 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.

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
```

Initial equations and algorithms

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 fixed;
parameter Boolean steadyState;
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, $\frac{dx}{dt} = \mathbf{f}(x, t)$, *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 $x(t_0)$ and $\frac{dx}{dt}(t_0)$.

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

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

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, $\frac{dx}{dt}(t_0)$, $\mathbf{x}(t_0)$, $\mathbf{y}(t_0)$:

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

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.

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` and the start value being a literal. Setting parameters may of course influence an active start value bound to a parameter expression.

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

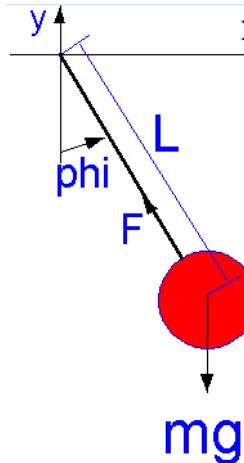
Non-linear algebraic loops

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 include 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. They will hopefully find the positive solution for x , because, it is closer to L than the negative solution.

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

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 `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 reparametrizations 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

1. Add a parameter P_0 to specify the power dissipation.
2. Set `fixed=false` for parameter R .
3. Add an initial equation section with $v*i = P_0$.

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);

```

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 ($= v(t_0 + \epsilon)$), but the **pre**-value of v ($= v(t_0 - \epsilon)$) 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 num-

ber 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 modeller to provide initial values for all discrete variables, Modelica only requires to specify initial values of discrete variables which influence the simulation result. Otherwise, a default value may be used.

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; $\text{der}(x) = 0$; While for the discrete state, xd, it means $\text{pre}(xd) = xd$; 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
```

Standard libraries

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 sublibraries

Constant	Mathematical and physical constants (pi, eps, h, ...).
Icons	Icon definitions of general interest.
Math	Mathematical functions (such as sin, cos).
SIunits	SI-unit type definitions (such as Voltage, Torque).
Blocks	Input/output blocks.
Electrical	Electric and electronic components.

Mechanics	1D-rotational and 1D-translational mechanical components.
Thermal	Components for Thermal systems.

The package ModelicaAdditions is a collection of libraries which is supplied by DLR (German Aerospace Research Establishment) to provide often needed components which are missing in Modelica Standard Library. It is planned to provide the components of ModelicaAdditions in an improved form in a future version of Modelica Standard Library. ModelicaAdditions includes currently of the following sublibraries

Blocks	Additional input/output control blocks.
HeatFlow1D	Made obsolete by Modelica.Thermal.HeatTransfer.
MultiBody	3D-mechanical systems.
Petri Nets	Components to model state machines and Petri nets.
Tables	Linear interpolation in one and two dimensions.

Summary

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 has been clearly demonstrated and compared to traditional block diagram modeling. It has also been pointed out that tools can incorporate computer algebra methods to translate the high-level Modelica descriptions to efficient simulation code.

References

- Abadi, M. and L. Cardelli (1996): *A Theory of Objects*. Springer-Verlag.
- Barton P.I. (1992).: *The Modelling 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 SI-DOPS+ 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.
- Cellier F.E. (1979): *Combined Continuous/Discrete System Simulation by Use of Digital Computers: Techniques and Tools*. Diss ETH No 6483, ETH Zürich, Switzerland.
- 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.
- Dymola. *Homepage*: <http://www.dynasim.se/>.
- Eich-Soellner E., and C. Führer. *Numerical Methods in Multibody Dynamics*. Teubner, 1998.
- 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., D. Brück, and M. Otter (1996): *Dymola — User's Manual*. Dynasim AB, Research Park Ideon, Lund, Sweden.
- 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 *Pro-*

ceedings of the IEEE International Conference on Information Visualisation. IEEE Computer Society, London, UK.

Ernst, T., M. Klose, and H. Tummescheit (1997) : “Modelica and Smile — A case study applying object-oriented concepts to multi-facet modeling.” In *Proceedings of the 1997 European Simulation Symposium (ESS'97)*. The Society for Computer Simulation, Passau, Germany.

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

Gautier T., P. Le Guernic, and O. Maffeis (1994): “For a New Real-Time Methodology.” Publication Interne No. 870, Institut de Recherche en Informatique et Systemes Aleatoires, Campus de Beaulieu, 35042 Rennes Cedex, France.

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

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

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

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

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

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

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

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

Mattsson, S. E., H. Elmquist, 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.
- 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.
- 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.

DEVELOPING A MODEL

Developing a model

This chapter describes the Dymola environment for developing models. The general aspects of the Dymola main window are described first, followed by a more detailed command reference. Simulation of the developed model is described in “Simulating a model” on page 147.

General concepts

Window types

Dymola uses two types of windows (the main Dymola window and the library window), that support different kinds of operations and also have different visual appearance.

Window modes

The main Dymola window can operate in two modes, Modeling and Simulation. The mode of the window controls which operations are available and the default appearance of the window. Going back to a previous mode restores its layout.

This chapter discusses the graphical editor, i.e. the modeling mode. The simulation mode is described in “Simulating a model” on page 147.

The mode of the window is set by clicking on the tabs at the lower-right corner of the Dy-mola window. Alternatively the keyboard shortcuts **Ctrl+F1** and **Ctrl+F2** or the Window/Mode menu can be used, see “Window menu” on page 142.

Window mode tabs.



Modeling Changes the window to modeling mode. This mode is used to browse models, compose new models or to change existing models.

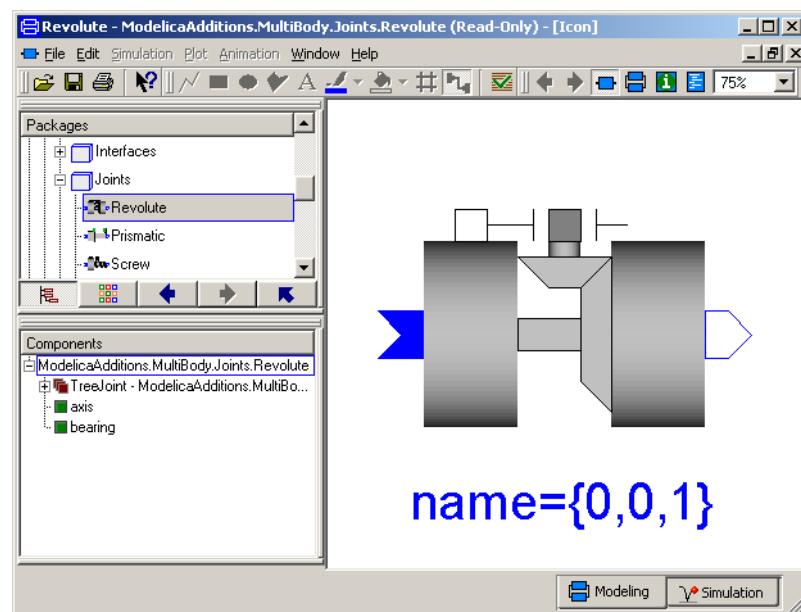
Simulation Changes the window to simulation mode. This mode is used to set up an experiment and simulate a model. The results can be plotted and visualized with 3D animation.

Edit window

An edit window is used to compose models from existing classes (models and connectors), connections between connectors and graphical information. An edit window by default shows a single layer of a class; each layer represents a different aspect of the class, see “Class layers” on page 111.

The edit window has a toolbar with buttons to select layer and to start drawing operations. There are browsers for packages and the component structure along the left side. The user may insert components into a model by dragging components from the package browser.

Editor showing the icon layer.



Package and component browsers

The Dymola window contains 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 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.

The diagram window and the component browser are synchronized to give a consistent view. When you select a component in the diagram window, it is also highlighted in the component browser and vice versa. The diagram window gives the top-level component structure of a component, while the component browser gives 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.

The contents of the package browser and the component browser is by default sorted according to order of declaration in the enclosing class. Clicking on the header line sorts the contents in alphabetical order. Clicking a second time restores declaration order.

Modified classes are marked in red in the package browser until they are saved. A package (and all classes inside it) is marked read-only if the Modelica file is read-only.

Package browser context menu

Pressing the right mouse button in the package browser presents a menu with the following choices. A similar menu is presented when pressing the right mouse button in the component browser.

Open Library Window Creates a new library window, see below.

Open Class in This Window Opens the class of the selected class in this window.

Open Class in New Window Opens a new main window with the selected class.

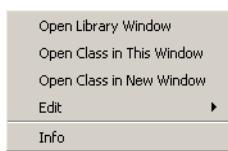
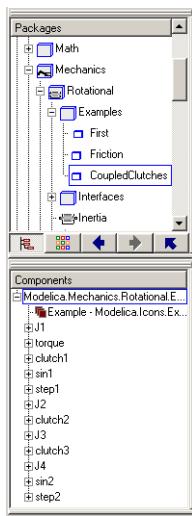
Check Checks the selected class for errors. See “Edit/Check” on page 140.

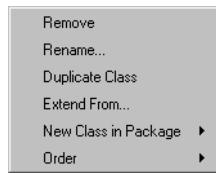
Search... Searches the selected class for all classes or components matching a search pattern. See “File/Search...” on page 134.

Close Closes an open hierarchy and all lower levels.

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.

You can also use “What’s This” on each class to display its documentation in a small popup-window inside Dymola.





Several editing operations are also available, in particular to create/remove/modify classes of a package.

Remove Removes 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”.

Duplicate Class Duplicates a class, see “File/New.../Duplicate Class” on page 133.

Extend From Creates a new class, which extends from the selected class. See also “File/New.../Model etc.” on page 131.

New Class in Package Creates a new class of the desired type in the selected package. See also “File/New.../Model etc.” on page 131.

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.

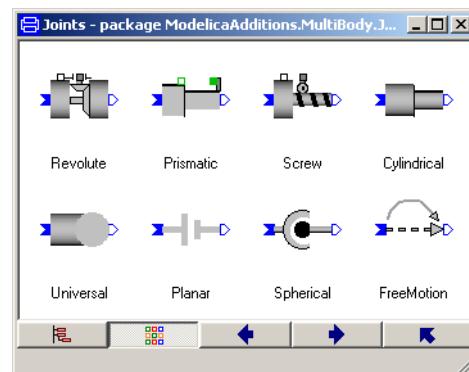
Note that dangling references to the removed or renamed class may not be detected. Make “Check” on the enclosing package to detect some potential problems, see “Edit/Check” on page 140.

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. Library windows do not allow editing and there is no toolbar.

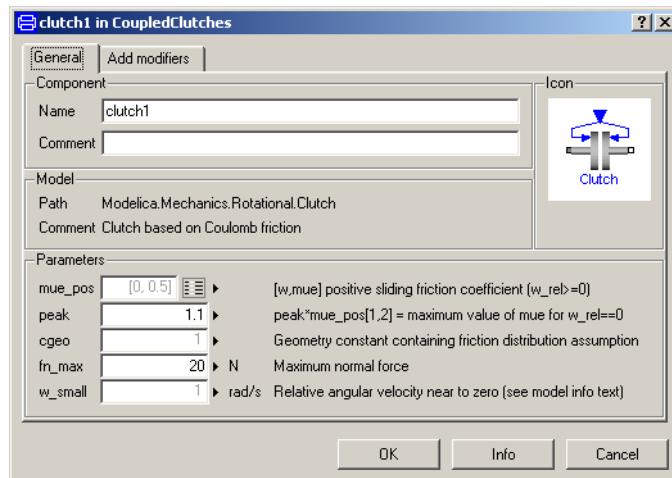
A library window with icon view only.



Modifier window

Modifiers of a component (and possibly its sub-components) are viewed and modified using a modifier window. It also shows the name, icon, comments and full path of the corresponding model. See also “Modifier window” on page 116.

A modifier window.



Class layers

Edit windows use four layers to represent different aspects of a class. The first two layers are graphical:

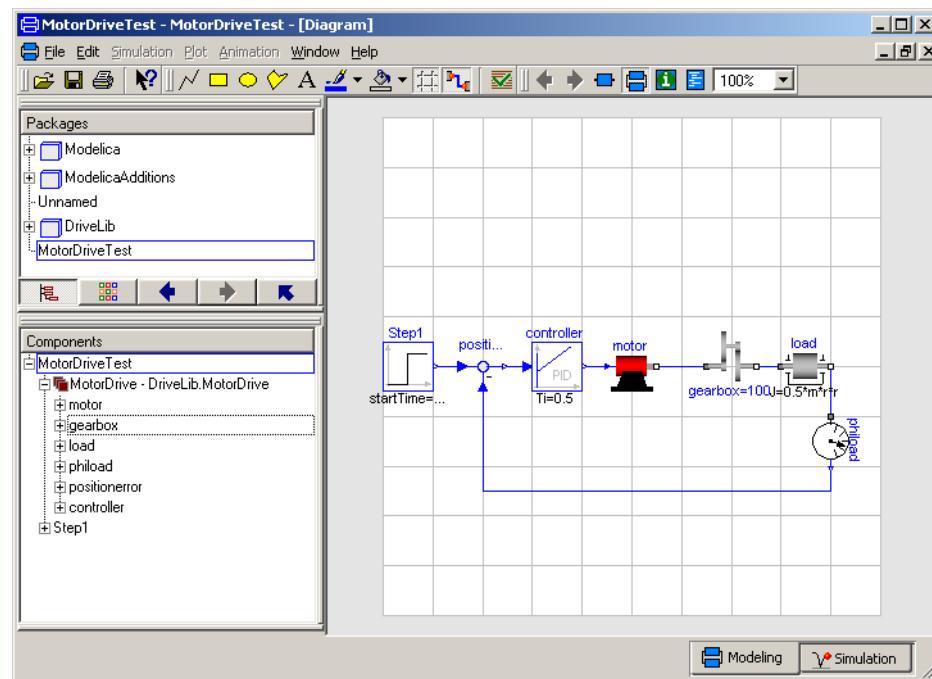
- The icon layer represents the class when it is used as a component or connector in another model, that is, the diagram layer displays the icon layer of its components. The surrounding background of the icon layer is light blue when editing is possible.
- The diagram layer shows the major contents of 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 other two layers use a textual representation:

- The documentation layer shows the one-line description of the class, plus the longer info text that gives a complete explanation of its behavior. These texts are available when browsing classes and components. It is also used for automatically generated HTML documentation.
- 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.

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.

Diagram layer of an edit window.



Modelica text of a model.

```

SICunits.Angle q(final fixed=startValueFixed);
SICunits.AngularVelocity qd(final fixed=startValueFixed);
SICunits.AngularAcceleration qdd;
SICunits.Angle qq;
Real mn[3];
Real sinq;
Real cosq;

equation
  axis.phi = q;
  bearing.phi = 0;

// define states
qd = der(q);
qdd = der(qd);

/*rotation matrix*/
mn = n/sqrt(n*n);
qq = q - q0*PI/180;
sinq = sin(qq);

```

Class documentation

Two levels of documentation is available in the documentation layer or in the component or connector modifier windows, if the author of the model has provided it. If no documentation is available in the model, Dymola will search the base classes for documentation.

Description and information

The first level is a short one-line description of the class, which is shown in the documentation layer and in the modifier window.

The second level of extended documentation is available in the documentation layer, or by using the right mouse button menu. Other comments are associated with declarations or equations. Units and declaration comments are also shown in the modifier window.

HTML documentation

Dymola can automatically produce HTML code for Modelica models. The HTML file may contain all classes used as base classes or as components; this ensures a complete description of the model or the package. All references to classes are clickable links, 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.

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 surrounding parts of the window are empty if the aspect ratios of master coordinate system and window do not match. The user can change the zoom factor to show more or less of the model. 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 in the Graphics tab of Edit/Attributes. 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.

Model editing

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.

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.
- Clicking on an unselected object while holding down the Shift key toggles the select status of this object, without unselecting the previous selection.
- 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 “Modifier window” on page 116.

See also “Edit/Select All” on page 139.

Context menus

Right mouse button presents a context menu.

Pressing the right mouse button usually presents a menu with operations suitable for the selected object. Context menus are presented for components and connectors, for lines and connections, 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.

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 de-

sired 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, 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). 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.

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 Edit menu, for example, rotation and flipping horizontally and vertically.

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.

Components and connectors

Inserting a component or a connector

Components and connectors are inserted by dragging a class from the package browser, or a library window, into the editor. 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). Name, comment and modifiers can be changed in the modifier window, see “Modifier window” on page 116.

Size can be adjusted by moving the handles. Orientation is changed with commands in the Edit menu.

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`.

Context menu for components



Pressing the right mouse button when a component is selected presents a menu with the following choices. A similar menu is presented when pressing the right mouse button in the component browser.

Parameters... Opens the modifier window, see below.

Show Component Shows the class of the component or connector in this window. Press the Previous button to go back.

Open Class in New Window Opens a new main window with the class of the component or connector.

Attributes... Opens a Component Attribute dialog, see “Component attributes” on page 120.

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.

Modifier window

Double-clicking on a component or connector in the graphical editor displays a dialog window with additional information. The modifier window can also be started from “Parameters...” in the context menu of components and extends-clauses in the component browser. This makes it possible to directly change inherited parameters.

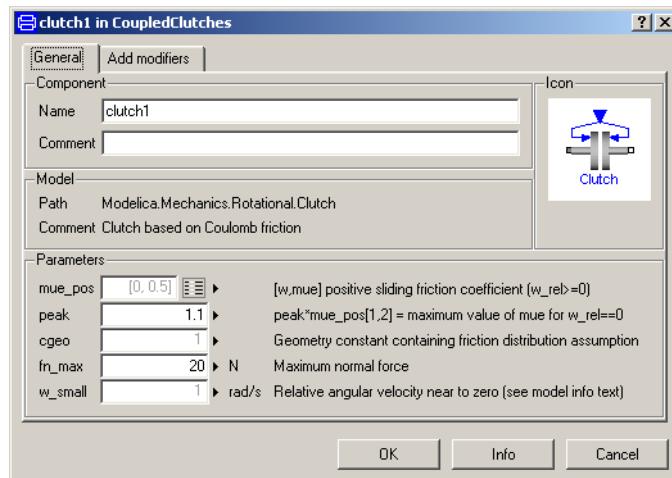
In the “General” tab, the first section shows the name of the component or connector and the component-specific comment. The second section shows the full path of the corresponding class and the class-specific comment. The icon is shown on the right.

The rest of the modifier window contains the modifiers of the component or connector. 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.

The parameters of the modifier window 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.

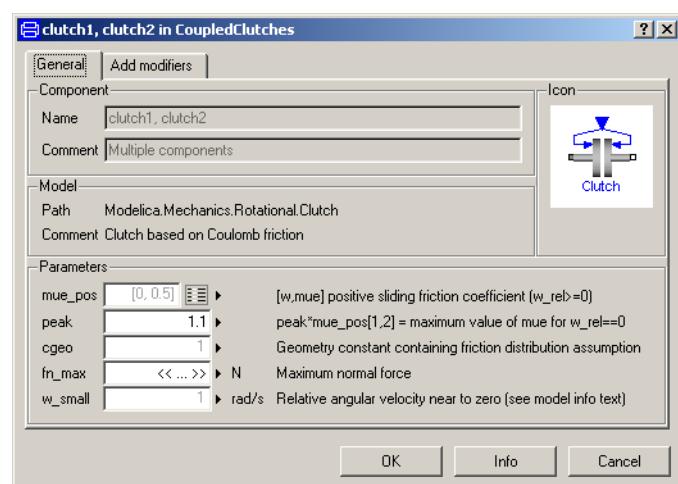
Parameters can be organized in Tabs and Groups.

A modifier window.



When multiple components have been selected, the modifier window 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 modifier window for multiple selection.



Component and connector 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.

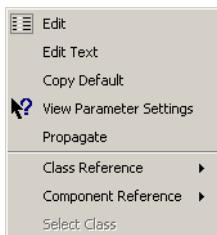
The actual value of a variable is changed by clicking in the corresponding field and typing in the new value. An existing actual value is erased by clicking in the corresponding field and pressing the Delete key. The Tab/Shift-Tab keys move from one input field to another.

The Add Modifiers tab is used to add new modifiers, e.g., start values or nested modifiers.

Pressing the OK button will generate a modifier list for the component. Only modifiers with non-empty value fields are used.

The modifiers that appear and can be set are:

- Normal parameters, i.e. variables declared with the keyword parameter in the model. These can be set to a value that is either numeric or computed from other parameters.
- Class parameters, i.e. replaceable classes in the model, that can be redeclared to another class here. 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 declaration inside choices refer to global scope, i.e. must use full lexical path.
- Hierarchical and other forms of free form 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.



Grey backgrounds for modifiers 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 modifier field, or using “View Parameter Settings” in its context menu.

The context menu of a modifier can be accessed by pressing the triangle to the right of the input field (or by using the right mouse button inside the field). The context menu allows you to:

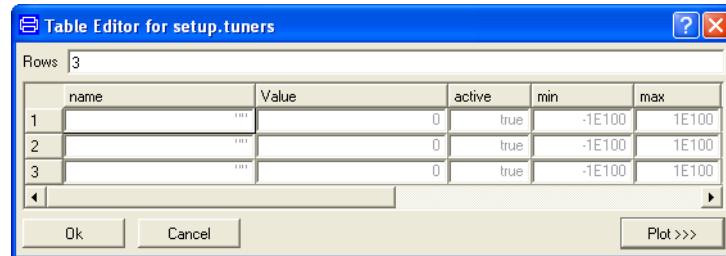
Edit Allows hierarchical inspection and editing. This uses a matrix editor for matrices (see page 119), a function call editor for function calls, and a parameter dialog for structured modifiers and redeclarations.

Edit Text Shows a larger input field for modifying long modifiers.

Copy Default Copies the text of the default value to the input field for further editing.

Edit Combined Edits an array of calls to functions or record constructors.

Editor for array of records.



View Parameter Settings 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 $p=p$. Default values are not propagated to the new parameter declaration.

Class Reference Inserts a reference to a class in the model.

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 redeclarations with all matching classes.

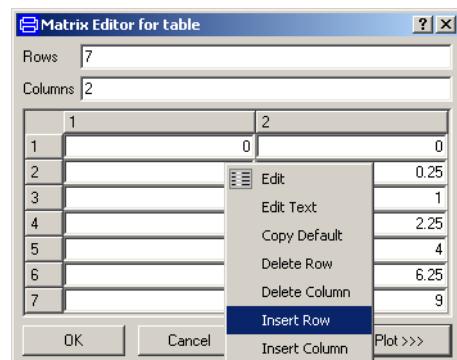
Select Record Selects a record constructor or constant among suitable candidates.

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.

Array and matrix editor

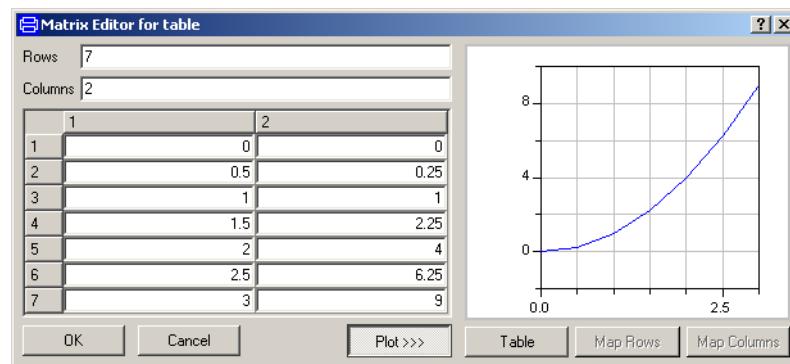
The array and matrix editor allows editing of both elements and array size. A context menu available to insert rows (before the selected entry) and delete rows and columns. Insertion after the last entry is performed by increasing the size.

Editor for matrices.



The matrix editor has an extended dialog for plotting one- and two-dimensional table functions, which is displayed by pressing the Plot >>> button.

Matrix editor with 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.

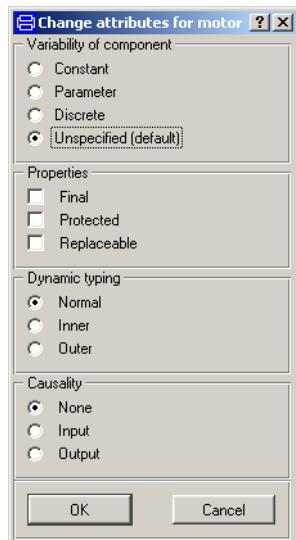
Map Columns Plots columns of a 2D map. Plots columns 2..n versus the first column, except first element of each column.

At present it's not possible to directly adjust the size of the plot. However, it is possible to adjust the size of the matrix editor when the plot is not shown. When the `Plot >>` button is pressed the plot size will be squared with the same height as the matrix editor.

Component attributes

Several component attributes can be changed with `Attributes...` from the context menu. These attributes reflect component properties specified in the Modelica language. The illustration below shows the default settings.

The component attributes dialog.



Connections



A connection is a graphical representation of a `connect` statement between two connectors. Connections can be made when Dymola is in connect mode, as indicated by the pressed-down tool button. If the model is read-only, or connect mode is off, connections cannot be made.

Creating a connection

Connections are defined interactively in Dymola in a manner similar to drawing lines.

- Click on a connector and draw a line, possibly with multiple line segments (see “Lines and polygons” on page 123). 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, connections are manhattanized immediately when created, moved or reshaped. If automatic `manhattanize` is on, moving a component automatically manhattanizes all connections to the component.

Connections are immediately checked.

When a connection has been defined some checks are performed automatically.

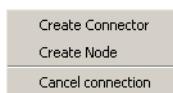
- A connection that begins and ends at the same connector is not allowed.

- 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.

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.

Note that drawing a 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 a 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.

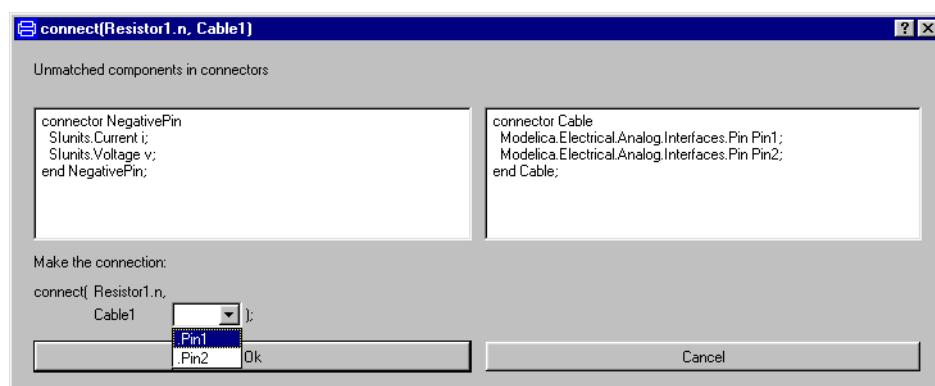
Create Node Creates a new protected connector (internal node) at the cursor position and the connection is completed. This connector has the same type as the connector at the start of the connection. The size of the new connector is 1/10 of the normal component size.

Cancel connection Cancels the connection operation; no connection is made.

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 subconnector is presented.

Nested connector dialog

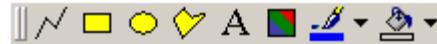


In this case a connection from the Pin Resistor1.n to the connector Cable1 is made. Cable has two nested connectors Pin1 and Pin2. The pull-down choices shows these alternatives.

Creating graphical objects

Graphical objects are drawn by clicking on buttons in the draw toolbar. The corresponding toolbar button is shown in a depressed state while the drawing operation is in progress. The toolbar also contain two buttons for setting line and fill style attributes, with associated pull-down menus.

The drawing toolbar.



The graphical tools described in this section 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.

Lines and polygons

Click on the polyline 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.

Polygons are automatically closed by Dymola.

Skewed lines can be cleaned up with Edit/Manhattanize which applies a manhattan-mode algorithm to the intermediate points of the line

Note that drawing a line between two connectors does *not* create a connection, although the difference may be impossible to see in the editor.

Lines are not connections.

Rectangles and ellipses

Click on the rectangle or ellipse button in the toolbar, and draw the shape in an edit window. Ellipses are drawn to touch the bounding box at four points.

Text

Click on the text button in the toolbar, and draw the text object in an edit window. Dymola prompts for a new text string in a separate dialog window.

The text is centered in the bounding box; the size of the text is chosen as big as possible without overflowing the bounding box. If the minimum font size option is set, 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.

These tokens are expanded in a text string:

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

Double-clicking on a text displays a dialog window for changing its contents.

Bitmap

Draws a bitmap which is read from an external file. Click on the bitmap button in the toolbar, and draw the outline in an edit window, 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.

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.

Changing graphical attributes

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.

Line style



The line style attributes are

- Color, (or no color, for example to draw a filled area without border).
- Line style (solid, dashed, dotted, dash-dot or dash-dot-dot).
- Thickness (single, double or quad).
- Arrow style (none, start, end, both or harpoon).

Fill style



The fill style attributes are

- Color, (or no color, for example to draw a rectangle).
- Fill pattern (solid, left diagonal, right diagonal or cross pattern).
- Gradient (flat, horizontal, vertical or spherical).

Modelica text

Modelica text such as declarations, equations and algorithms are edited in the Modelica text layer. By default it contains the Modelica text for declarations of non-graphical components of the class, e.g., constants, parameters and variables, and equations of the model.

Modelica text layer.

```
Revolute - ModelicaAdditions.MultiBody.Joints.Revolute (Read-Only) - [Modelica Text]
File Edit Simulation Plot Animation Window Help
100%
SIunits.Angle q(final fixed=startValueFixed);
SIunits.AngularVelocity qd(final fixed=startValueFixed);
SIunits.AngularAcceleration qdd;
SIunits.Angle qq;
Real mn[3];
Real sinq;
Real cosq;

equation
axis.phi = q;
bearing.phi = 0;

// define states
qd = der(q);
qdd = der(qd);

/*rotation matrix*/
mn = n/sqrt(n*n);
qq = q - q0*PI/180;
sinq = sin(qq);
```



Graphical components and connections are by default indicated by icons inserted into the text. Using the context menu it is possible to show the corresponding Modelica text.

Context menu

Undo	Ctrl+Z
Redo	Ctrl+Y
Cut	Ctrl+X
Copy	Ctrl+C
Paste	Ctrl+V
Select All	Ctrl+A
Find	Ctrl+F
Goto line	Ctrl+G
Expand	▶
Highlight Syntax	
Settings	▶
Insert	▶
Edit function call	
Selected Class	

Pressing the right mouse button presents a context menu with common text operations. Most of these operations are also available in the Edit menu.

Undo, Redo Undoes and redoes the last editing operation in the text editor. Same as Edit/ Undo and Edit/Redo.

Cut, Copy, Paste Copies text between the clipboard and the editor.

Delete Deletes all text in the editor.

Select All Selects all text in the Modelica text layer.

Find Find and replace text in the text editor. Enter the text you search for and press OK.

Goto Line Sets cursor at specified line and scrolls window if necessary.

Expand Selects the expansion level. Normally graphical components and connections are represented with icons in the corresponding Modelica text. By editing the expand-level you can see these, and also the contents of the annotations.

Highlight Syntax Scans the text in the editor and applies syntax highlighting for the Modelica language. Any syntax errors are reported in the message window.

Settings Defines settings that affect the editor.

Insert Presents a submenu with common Modelica constructs. This makes it easier to enter Modelica code.

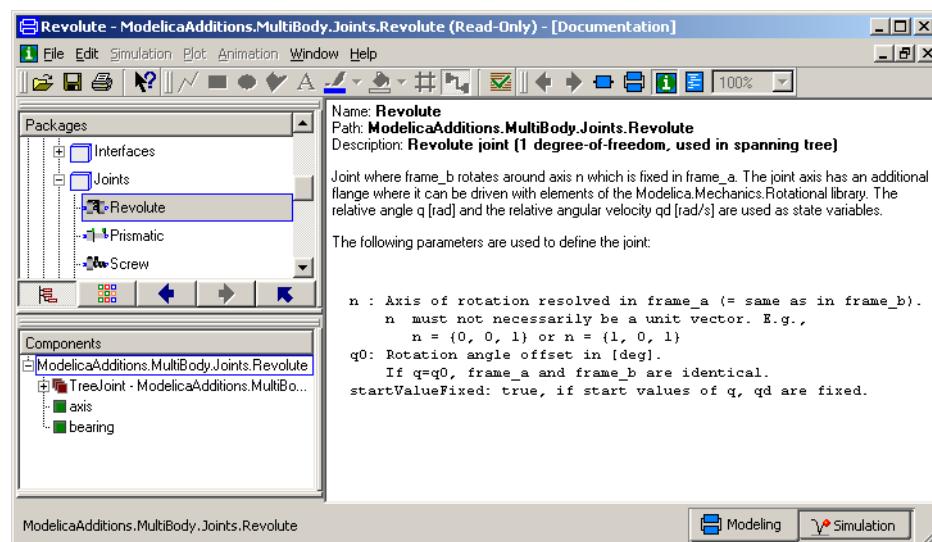
Edit function call Inside a function call this allows you to modify the arguments using a modifier window.

Selected Class Allows you to open the selected class, or its documentation.

Documentation

Class documentation consists of a one-line description and a larger documentation text. This information, as well as full model path, is shown in the documentation layer.

Documentation layer of a model (viewing).



Context menu when viewing

Copy	Ctrl+C
Select All	Ctrl+A
Find	Ctrl+F
Goto line	Ctrl+G
Edit Source	

Pressing the right mouse button presents a context menu with common text operations. Most of these operations are also available in the Edit menu.

Copy Copies text from the documentation layer to the clipboard.

Select All Selects all text in the documentation layer.

Find Find and replace text in the text editor. Enter the text you search for and press OK.

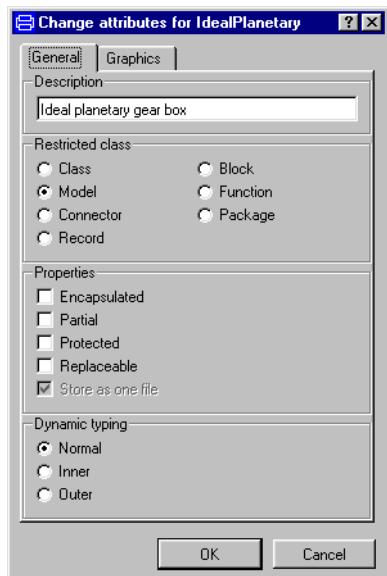
Goto Line Sets cursor at specified line and scrolls window if necessary.

Edit Source Switches the documentation layer from viewing mode to editing mode, see below.

Editing description

The description string is edited by selecting Edit/Attributes from the menubar. It is also possible to modify other attributes of the class, and to select if the class is stored in one file or hierarchically, see “Edit/Attributes...” on page 140.

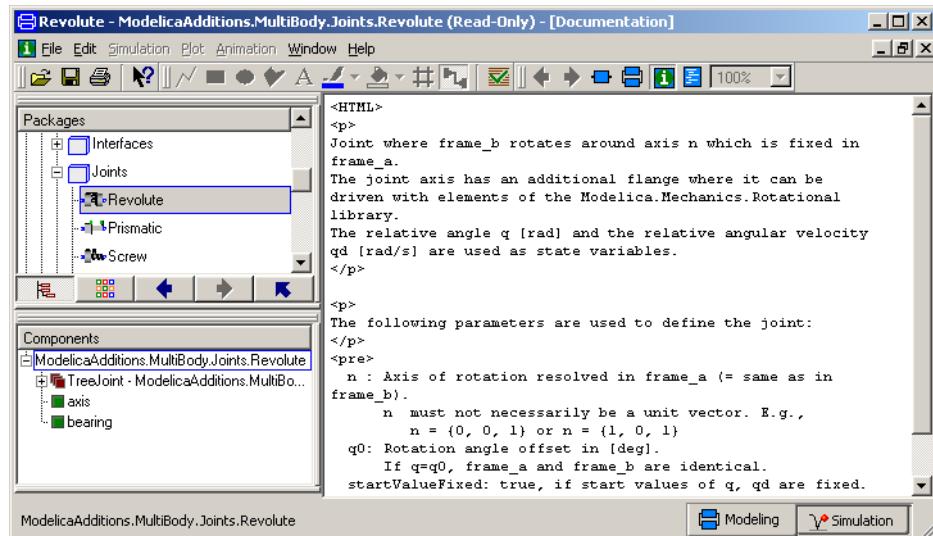
Model attributes.



Editing documentation text

Choosing Edit Source from the context menu switches the documentation layer to editing mode. In this mode the unformatted HTML code of the documentation is shown, and can be edited.

Documentation layer of a model (editing).



Arbitrary HTML can be used for documentation.

The user may include arbitrary HTML code in the documentation by enclosing it in an `<HTML> </HTML>` escape sequence in the textual documentation. This makes it possible

to take advantage of images, tables and other forms of formatting. For example, a plot image can be referenced with:

```
<HTML>
The result is shown here: <IMG SRC="../images/plot.gif">
</HTML>
```

In addition the browser also supports universal resource identifiers for 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.

Context menu when editing

Undo	Ctrl+Z
Redo	Ctrl+Y
Cut	Ctrl+X
Copy	Ctrl+C
Paste	Ctrl+V
Clear	
Select All	Ctrl+A
Find	Ctrl+F
Goto line	Ctrl+G
▼	Edit Source

Pressing the right mouse button presents a context menu with common text operations. Most of these operations are also available in the Edit menu.

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 all text in the editor.

Select All Selects all text in the documentation layer.

Find Find and replace text in the text editor. Enter the text you search for and press OK.

Goto Line Sets cursor at specified line and scrolls window if necessary.

Edit Source Switches the documentation layer back to viewing mode.

HTML documentation

Dymola can 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.

Pressing the Info... button will use HTML documentation if it is available, both for the Dymola standard packages and for classes written by users. This requires the conventions of the default online documentation.

External references

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. What kind of information is emitted for each class is controlled by another set of options.

This is the suggested default mode.

Generate online documentation

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 subpackage. 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

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

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.

Do not generate external links

HTML without any links to other files is generated. This setting provides pretty-printed documentation for a single model or package only.

HTML options

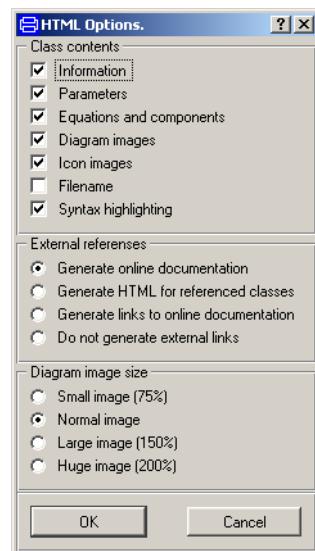
The generated HTML code is controlled with regards to documentation contents, references to other files and the size of graphics.

Class contents 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 parameter section is a tabular description of all parameters, including parameters inherited from base classes.

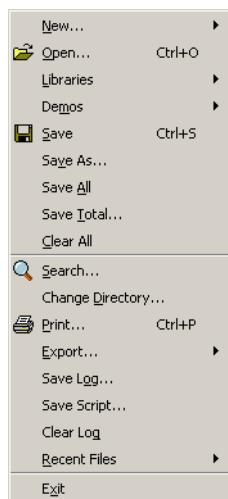
External references Controls what kind of HTML documentation is generated, see above.

Diagram image size 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.

The HTML options dialog, default settings shown.



Editor command reference



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 143.

File/New.../Model etc.

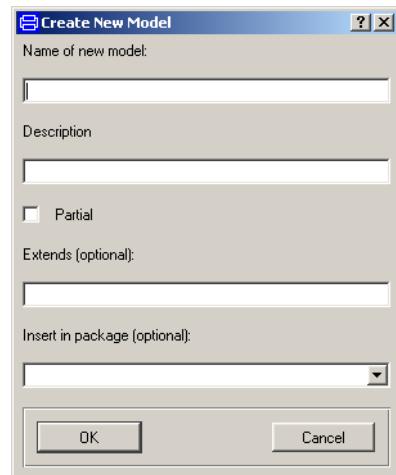
Creates a new model, connector, record, block or function. Dymola presents a dialog window where the user must enter the name of the new model, a short description, and if this is a partial model. The name must be a legal Modelica identifier and not clash with existing models.

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.

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.

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

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/Attributes...” on page 140, or for a top-level package using “Save As” and not changing the name, see “File/Save As” on page 134.

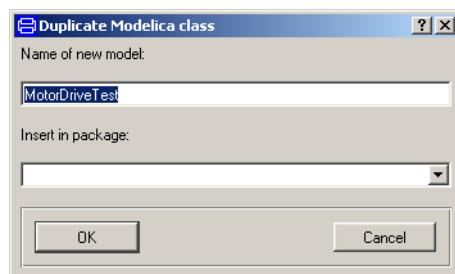
Creating a new package.



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.



File/Open...

Reads the contents of a Modelica file. The names 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.

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

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 statusbar with File/Open.

The library menu is built with library commands in `dymola\insert\dymodraw.ini`.

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 “Simulation/Run Script...” on page 149.

File/Save

Saves the class in the current window.

- If the original model definition was read from a file, that file is updated.
- Otherwise, a 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 a 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 133. 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.

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.

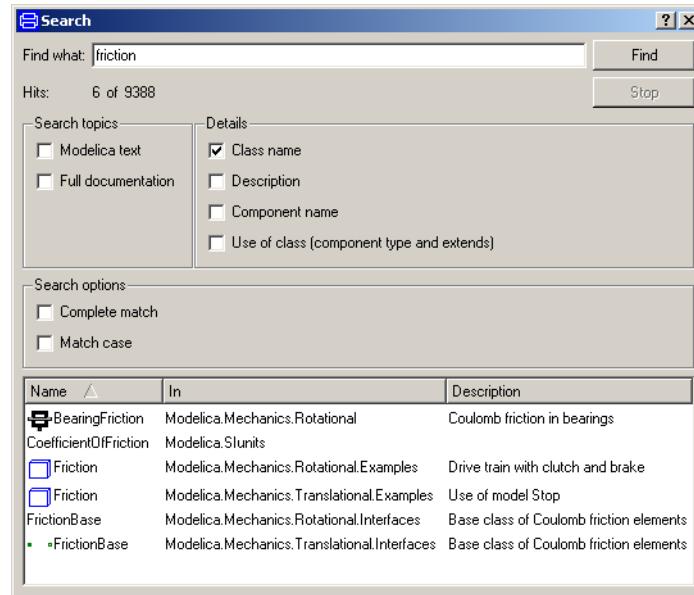
File/Clear All

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

File/Search...

Shows all classes or components matching a search pattern. File/Search... will search all classes read by Dymola. Selecting Search... in the package browser will only search the selected package.

The search dialog.



The search pattern can be matched against two general topics of the classes:

Modelica text Match search pattern anywhere in Modelica text, except for documentation and other annotations. The full Modelica text of a package is not searched, only the classes inside the package. 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 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. Examples:

```

model M
  extends pattern;
  pattern comp_name;
  CompType c1(redclare 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.

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.

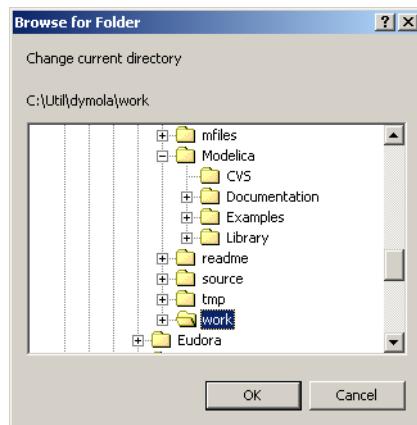
File/Change Directory...

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.

cd

Change Directory may also be given as a textual command. A cd command without directory specification prints the current working directory in the log window.

Changing directory.



File/Print...

Prints the contents of the window on the current printer. This command is only available in the Windows version of Dymola.

File/Export/To Clipboard



Copies the contents of the active window to the clipboard in Enhanced Metafile Format, which can then be pasted in common word processors. Such an image can be rescaled and edited in e.g. Microsoft Word. This command is only available in the Windows version of Dymola.

File/Export/Image...

Saves a PNG image of the contents of the active window (without window borders). The image is identical to the image shown in the window, so the size and representation can be changed by first resizing the window.

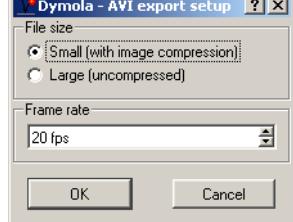
Exported images are included in the command log.

File/Export/Animation

Saves the current animation window in AVI (Audio-Video Interleaved), animated GIF or VRML format.

To export an animation in AVI format, the user must select small 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.

AVI export setup.



To create animated GIF images, Dymola requires that GIF Construction Set™ from Alchemy Mindworks has been installed on the computer. Environment variable DYMOLAGIFCON must point to the executable program, and Dymola must be started afterwards.

File/Export/Setup HTML...

Sets up options for exporting HTML. See “HTML documentation” on page 129.

File/Export/HTML...

Exports comprehensive model and library documentation in standardized HTML format according to the options in File/Export/Setup/HTML. See “HTML documentation” on page 129.

File/Save Log...

Saves the contents of the command log window to file. The user is prompted for a filename. The contents of the command log can be saved in three different formats.

- HTML log with embedded images created with File/Export/Image. This format is the closest to the command log as shown in Dymola.
- Textual log, without any images but including command output.
- As script file, containing all commands given but without the output from the commands.

File/Clear Log

Erases the contents of the command log.

File/Recent Files

Shows a list of the most recent files which were opened with File/Open... (see page 133) or Simulation/Run Script... (see page 149). Selecting one of the files opens it again, typically to open models after File/Clear All or to re-run a script.

File/Exit

Terminates the program. Before terminating, Dymola will ask if modified model classes should be saved, one by one.

`exit`

The exit command can also be given as a textual command, for example in a script.

Edit menu

Toolbar



The toolbar contains buttons for the drawing operations, to toggle gridlines, and to toggle connect mode. The final button checks the model for errors.



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 115 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/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/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/Check

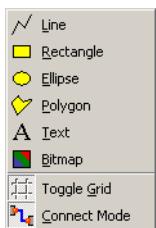


Checks the class in the current window for errors. The class is checked for syntactic errors, references to undefined classes and the consistency of equations. No simulation code is generated.

Edit/Draw

These are the commands to interactively draw new shapes, and related setup. See also “Creating graphical objects” on page 123.

- Drawing graphical shapes (line, rectangle, ellipse, polygon, text and bitmap).
- Show gridlines to make alignment of objects easier.
- Toggle connect mode, see “Connections” on page 121.

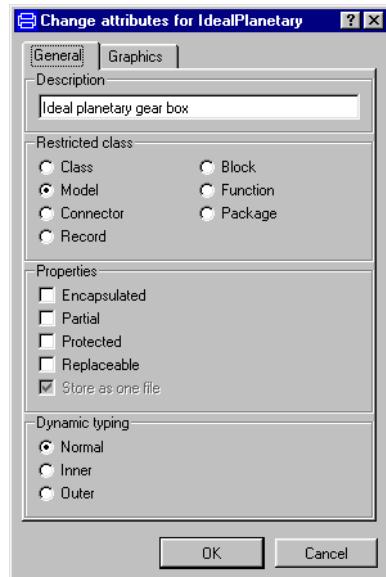


Edit/Attributes...

The General tab contains 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.
- Dynamic typing (`inner` and `outer`).

General model attributes.

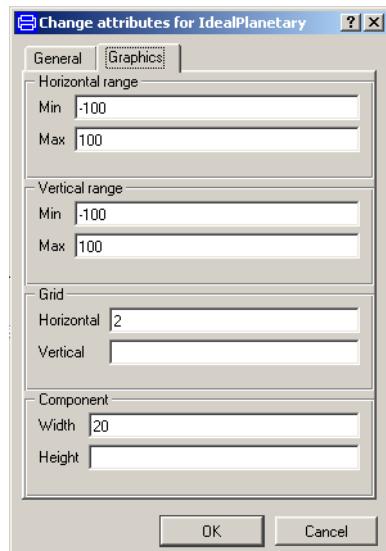


The Graphics tab contains options that affect the graphics of the class.

- Change the coordinate system of the current class.
- Change the grid and the default component size of the current class.

See also “Coordinate system” on page 113.

Graphical model attributes.

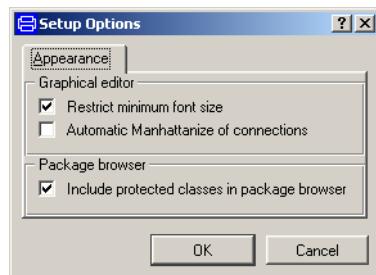


Edit/Options

The Appearance tab contains two options that affect the operation of the graphical editor.

- Restrict minimum font size, to make small texts more easily readable. As a consequence, these texts may overflow their bounding boxes.
- Automatic Manhattanize of connections. If enabled, connections are manhattanized immediately when created, moved or reshaped.
- Include protected classes in package browser. If true, protected classes are shown in the package browser. If false protected classes are not shown.

Editor options.



Window menu

Toolbar



The toolbar contains buttons showing the previous and next model viewed in the window, and for viewing different layers of the model, see also “Class layers” on page 111. The last control sets the zoom factor.

Recent Models



Clicking on the 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 model to allow arbitrary selection.

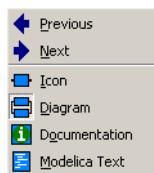
Window/Mode/Modeling (Ctrl+F1)

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.

Window/View/Modelica Text

Displays the declarations and equations of the class.

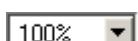
Window/New Library Window

Creates a new library window, see “Library window” on page 110.

Window/New Dymola Window

Creates a new modeling and simulation window.

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 makes it possible to move around in the model.

Help menu

Help/What's This



Displays interactive help for the graphical user interface. After selecting help/What's This the cursor changes. Then click on the item in the user interface you want help on.

Help/Contents

Opens the online manual for Dymola.

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 support

Opens a mail window for sending a message to the support staff at Dynasim.

Help/Dynasim website

Opens a web browser window with the Dynasim webpage.

Help/About

Displays copyright and license information about Dymola.

Special keyboard commands

Delete

Deletes the current selection.

Shift

Enables multiple selection of objects. Pressing Shift at the end of the connection operation negates the meaning of the automatic manhattanize option.

Ctrl+F1 and Ctrl+F2

Changes window mode, see “Window/Mode/Modeling (Ctrl+F1)” on page 142.

Arrow keys

In the diagram and icon layers, selected graphical objects can be moved by pressing the arrow keys. See also “Moving objects” on page 114.

Model editor initialization

The initialization script file, called `dymodraw.ini`, contains commands that are executed at program startup to configure the model editor for a particular user. A default file is provided with Dymola, and the user may create a customized file based on the default file, for example, to add more library windows.

Dymola uses the environment variable `DYMOLAPATH` to locate `dymodraw.ini`. The contents is a list of directories separated by space or ;. Dymola will search for these files in the current directory, then in each of the directories specified in `DYMOLAPATH`, and finally in the default Dymola directory.

Filenames may begin with a reference to an environment variable in the following form:

```
$DYMOLA/insert/dymodraw.ini
```

The current value of environment variable `DYMOLA` is substituted for `$DYMOLA`.

In addition to the commands described below, comments can be inserted. Any text from the # character until the end of the line is regarded as a comment.

SIMULATING A MODEL

Simulating a model

This chapter describes how to simulate a Modelica model in Dymola. First, the basic steps in setting up a simulation, running it and plotting and animating the results are described. See also “Getting started with Dymola” on page 21, which includes several examples.

The section “Simulation menu” on page 149 describes the commands and options for setting up and running a simulation. Plotting of results are described in the section “Plot window” on page 156 and animation in the section “Animation window” on page 166. The section “Scripting language” on page 173 describes the scripting feature. How to find errors and improve simulation efficiency are discussed in the section “Debugging models” on page 182. The two sections “Inline integration” on page 190 and “Mode handling” on page 192 describes to approaches to increase simulation speed for real-time simulation.

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 section “Scripting language” on page 173.

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.

Setting parameters and initial conditions

After translation new values for parameters and initial values can be entered using the variable selector, which displays a tree of all variables.

**The variable selector
for setting parameters
and initial values.**

Variables	Values	Unit	Description
J1			
phi	0	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))
der(phi, 2)		rad/s^2	der(Absolute rotation angle of component (= flange_a.phi = flange_b.phi), 2)
flange_a			
flange_b			
J	1	kg.m2	Moment of inertia
w	10	rad/s	Absolute angular velocity of component
der(w)		rad/s/s	der(Absolute angular velocity of component)
a		rad/s^2	Absolute angular acceleration of component
torque			
clutch1			
sin1			
step1			
J2			
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))
der(phi, 2)		rad/s^2	der(Absolute rotation angle of component (= flange_a.phi = flange_b.phi), 2)
flange_a			
flange_b			
J	1	kg.m2	Moment of inertia
w	0	rad/s	Absolute angular velocity of component
der(w)		rad/s/s	der(Absolute angular velocity of component)

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.

Specify simulation run



To set up the experiment, click on the Setup Experiment button to get a menu for setting simulation time, output interval and specifying integration method etc.

Perform simulation

To run the simulation, click on the Simulate button.

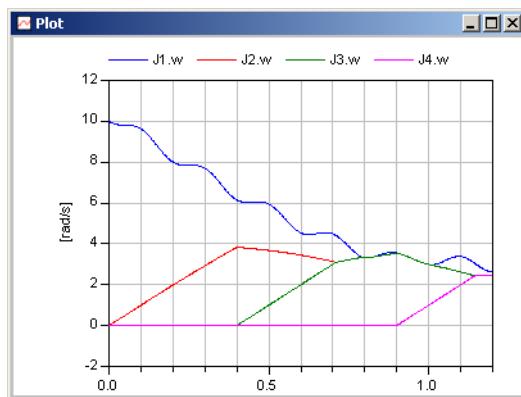


Plot results

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.

Variables to be plotted are selected by clicking in the variable selector. When a variable is selected the square in front of the name is ticked. The variable selector above has J1.w and J2.w selected for plotting.

A plot window, active in the variable selector.



Simulation menu



The simulation menu includes commands to setup and run simulations.

The most common simulation commands are available in the tool bar:



Simulation/Run Script...

Executes the commands in the script file. The scripting facilities are described in more detail in “Scripting language” on page 173.



Simulation/Translate

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.



Simulation/Simulate

Simulates the model for one simulation period according to the specified experiment 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

Continues the simulation for one simulation period (start time to stop time) from the previous stop time.



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.

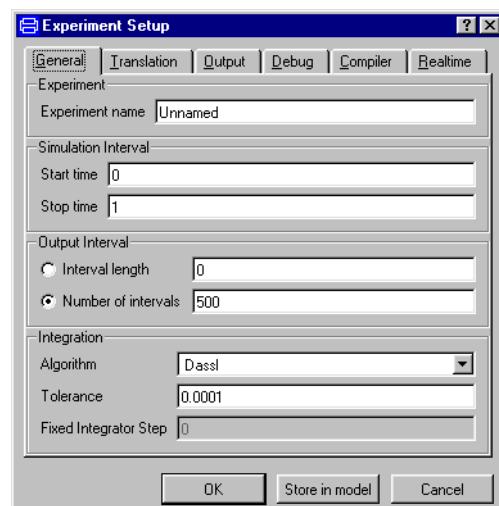
Simulation/Linearize

The command Simulate/Linearize calculates a linearized model at the initial values. The linearized model is stored in Matlab format and can be loaded into Matlab with the m-file tloadlin.

Simulation/Setup

Opens a dialogue for specifying experiment name, simulation interval, integration method etc.

General tab of Simulation/Setup.



General tab includes

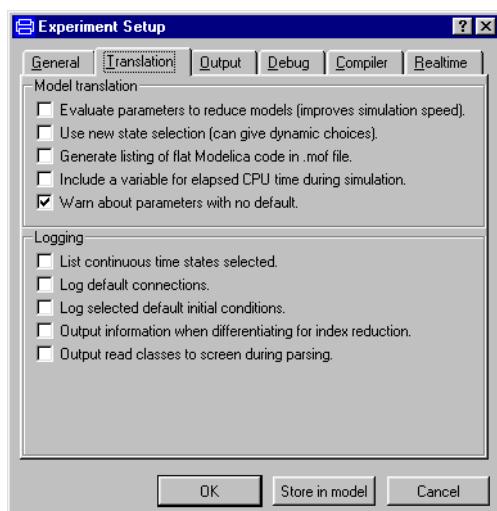
- **Experiment name** to specify the name of the experiment. It is used to name the result file.

- **Simulation Interval** to specify **Start time** and **Stop time** for the simulation.
- **Output Interval** 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.
- **Integration** 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.

The information from this tab can also be stored in the model, by selecting “store in model”.

Model translation tab includes

Model translation tab of Simulation/Setup.



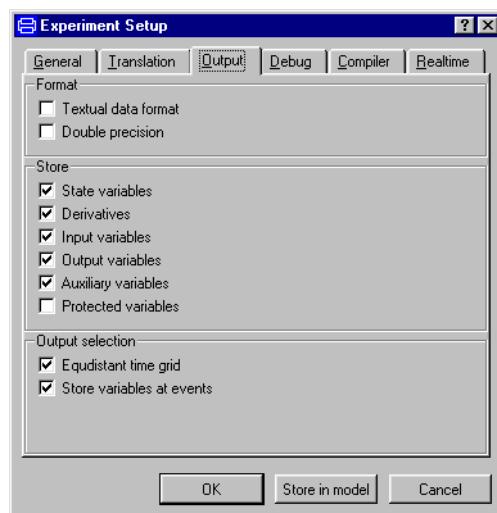
- **Evaluate parameters** to have all parameters except top-level parameters evaluated at translation in order to generate more efficient simulation code. These parameters cannot be set interactively before a simulation run.
- **Generate listing of flat Modelica code in .mof file** to output a listing of all variable declarations and equations.
- **Include a variable for elapsed CPU time during simulation** introduces a new extra variable, CPUtime, with the accumulated CPU time during simulation. The slope of this signal is an indirect measure of the computational complexity of the simulation.
- **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 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 inac-

tive start values and make their start values active by virtually turning their fixed to true to get a structurally well posed initialization problem.

- **Output information on the automatic state selection** reports on the state selection for index reduction.
- **Output information when differentiating for index reduction** reports about which equations that are differentiated.
- **Output read classes to screen during parsing**
- **Warn about parameters with no default** lists all free parameters not given an explicit value in the model.

Output tab includes

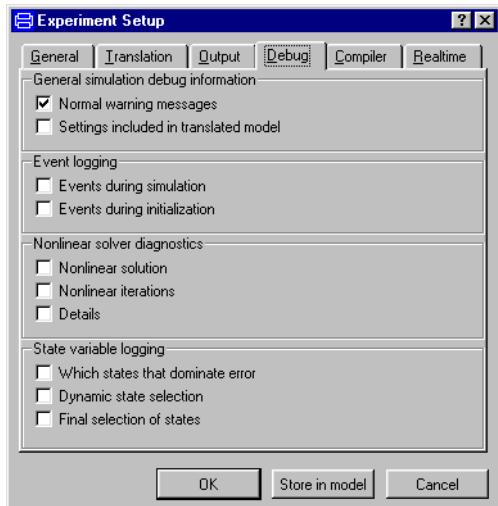
Output tab of Simulation/Setup.



- **Textual data format** stores result in textual format instead of binary.
- **Double precision** stores results in double precision instead of single precision.
- **Store** defines which categories of variables to store.
- **Equidistant grid** stores result equidistantly as given by simulation setup.
- **Store variables at events** stores variables before and after events.

The information from this tab can also be stored in the model, by selecting store in model.

Debug tab includes



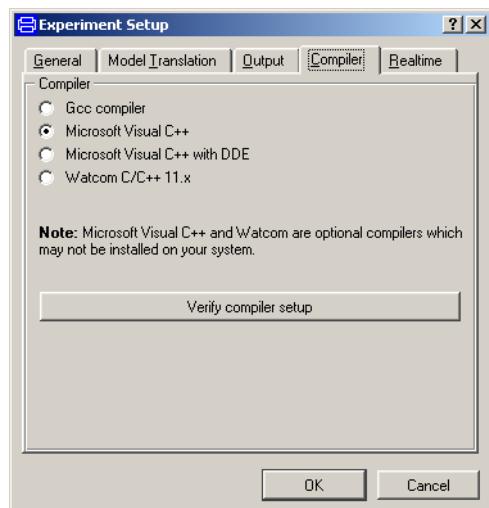
This tab makes it possible to track down problems during simulation, e.g. chattering, problems with non-linear systems of equations, state-selection. How to use these are described in more detail in “Debugging models” on page 182.

Normally these settings do not require a new translation, but does on the other hand only take effect when running inside Dymola.

- **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.
- **Events during simulation** Log events during the simulation. Useful for finding errors in the event logic including chattering.
- **Events during initialization** Also log events during the initial phase.
- **Nonlinear...** For finding problems with non-linear system of equations, e.g. ill-posed problems, or missing noEvent (see “Using noEvent” on page 259).
- **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.

Compiler tab

Compiler tab of Simulation/Setup.



Allows the user to change the compiler used by Dymola to compile simulation code into an executable program that performs the simulation.

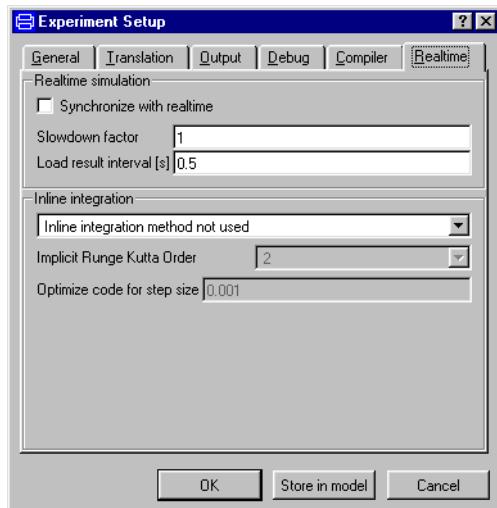
Dymola uses a C compiler to compile the generated C code into a simulation program. The default compiler shipped with Dymola on PC is a distribution of the GNU C Compiler (gcc). Users of Microsoft Visual C++ or WATCOM C/C++ may use these compilers instead. If support for a compiler has not been selected when installing Dymola, the corresponding choice is disabled, see also "Multiple compiler support" on page 282.

On UNIX system the default compiler is used. The compilation is controlled by the shell script `dymola/insert/dsbuild.sh`.

Several different kinds of problems may occasionally arise when compiling the generated C code produced by Dymola. Pressing the "Verify compiler setup" button performs several tests and reports any potential problems. See also "Troubleshooting" on page 290.

Realtime tab

Realtime tab of Simulation/Setup.



Activating items in this tab requires the real-time simulation option of Dymola and that you use MicroSoft Visual C++ with DDE as compiler.

- **Synchronize with realtime** This enables (soft) realtime simulation with frequency of about 20Hz running under Windows . For hardware-in-the-loop simulation see the section on other simulation environments.
- **Slowdown factor** This makes it possible to simulate in scaled realtime.
- **Load result interval [s]** How often the result file should be loaded. Online plots are updated this often.
- **Inline integration** Select inline integration method. The implicit methods are important for real-time integration of stiff systems (i.e. models with time-scales faster than the). For the higher order methods it is possible to select order, and for all implicit methods 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.



Simulation/Visualize

Visualizes initial configuration of 3D models.

Simulation>Show Log

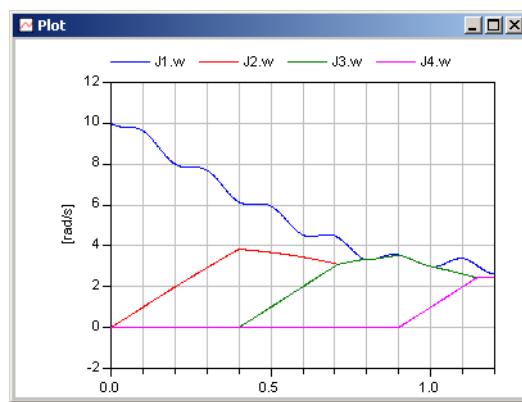
Opens the message window and shows a log for the last simulate command.

Plot window

Dymola supports plotting of any variable. In Simulation mode, multiple plot windows may be created. Each plot window may contain several diagrams. Multiple curves in each diagram are allowed. Multiple diagrams in a plot window allow the user to lay out the curves nicely with aligned time axis and different heights and y-scales of the diagrams.

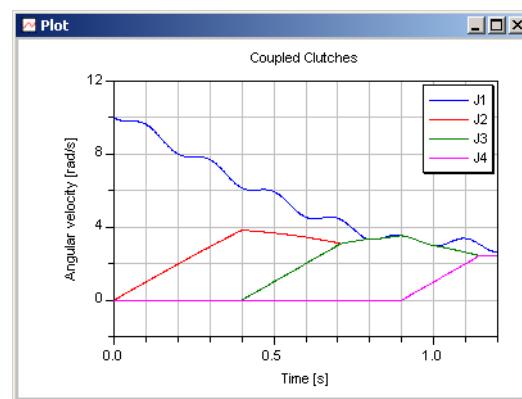
The default layout of plot window displays the names of plotted signals in a legend above the diagram, and the unit of the signals (if available) along the vertical axis.

Default plot window.



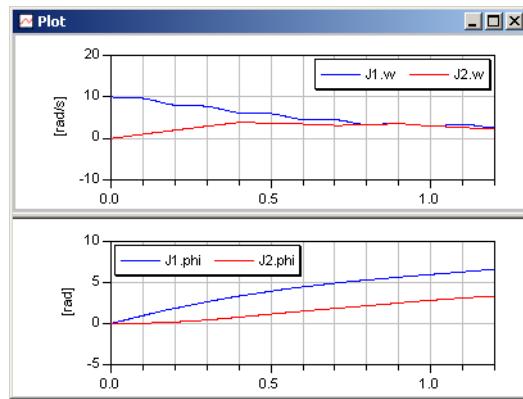
The user can set diagram heading and axis titles. The legend can be located in several different places, for example inside the diagram. See also “Plot/Setup...” on page 161.

User-defined titles and location of legend.



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 the plot commands are directed to this diagram.

A plot window where the upper diagram is active.

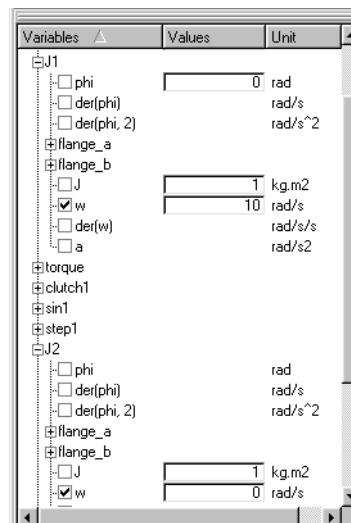


Variable selector

The variable selector displays a tree view of all variables. Each submodel represents one level of the hierarchy, and all variables of a submodel 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.

If more than one variable is selected, they will be plotted in a single diagram with a common scale. Variables that contain data for 3 D animation are not shown.

The variable selector window.



Selecting plot variables

Operations when selecting plot variables are

- **Click on the + symbol in front of a submodel name, or double-click on the submodel name.** If not already open, opens the submodel and displays variables and nested submodels. Otherwise the submodel is closed.
- **Click on a variable.** Plots the variable if it was previously unselected; 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 submodel instance.



The Erase Curves button erases all variables (see “Plot/Erase Curves” on page 161).

J1.w
J2.w
Time

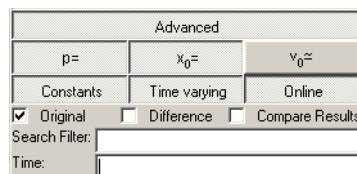
Selecting independent variable

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 selector, put the cursor on the variable and press right mouse button. A menu pops up where the first alternative is the current independent variable and then follows the selected variable. Time is always an alternative to allow going back to plotting trend curves.

Advanced mode

Pressing the “Advanced” button in the plot selector displays buttons for selecting which categories of variables are displayed in the plot selector. It also allows plotting of the difference between signals and to compare the results of multiple simulations.

Advanced plot selector mode.



p= Set parameters of translated model. You must first translate the model.

x₀= Set initial conditions of translated model. You must first translate the model.

v₀= 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 plot-selector. 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.

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 plot selector. 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 occurrences 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 plot selector will show the values of variables for this time. Enter time and press return.

Relative tic mark labels

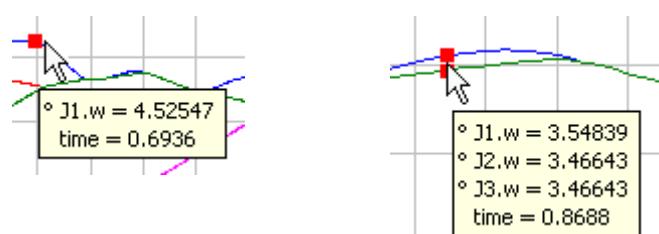
When the range is too small to display absolute tic mark labels, diagrams use relative axes instead of absolute axes. This mode is highlighted by bold+underlined text for the base number and “+” signs in front of the relative offsets.

Plot window interaction

Dynamic tooltips

Resting the cursor over a plotted signal displays a tooltip with information about the closest data point. The data point is also highlighted. 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.



Tooltip for multiple signals and a common independent variable.

Zooming 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.

- **Clicking on the left mouse button.** Zooms the diagram 100% around the clicked point.
- **Pressing the left mouse button and spanning a rectangle.** Zooms in on the drawn rectangle, which is scaled to fit the diagram.
- **Clicking on the right mouse button.** Returns to the previous zoom level. Limits defined with menu command Plot/Range are also used in the context.

File menu

File/Export Image...

Saves PNG image of active plot window, excluding window borders and menubar. The user is prompted for a file name.



Plot menu

The plot menu includes commands to load results, create and edit plots and diagrams.

The most common plot commands are available in the tool bar:



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.



Plot/New Plot Window

Creates a new active plot window which is initially empty.



Plot/New Diagram

Creates a new active diagram in the active plot window.



Plot/Delete Diagram

Deletes the currently active diagram.



Plot/Rescale

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 Curves

Erases all curves in the active diagram. If automatic rescaling is enabled, the diagram is rescaled to the default scale.

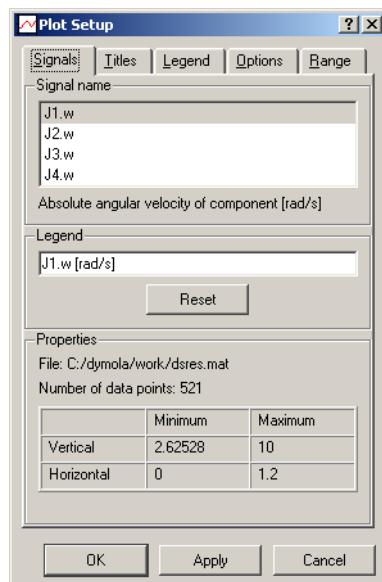


Plot/Toggle Grid

Enables grid lines in the diagram if they are not visible, otherwise disables grid lines.

Plot/Setup...

Signals tab of the plot window setup.

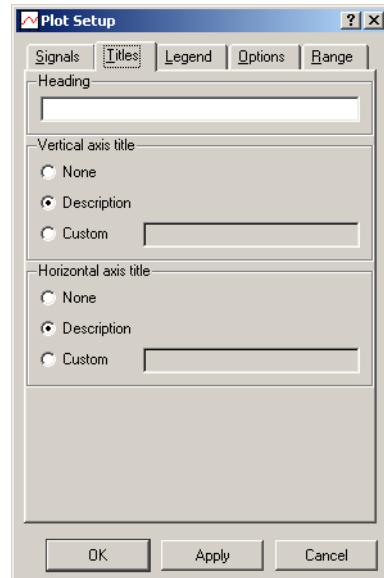


Signals tab Selecting a signal displays the corresponding information in the rest of the window. It is also possible to navigate by pressing the Up and Down arrow keys to select a signal. If a comment for the signal exists it is shown below the listbox.

The **Legend** field contains the current description of the selected signal. Edit the text and press **Apply** to change the legend. If the text is empty, no legend is shown for this signal. Pressing the **Reset** button fills the edit field with the default legend. User-defined legends are shown on page 156.

The **Properties** group shows the name of the result file, the number of data points, as well as minimum and maximum values of the signal. Plotted values can interrogated in the diagram, see “Dynamic tooltips” on page 159.

Titles tab of the plot window setup.



Titles tab 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 signals.

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.

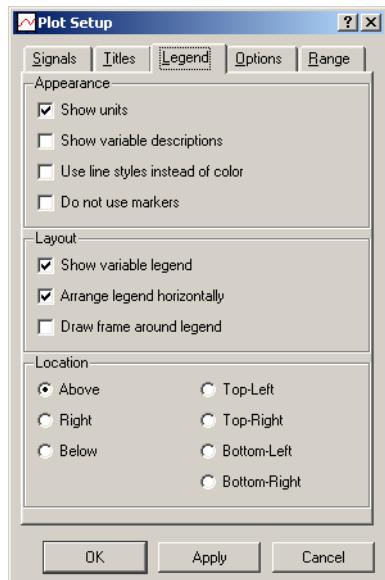
None No title is shown by the horizontal axis.

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.

Examples of user-defined heading and custom axis titles are shown on page 156.

Legend tab of the plot window setup.



Legend tab The legend displays names and other properties about plotted signals. The appearance of the signals is set as follows:

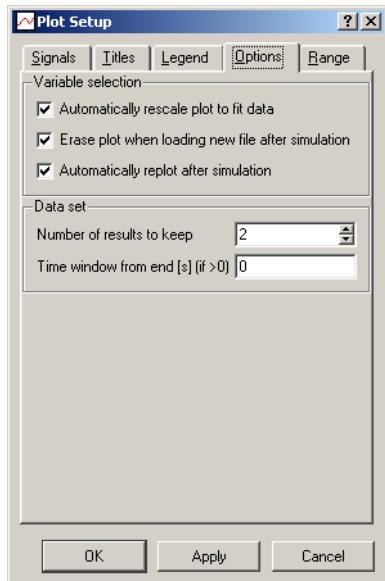
- **Show units** of variables in the vertical axis title or in the legend. The unit is shown in the axis title (if all signals have the same unit) or in the legend (if there are multiple units on one axis).
- **Show variables description** adds the description of the variable, as specified in the model, to the vertical axis title or the legend. The variable description is shown along the axis, if it is the same for all signals.
- **Use line styles instead of color** (solid, dashed, dotted) to distinguish curves. This is useful for printing diagrams on monochrome printers. If disabled, curves are distinguished by color.
- **Do not use markers** to distinguish curves with identical color or line style. By default markers are used when so many curves are plotted that colors or line styles must be reused.

The layout of the legend is controlled by these settings:

- **Show variable legend** is the default. If unchecked, no legend is displayed.
- **Arrange legend horizontally** lists the signals from left to right in the legend (the default). If unchecked, the signals are listed vertically in the legend.
- **Draw a frame around the legend** when outside of diagram. Inside the diagram the frame is always drawn.

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 page 156 for examples of diagrams with different locations of the legend.

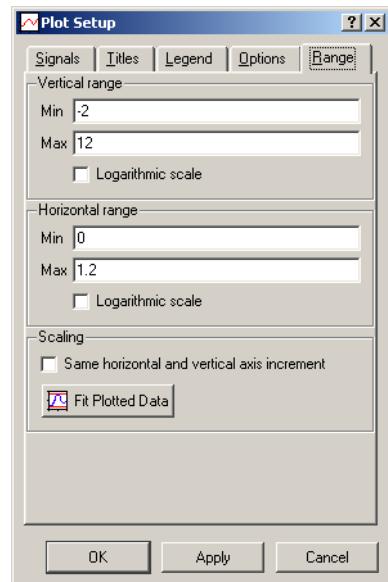
Options tab of the plot window setup.



Options tab Sets up options controlling plotted variables:

- **Automatically rescale plot to fit data** when zoomed out. When diagram is zoomed in, automatic rescaling is disabled.
- **Erase plot when loading new file after simulation** to have automatic erase of the diagram when new results are load (and thus when a simulation is run).
- **Automatically replot after simulation**; if not selected signals must be manually plotted after a simulation.
- **Number of results to keep when simulating**. Manually loaded results are not included.
- **Time window from end** defines the size of the time window during online plotting.

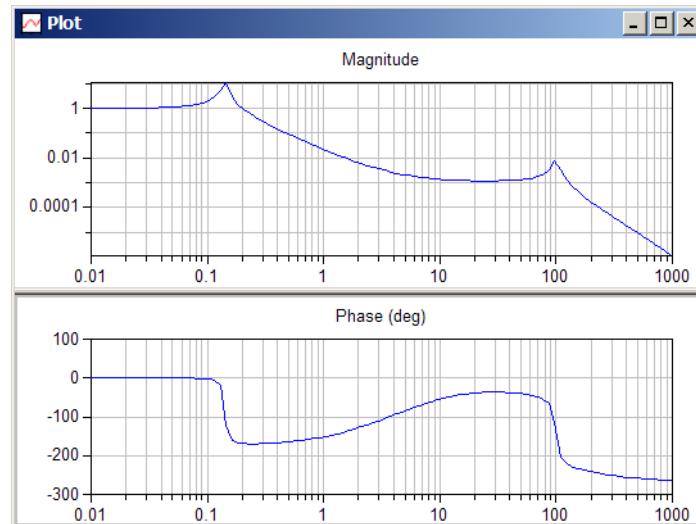
Range tab of the plot window setup.



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.

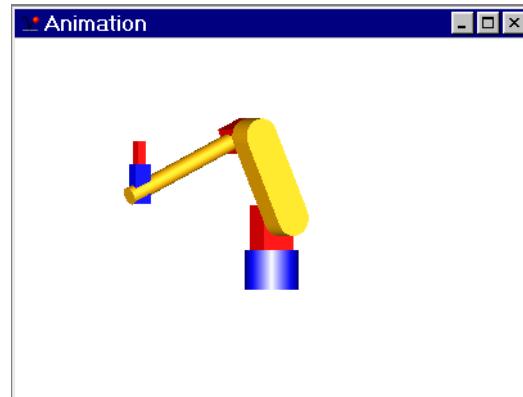
Logarithmic scale used in Bode plot.



Animation window

Animating models requires the Animation option.

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 submodels, 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, Beam, Gearwheel 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.

Defining Graphical Objects

Graphical 3D objects can either be created by using one of the models in ModelicaAdditions.MultiBody.Parts, or by directly declaring a VisualShape object. The vector visualizers are created by using one of the models in ModelicaAdditions.MultiBody.Visualizers, note that the force and torque sensors must be inserted between the objects. The model VisualShape has the following interface:

```
model VisualShape "General Visual Object."
parameter Real r0[3] = {0, 0, 0}
  "Origin of visual object.";
parameter Real Length = 1 "Length of visual object.;"
```

```

parameter Real Width = 1 "Width of visual object.";
parameter Real Height = 1 "Height of visual object.";
parameter Real LengthDirection[3] = {1, 0, 0}
    "Vector in length direction.";
parameter Real WidthDirection[3] = {0, 1, 0}
    "Vector in width direction.";
parameter String Shape = "box" "Name of shape";
parameter Real Material[4] = {1, 0, 0, 0.5}
    "Color and specular coefficient.";
parameter Real Extra = 0.0
    "Additional parameter for cone and pipe.";
input Real S[3,3] "3 x 3 transformation matrix.";
input Real r[3] "Position of visual object.";
...
end VisualShape;

```

Note that r is given in the global coordinate system, whereas the vectors r0, LengthDirection, and WidthDirection are in the local coordinate system as defined by S.

Shape specifies the name of the shape. Predefined shapes are: “box”, “sphere”, “cylinder”, “cone”, “pipe”, “beam”, “gearwheel” and “wirebox”. External shapes are specified as DXF-files (AutoCAD R12/LT2 format, only 3DFace is supported). External shape must be named like “1”, “2” etc. The corresponding definitions should be in files 1.dxf, 2.dxf etc. 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 are found relative to the directory of the current model.

Material[4] = {r,g,b,specular} specifies the color of the object. {r,g,b} affects the color of diffuse and ambient reflected light. Specular is a coefficient defining white specular reflection. Note, r g, b and specular are given in the range 0-1. Specular=1 gives a metallic appearance.

The position r and orientation S are inputs to control the movements of the object. An example showing some of the available shapes is given below.

```

model shapes
parameter Real r[3] = {0,0,0};
parameter Real Length=0.2,Width=0.2,Height=0.2;
parameter Real LengthDirection[3] = {1,0,0};
parameter Real WidthDirection[3] = {0,1,0};
VisualShape v1 (r0 = r+{0,-0.5,0}, Shape = "box",
    Length = Length, Width = Width, Height= Height,
    LengthDirection = LengthDirection,
    WidthDirection = WidthDirection);
VisualShape v2 (r0 = r+{0.4,-0.5,-0.1},
    Shape = "cylinder", Material = {0, 1, 0, 0.75},...);
VisualShape v3 (r0 = r+{0.8,-0.5,-0.2}, Shape = "cone",
    Material = {0, 0, 1, 1},...);
VisualShape v4(r0 = r+{0,-0.1, 0}, Shape = "cone",
    Extra = 0.5, Material = {1, 1, 0, 0.5}, ...);

```

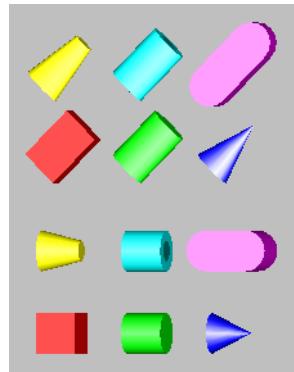
```

VisualShape v5 (r0 = r+{0.4,-0.1,-0.1},Shape = "pipe",
    Extra = 0.5, Material = {0, 1, 1, 0.75},...);
VisualShape v6 (r0 = r+{0.8,-0.1,-0.2}, Shape = "beam",
    Material = {1, 0, 1, 1});
equation
    v1.S = identity(3);
    v1.r = {0,0,0};
...
end shapes;

model VisualTest
    shapes s1(r=[0, -0.2, 0]);
    shapes s2(r=[0, 0.6, 0],
        Length=0.3, Width=0.2, Height=0.1,
        LengthDirection={1, 1, 0},
        WidthDirection={-1, 1, 0})
        "Rotated 45 degrees counter-clockwise";
end VisualTest;

```

Overview of available shapes.

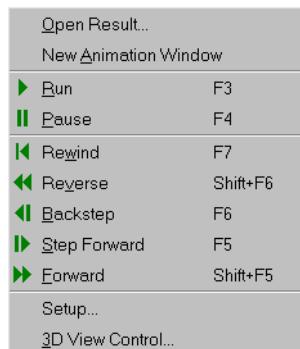


File menu

In the File menu the commands Export Image... and Export Animation... provide export of the animation window in several file formats, see page 136.

Animation menu

Animation menu



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.

The animation toolbar



Animation/Run (function key F3)

Starts an animation. The animation runs from the current frame until the last frame. If Dymola is in continuous mode, 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.



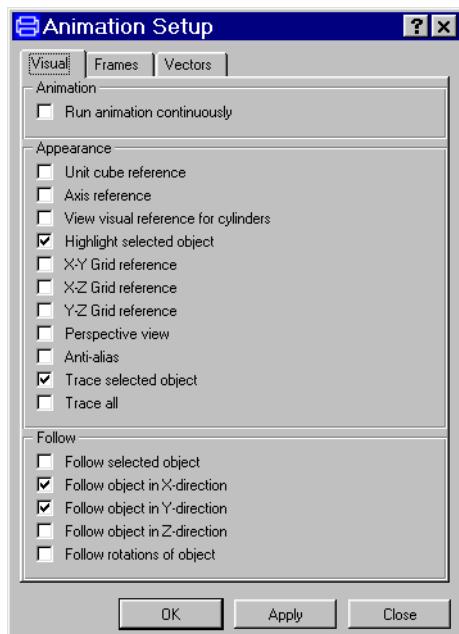
Animation/Forward (function key Shift+F5)

Moves the animation forward at high speed.

Animation/Setup...

Changes several options that control animation and viewing of animated objects.

The animation setup dialog.



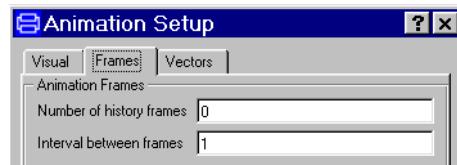
Visual tab contains

- **Run animation continuously**, which toggles continuous mode on or off. If continuous mode is on, running animations automatically restart 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.

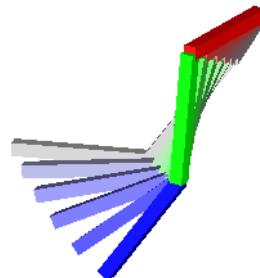
Frames tab sets a number of attributes related to history frames. History frames gradually fade to the grey color of the background.

Animation frames tab.

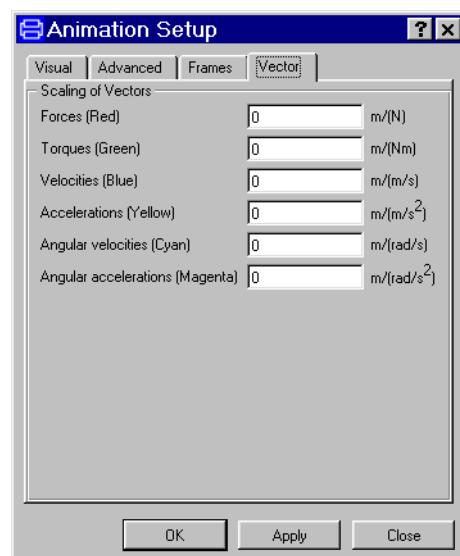


- **Number of history frames** introduces motion blur showing several frames. Zero will turn off all history frames. A large number of history frames may slow down the animation considerably. If the number of history frames times the interval between history frames is greater than the total number of 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.

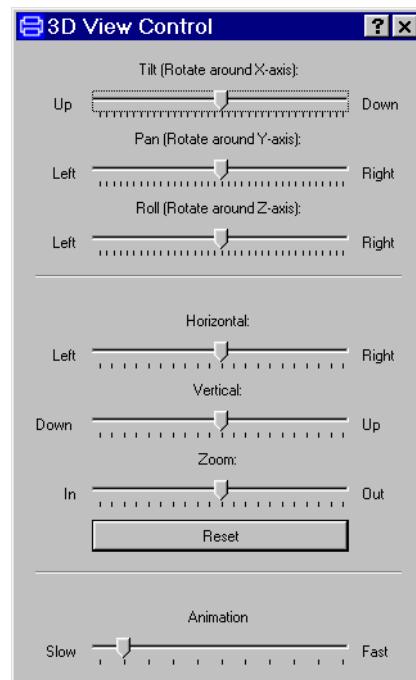
History frames for the Furuta pendulum.



Vector tab is for controlling the scaling of vector visualizers (found in ModelicaAdditions.MultiBody.Visualizers). 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 ($=1e-3 \text{ m/N} \cdot 100\text{N}$) long in scale of the diagram.

Vector scaling setup.**Animation/3D View Control...**

Creates a window with controls for 3D viewing. The view controls act on an imaginary camera; panning the camera left moves the subject closer to the right edge of the window.

3D view control.

Scripting language

There is a script facility that makes it possible to load model libraries, set parameters, set start values, simulate and plot variables by executing scripts. This is how the demo examples are setup. The script facility is useful when running a series of simulations as for example a parameter study.

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. Named arguments allowed to functions like simulateModel with default values. Modelica expressions can be stored in script files or as Modelica functions.

Interaction

By typing a Modelica expression at the command prompt, the result is output:

```
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. The character - denotes a prompt.

It is possible to continue a statement on several input lines

```
 {{1,2},{3,4}}*{{1,2},{3,4}}*
 Continue line:
 transpose({{1,2},{3,4}}*{{1,2},{3,4}})*
 Continue line:
 Modelica.Math.sin({{1,2},{3,4}}*{{1,2},{3,4}})
 =
 [309.234551260163, (-83.9358210418974);
  674.574723255002, (-183.082439325942)]
```

by ending a line before a complete assignment or expression has been entered. This rule allows omitting the semicolon at the end of a statement and/or using a continuation symbol. A tool might enlarge input window to show all pending input lines.

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, 1.5, 2}

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] = 7.5

```

```

s[2]
= "script"

t = s[1] + " " + s[2] + "ing: r[2] = " + realstring(r[2],1,
1)
Declaring variable: String t ;

t
= "Modelica scripting: r[2] = 7.5"

```

A list of the interactive variables and their values can be obtained by the command list():

```

list()
Integer i[3] = {0, 0, 0};
Real r[3] = {1, 7.5, 2};
Boolean b[2] = {false, true};
String s[2] = {"Modelica", "script"};
String t = "Modelica scripting: r[2] = 7.5";

```

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 Setup menu, where e.g. Evaluate corresponds to selecting Evaluate parameters.

Functions for string manipulation

The string manipulation routines are constructed for formatting numerical results as strings. The routines are realString and integerString for converting numbers to strings and the Modelica built-in operator + that concatenate strings.

They behave as Modelica functions declared as:

```

function realString
    input Real number;
    input Integer minimumWidth:=1;
    input Integer precision:=6;
    output String result;
end realString

function integerString
    input Integer number;
    input Integer minimumWidth:=1;
    input Integer precision:=1;
    output String result;
end integerString;

```

These functions are can be used for presenting results and for accessing a sequence of files:

```

r:=12.3;

"Modelica scripting: r = "+realString(number=r);
="Modelica scripting: r = 12.3"

sumA=0;

for i in 1:10 loop
  A=readMatrix("a"+integerString(precision=2,
  number=i)+".txt","A",2,2);
  sumA:=sumA+sum(A);
end for;

```

Script files

Script files can be automatically constructed. First, select File/Clear Log. Then perform interactively the operations by using menu commands or by entering written commands. When done select File/Save Log. For “Save as type” select “As script file (*.mos)” see “File/Save Log...” on page 137. Dymola will record all written commands and also all menu commands that have an equivalent Modelica script function.

Since the script files are readable text-files they can easily be cleaned up, parameter values changed, and a simple interactive session can be changed to a parameter-sweep by adding an enclosing for-loop.

The scripts can later be run by selecting Run script in the menus, and interactively by using the @-command:

```
@ file_specification
```

The rest of the line following @ is treated as the file specification. The syntax of the file specification is thus operating system dependent. The script feature is recursive, i.e. a script file may run other other script files.

If an error is encountered in a command script, all script files are closed and interactive command input is resumed.

Help commands

These routines all available for interactive help:

help

```
help()
```

Gives a short overview of these commands

listfunctions

```
listfunctions()
```

A one-line description of all builtin functions.

```
listfunctions()
function ndims "number of dimensions of array";
```

```

function size "size of array";
function scalar "convert an array to a scalar";
function vector "convert an array to a vector";
function matrix "convert an array to a matrix";
function transpose "transpose a matrix";
function outerproduct
    "compute the outer product of two vectors";
function identity "identity matrix";
function diagonal "construct a diagonal matrix";
function zeros "construct an array of zeros";
function ones "construct an array of ones";
function fill "fill an array";
function symmetric "make matrix symmetric";
function cross "cross-product of 3-vectors";
function skew "skew-matrix constructed from 3-vector";
function cat "concatenate arrays";
function array "construct array of arrays";
function max "computes max of an array";
function min "computes min of an array";
function sum "computes sum of an array";
function product "computes product of an array";
function readMatrix "read a matrix from a file";
function readMatrixSize "read size of a matrix from a file";
function writeMatrix "write a matrix to a file";
function interpolateTrajectory
    "interpolates a trajectory on a file";
function readTrajectory
    "return all output points of trajectory";
function readTrajectorySize
    "return number of output points of trajectory";
function linspace "vector of linearly spaced values";
function assert "assert that a condition is true";
function terminate "terminate simulation";
function help "List help commands";
function list "list variables on screen";
function listfunctions "list builtin functions on screen";
function document "write calling syntax for named function";
function realString "Convert a real to a string.";
function integerString "Convert an integer to a string.";
function cardinality "cardinality of connector";
function direction "direction of connector";
function constrain
    "a special operator for kinematic loop constraints";
function initialized "start of result from initializer";
function cos;
function sin;
function tan;
function exp;
function ln;
function sqrt;

```

```

function arcsin;
function arccos;
function arctan;
function arctan2;
function sinh;
function cosh;
function tanh;
function ceil;
function floor;
function integer;
function sign;
function simplesign;
function div;
function rem;
function abs;
function delay;
function openModel "open a Modelica-file";
function instantiateModel "instantiate a model";
function translateModel "translate a model";
function checkModel "check a model";
function closeModel;
function simulateModel "simulate a Modelica model";
function linearizeModel "linearize a Modelica model";
function plot "plot given variables";
function printPlot "plot and print given variables";
function plotArray "plot given data";
function printPlotArray "plot and print given data";

```

document

```
document ("func")
```

Write the Modelica declaration of the given builtin function, and describe the function.

```

document ("document")
function document "write calling syntax for named function"
    input String _function "name of builtin function";
    output Boolean _result "true if successful(i.e. function
existed)";
end document;
= true

```

Simulator API

These routines all return a Boolean status to indicate the success (true) or failure (false).

openModel

```
openModel ("file.mo")
```

Reads the specified file and displays its window. This corresponds to File/Open in the menus. Note: This will automatically cd to the right directory.

checkModel

```
checkModel(problem="model")
```

Check the model validity. This corresponds to Check Model in the menus.

translateModel

```
translateModel(problem="model")
```

Compile the model (with current settings). This corresponds to Translate Model in the menus

simulateModel

```
simulateModel(problem="model", startTime=0, stopTime=1,  
              numberOfIntervals=0, outputInterval=0, method="dassl",  
              tolerance=1e-4, resultFile="model")
```

Simulate the model for the given time. Method is a string with the name of the integration method; the names correspond to the ones found in the popup-menu and the string is case insensitive. 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.

closeModel

```
closeModel()
```

Erases all model variables from the work-space. You have to call closeModel() and reset all parameters and starting values to simulate the same model with different parameters and/or start values.

importInitial

```
importInitial(dsName="dsfinal.txt")
```

Sets up integration or linearization to start from the initial conditions given in the file (including start and stop-time and choice of integration method). The default is "dsfinal.txt" which in combination with **simulate** corresponds to **continue**.

After calling importInitial it is possible to override specific parameters or start-values before simulating.

Note: Basically importInitial() corresponds to copying dsfinal.txt to dsin.txt.

exportInitial

```
exportInitial(dsName="....txt", scriptName="....mos")
```

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.

plot

```
plot({"plot1","plot2","plot3"....})
```

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.

plotArray

```
plotArray(x=1:10,y=sin(1:10))
```

X-y plot for given values.

printPlot

```
printPlot({"plot1","plot2","plot3"....})
```

Plot the variables and furthermore prints the resulting plot on the default printer.

list

```
list()
```

Lists the variables in the variable workspace with their type and value. Predefined variables are also described by a comment.

The command Clear All also clears the variable workspace.

eraseClasses

```
eraseClasses({"model1","PackageA.model2", ...})
```

Erases the given models. It requires that no models outside of this list depend on them. This is not primarily an interactive function, but designed to be called by other programs constructing and changing models. Corresponds to ‘Delete’ in package windows.

Note: translateModel and simulateModel 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);

Note: If you give the name of the model you can skip translateModel and go directly to simulateModel.

Note: The current version does not allow you to first compile the model and then change translator switches (or parameters if you have Evaluate=true), and then simulate the model.

You must call closeModel and translateModel if you have compiled the model and wants to change them.

Script functions

The Modelica based scripting language is extensible since functions written in Modelica can be called in the same way as the predefined functions. Thus the functions in the Modelica standard library can be used directly from the scripting language, allowing e.g. table interpolation.

As an example consider a function for reading the A-matrix of a linearization using readMatrix. The function can be written as follows:

```
function ReadA
    input String linfile:="dslin.mat";
    input String abcdname:="ABCD";
    input String nxname:="nx";

    protected
        Integer ABCDsizes[2]:=readMatrixSize(linfile, abcdname);
        Real ABCD[ABCDsizes[1], ABCDsizes[2]];
        Integer nx:=
            integer(scalar(readMatrix(linfile, nxname, 1, 1)));

    public
        output Real A[nx, nx];

    algorithm
        ABCD := readMatrix(linfile, abcdname,
                            size(ABCD, 1), size(ABCD, 2));
        A := ABCD[1:nx, 1:nx];
    end ReadA;
```

Because of the default values the function can be called as ReadA() without giving any arguments.

This can easily lead to a large number of top-level functions, and in order to avoid this it is advisable to create a Modelica package, e.g. myFunctions and place ReadA inside this package. The myFunctions package must be stored in the MODELICAPATH.

Provided this is done it then possible to call myFunctions.ReadA() from the command line or inside models without having to load any files first.

It is also possible 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 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.

Debugging models

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 overspecified problem is found. It may be the case that this subproblem also has under specified parts, i.e., variables that cannot be uniquely determined. Moreover, the subproblems 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.

Basic steps in debugging models

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, Sampled systems, MultiBody, 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 standard notation and types.
- Regularly check models and components to detect errors, see “File/Exit” on page 138.
- 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.
- Do not overuse advanced features of Modelica, such as replaceable classes, inner/outer.
- Use SaveAs or Duplicate to move a class or package in package hierarchies.

- Examine the start-values (x_0) and parameters to see that the start values are correct, see “Basic steps” on page 147.
- Avoid algorithms in non-functions.
- Beware of graphical lines that look like connections. Connections are created directly by drawing from a connector, see “Creating a connection” on page 121.

Several of these steps are common with normal object-oriented development.

Finding errors in models

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 160), 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 steps above, in particular that you have not ignored any warnings and errors detected by the translator, and have tested each submodel in isolation.

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 153. This makes it possible to track down errors in the discrete part of a hybrid system.

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 152, 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 an 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. It is also possible to linearize around various operating points in order to determine the stability of the linearized system. 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 submodel. 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).

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.

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.

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 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 deselecting storing of ‘States’, ‘Derivatives’, ‘Outputs’, and ‘Auxiliary Variables’, see “Output tab” on page 152, and then re-running the simulation. In this case no result file will be generated. By looking at the logfile it is possible to compare the time to normal simulations.

For very nice problems 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 “Simulation/Setup” on page 150. 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.

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 152, 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 148.

In some cases it is possible to replace a detailed model generating many events by an averaging modes 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.

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 in order to see that the simulation is progressing correctly.

It is also possible to activate these using from these individually before the simulation is started using the “Debug tab” on page 153, or by using the debug monitor 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 Microsoft Visual C++ compiler with DDE-support is **not** selected as compiler since the commands are entered in the DOS-window created when using the other compilers. See also “Compiler tab” on page 154.

Restore the DOS-window 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 in-

formation is 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:

Command name (abbreviation)	Action
continue (c)	continue simulation
qui t(q)	stop simulation
log (l)	give help for commands starting with l
log event true (le t)	activate logging of events
log norm true (ln t)	activate logging of dominating component
log singular true (ls t)	log and continue if singular systems

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.

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 has the error in the component exceeded the tolerance and forced the integrator to reduce the stepsize
Dominate error	How often has the component dominated the error estimate
Exceeds 10% of error	How often has the error contribution from the component exceeded 10% of total error estimate
Component	Name of the state
(#number)	State number

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.

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 the Microsoft Visual C++ compiler with DDE-support, since it has access to highly accurate timers (accuracy in microseconds or nano-seconds), see “Compiler tab” on page 154. Selecting other compilers will only generate timers with milli-second 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	Is the total time for all model evaluations
1	Is the total time for model evaluations during event handling
2	Is an empty block, included just for comparison.
3	Is 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 more fine 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 more coarse 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 observe 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.

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 Elmqvist *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 method. In the most simple case, using the Euler forward method, 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:

$$x_{n+1} = x_n + h \cdot f(x_n, t); \quad x_0 \text{ is known}$$

which is used to “solve” the ODE.

Unfortunately, explicit integration methods are not well suited if systems are stiff. These are systems with dynamically fast and highly damped components. If an explicit method is used to integrate such systems, the step size is limited due to stability problems of the integration method. If the step size is too large, then the computed trajectory starts to oscillate and diverge. The standard cure is to use implicit methods. However, this leads to a non-linear equation system that has to be solved at each step. The simplest example of an implicit method 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.

Inline integration in Dymola

Inline integration is turned on by issuing the command

```
Advanced_INLINEIntegration = true
```

on the command line before translating the model.

Euler forward is the default method. To select Euler backward issue the command

```
Advanced_INLINEIntegrationExplicit = false
```

Translate the model. Click the Specify simulation run button and select Euler as the integration method. You will get an error message if improper algorithm is selected. Step size can be implicitly defined as output interval or explicitly set by specifying “Fixed integrator Step”.

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.

Mode handling

Mode handling requires the Realtime option.

In many systems the structure of the system depends on discrete states such that there is a linear system with a Jacobian that can only have a number of discrete values during a simulation. Normally we factorize the Jacobian at every step, but during continuous integration the matrix is not re-factorized, and thus the solution of the linear system contributes little to the overall time.

However, for real-time simulation we have a dead-line for each step and do thus want to decrease the maximum time for each step. Mode-handling does this by pre-compiling all modes (i.e. all possible Jacobians) used in the simulation. This avoids the costly re-factorizations, allows further symbolic manipulation for each mode, and by only performing it for the modes that actually occur it avoids an exponential explosion in code size.

Collecting modes

In order to collect modes one must first decide which variables determine the modes (this is done automatically by Dymola) and then which modes are interesting (automatically collected from simulations).

The following describes the preparatory steps during off-line simulation, and the result of these steps is a new model that with mode variables and modes.

1. open \${model} (to show it)

2. set Hidden.GetModeVariables=true
3. translate \${model} (this automatically detects mode variables and outputs code for finding new modes in dsmodel.c)
If the manual selection of mode variables fails it possible to manually set BigModeVariables in the model.
4. simulate to collect mode values in \${model}Modes.mat and output new model with mode values in \${model}ModeValues.mo
5. repeat the last step under various conditions to get additional modes (automatically merged)

Using mode information in real-time simulation

The above describes how to obtain the modes during off-line simulation, and we are now ready to use them during real-time simulation

6. (not necessary: set Hidden.GetModeVariables=false)
7. translate \${model}ModeValues using mode values (no new mode variables should be found) and simulate it (no new mode values should be found)

One potential problem is that intermediate modes (found during event iterations) can vary between off-line and real-time simulation. The only guard is to perform several simulations with different values. To debug this procedure a number of output variables are automatically generated to indicate which mode we are in and the size of the system of equations. These are outputs and can thus be accessed even on real-time hardware such as dSPACE.

Known Limitations

- The automatically collected modes are limited to 100 in numbers.
- There should be some possibility to automatically generate code for all modes.
- The switches should have names starting with Advanced instead of Hidden.
- The original model cannot appear inside a package (create a new model).
- If the mode variables are changed, but their number is constant you must remove the mat-file in order to avoid additional modes (with extra code).
- erasing \${model}Modes.mat and only redoing Table 7 on page 193 gives a new \${model}ModeValues that does not contain modes present in the model.

References

Mattsson, S. E., M. Otter, H. Elmquist (1999) "Modelica Hybrid Modeling and Efficient Simulation", in Proceedings of 38th IEEE Conference on Decision and Control, Phoenix, Arizona. Also available as <http://www.modelica.org/papers/ModelicaCDC99.pdf>.

DYNAMIC MODEL SIMULATOR

Dynamic Model Simulator

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 by Dymoview.

Dymosim can be used in other environments too. Special support is given for Matlab¹ and for ANDECS². However, it can also be used as a stand-alone program or can be called directly from a user-program as a C-function.

Who wrote Dymosim?

The first version of Dymosim was called DSSIM and was written by Norbert Gaus and Martin Otter as part of the ANDECS project in 1991 (Gaus91,Otter91). The DSSIM integrators are mainly based on high quality public domain software which was selected and enhanced by Claus Führer, Tuan van Tran and Martin Otter. In 1992-1993, major enhancements to DSSIM have been realized by Martin Otter (user interface, improved event handling), Reinhard Finsterwalder (off-line and on-line plotting features) and Alexander Lewald (parameterized input signals, especially suited for trajectory optimization).

In 1994, Dymosim, version 4.0, was realized by Martin Otter. Dymosim is essentially the kernel of DSSIM, version 3.3, which was separated from ANDECS and transformed from Fortran to C using the public domain f2c compiler of the Bell Laboratories. The file interface was newly written in C, replacing the previous interface of DSSIM to the RSYST database system.

Dynasim has then interfaced Dymola to this kernel and enhanced both the integrators and the interfaces between integrators and models.

Running Dymosim

When translating a model in Dymola an executable, Dymosim, is generated, and later when selecting *Perform simulation* 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 setting *Experiment, Preferences, Output Debug information* additional Debug information will be written to the log file.

By default, Dymosim stores the results of a simulation run in (Matlab) binary format on file (`dsres.mat`). The data can be plotted by clicking on the Plot Window icon of the workspace and selecting the variables to be plotted by *Variables*. Additional plot windows can be opened by clicking on the *Dymoview* icon in the workspace and by selecting *View, Plot Window*. Results from other files can be plotted within Dymoview by selecting the appropriate file via menu *File, Open Result*. As explained in the next section, the results can also be directly loaded into Matlab and plotted with the Matlab plot functions.

The translate commands generates the file `dsmodel.c`, compiles and links this to the generate the executable `dymosim.exe`, and then runs `dymosim -i` to generate default initial values. Compiling and linking uses a script `dsbuild` provided by Dymola in directory `$DYMOLA/bin`. If necessary, you can adapt this script to your needs, e.g. to choose different

-
1. Matlab stands for Mathematical laboratory and is a registered trademark of The Math-Works Inc.
 2. ANDECS stands for Analysis and Design of Controlled Systems and is a registered trademark of DLR.

compiler options.

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 Dymolas 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 line arguments, 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. The complete structure of the (self explanatory) input file is described on page Dymosim input file “`dsin.txt`”.

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`. The data on this file can be plotted by Dymoview. Alternatively, the result file can be imported into Matlab¹ by executing 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 the subdirectory `dymtools` another set of routines can be found that perform a similar task as `tload`, use `help dymtools` in Matlab for more information. The function `dymbrowse` allows you to select and plot signals directly in Matlab, and `dymload` and `dymget` allows you to extract signals from a result file. They use substantially less memory than `tload` by not storing the result in a simple matrix.

Another alternative for using less memory is to 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 plot selector, 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 in-

1. Matlab is a product of MathWorks Inc

put file of another model. It is convenient 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.

Dymosim and Matlab

Dymosim can be directly called as an m-function from Matlab. The power of Matlab can be used to perform parameter studies or optimization runs such that every “function evaluation” requires one simulation experiment. When called from Matlab, Dymosim runs as a separate process. The Dymosim executable has to be built in the same way as in the previous section. Dymosim is executed by the command

```
[s,n] = dymosim(experiment,x0,param);
```

where the three input arguments correspond to the three matrices of the Dymosim input file (see a more detailed description by command “`help dymosim`” within Matlab). The results of a simulation run are stored in matrix “`s`”. Every column of “`s`” contains the result data for one signal; column 1 is the time vector. The signal name of column “`i`”. is stored as string in row “`i`”. of character matrix “`n`”. A simulation of the Van der Pol equation, together with a plot of the result signals, can therefore be performed by the following commands:

```
[s,n] = dymosim( [0,50,0.1,0,1.E-4,1], [0.5;0], 0.2);
tplot(s,2:size(s,2),n) % plot all result signals
```

Often it is convenient to use the default input file `dsindef.txt` of Dymosim to define input parameters which do not change between simulation runs. The format of this file is identical to the Dymosim input file `dsin.txt`. If `dsindef.txt` is present, the data stored in this file are used as default values. The default values are overwritten by the data given as arguments to the `dymosim` m-file function.

Selecting the integration method

Dymosim provides a number of different integration methods for the simulation of dynamic systems. In this section the major characteristics of these methods are discussed, and rules of thumb are given which method should be selected for a problem at hand. Note however, that one should not rely on just one integration method for simulation experiments. Instead, some selected results should be checked by two or three other (different) integration methods.

First of all some important issues of integrators are explained, in order to classify the available integration methods 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 methods 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 component¹ 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 = tolrel`. This is not a good choice, if signals are permanently very small.

In the Dymosim input file it is possible to define a *scaling* value for every component of the state vector. The integrators effectively solve for the scaled state variables $x(i)/xscale(i)$. If some state variables are very small or if there are big differences in the magnitudes of elements of \mathbf{x} , the scaling vector should be assigned appropriate elements. E.g., if $x(1)$ is always near 10^{-5} and $x(2)$ is always near 10^5 , the scales `xscale(1) = 10^-5`, `xscale(2) = 10^5` should be used. For the scaled equations it makes again sense to use the relative tolerance as an absolute tolerance, too.

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

-
1. 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.

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 methods 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.

Variable step-size, dense output

Most integration methods available in Dymosim have a *variable step-size* algorithm. At every step, a method 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.

There is one important difference between integrators with respect to the step-size algorithm: The step-sizes of some integrators are not only influenced by the required tolerances but also by the communication time grid. The communication time grid is defined by the `StartTime`, the `StopTime` and a communication 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 communication points. The step-size is always chosen in such a way, that the integrator meets the grid points exactly. For such methods, the communication 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 communication step-size should be a multiple of this smaller fixed step-size.

On the other hand there are integration methods, called *dense output* methods, which treat communication points differently. The step-size of such integrators is not influenced by the communication grid. The step-size is only chosen according to the required tolerances and the estimated local error. Such methods integrate past the desired communication points and determine the values of the state variables \mathbf{x} at the communication points by interpolation, which involves no evaluation of the differential equation. This is advantageous, since the choice of the communication grid, even if a dense grid is used, has nearly no influence on the efficiency of the integration. In Dymosim, the maximum allowed step-size for dense output integrators can be explicitly restricted by parameter `method(hmax)` in the input file, additionally it is possible to turn off *dense output*.

Variable order

Integration methods approximate the solution $\mathbf{x}(t)$ internally by a polynomial of order k_{ord} . Some methods use a fixed order, other methods vary the order during the simulation. 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 $k_{ord}+1$ times. Therefore if it is known for example that the result of a system is not very smooth, a low order method should be chosen. In Dymosim, the maximum order of a variable order method can be explicitly set in the input file via `method(ordmax)`. If the

maximum order is set to one, the variable order integration methods reduce to the simple (explicit or implicit) Euler formula.

Stiff systems

“Usual” integrators get in trouble, if “fast” and “slow” signals are present in the solution $\mathbf{x}(t)$. 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 method is always limited by a method 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 method, 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 method 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 decreased, 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.

Dymosim integrators

At present, Dymosim provides ten different *variable step-size* integration methods. The most important characteristics of these methods are given in the following table:

alg.	method	model type	order	stiff	dense output	root finder	author(s)
1	DEABM	ODE	1-12	no	yes	no	Shampine, Gordon, Watts (Sham75, Sham80)
2	LSODE1	ODE	1-12	no	yes	no	Hindmarsh (Hind80)
3	LSODE2	ODE	1-5	yes	yes	no	Hindmarsh (Hind80)
4	LSODAR	ODE	1-12, 1-5	both	yes	yes	Petzold, Hindmarsh (Hind83)
5	DOPRI5	ODE	5	no	no	no	Kraft (Hair87)
6	DOPRI8	ODE	8	no	no	no	Kraft (Hair87)
7	GRK4T	ODE	4	no	no	no	Arnold (Kaps79)
8	DASSL	DAE	1-5	yes	yes	yes	Petzold (Petz82, Bren89)
9	ODASSL	ODAE	1-5	yes	yes	yes	Führer (Fueh88)
10	MEXX	ODAE	2-24	no	yes	no	Lubich (Lubi91)

There are also four different fixed step-size integration methods, intended for real-time sim-

ulations:

alg.	method	model type	order	stiff	dense output	root finder	author(s)
11	Euler	ODE	1	no	no	yes	Dynasim
12	Rkfix2	ODE	2	no	no	no	-
13	Rkfix3	ODE	3	no	no	no	-
14	Rkfix4	ODE	4	no	no	no	-

The three supported model types are abbreviated as:

ODE Ordinary Differential Equations.

DAE Differential Algebraic Equations (of perturbation index 1).

ODAE Overdetermined Differential Algebraic Equations.

In the following table, some hints are given when to use a specific integration method:

DEABM	If you don't know much about your problem or you don't want to think much about it, use method DEABM. DEABM is a robust code with conservative heuristics. Other codes like LSODE1 may be faster in some situations, however DEABM will often be more reliable. Furthermore DEABM is the only code in Dymosim, which will give you a warning if your problem appears to be stiff. If the calculation of the right hand side of the differential equation is relatively expensive and your problem is non-stiff or moderately stiff and/or you want a great many output points, use DEABM or LSODE1.
LSODE1	LSODE1 is similar to DEABM. It is also a multi-step method using the Adams/Basforth/Moulton formula. However some important details are realized differently. You can use LSODE1 in the same situations as DEABM.
LSODE2	If your system is stiff, use method LSODE2. This integrator uses a multi-step method with the BDF-formula, the so-called Gear method.
LSODAR	If you need a root finder to handle state events or if your system is stiff on some time intervals and non-stiff on other time intervals, use method LSODAR. LSODAR is a combination of LSODE1 and LSODE2. It starts with the algorithm of LSODE1 and switches between the algorithms of LSODE1 and LSODE2 depending on the stiffness of the system. If your system is non-stiff over the integration period you should get nearly the same results with respect to the number of function evaluations and computing time as with method LSODE1.
DOPRI5	If the calculation of the right hand side of the differential equation is relatively cheap and your problem is non-stiff, use method DOPRI5. This method works well for moderate relative error tolerance requirements of about $\text{tolrel}=10^{-4}$. For higher precision computations a higher order method like DOPRI8 might be more effective with respect to computing time. If you need a great many output points, DOPRI5 and DOPRI8 are not well suited. Use instead a dense output integrator, for example DEABM. Integrator DOPRI5 uses a Runge-Kutta method with the order-comparing step-size formulae of Prince and Dormand.

DOPRI8	Method DOPRI8 is similar to DOPRI5. The main difference lies in the different orders of the methods. DOPRI8 uses a Runge-Kutta method of order 8. DOPRI8 may be used in the same situations as DOPRI5 if a higher precision is required (say <code>tolrel=10^-8</code>).
GRK4T	If the calculation of the right hand side of the differential equation is relatively cheap and your problem is stiff and/or contains highly oscillating solution components, use method GRK4T. GRK4T is a A(89.3)-stable linearly-implicit Rosenbrock type method of order four, i.e, the stability region of the integrator is nearly the whole left half plane. GRK4T can be also used for non-stiff problems without any problems (but with a higher amount of computing time than a non-stiff solver as DOPRI5 or DOPRI8).
DASSL	If you have differential algebraic equations (DAE), you have to use DASSL, since this is the only DAE-solver available in Dymosim. DASSL is designed to integrate stiff systems using a BDF-method. Method DASSL calls integrator DASSL, if no indicator functions are defined (no step events can appear) and calls integrator DASSLRT otherwise. DASSLRT is a modified version of DASSL for which a root finder was added. Method DASSL may also be used to integrate (stiff) <i>ordinary differential equations</i> . Dymosim makes the necessary conversions of the right hand side.
ODASSL	ODASSL solves higher index differential algebraic equations or overdetermined differential algebraic equations. It is a modified version of DASSL. Therefore if you integrate an ODE system with ODASSL you will get nearly the same results with respect to the number of function evaluations and computing time as with method DASSL.
MEXX	MEXX is a variant of the MEXX family of codes to solve special index-2 differential algebraic equations with an optional integral invariant. It uses a half-explicit extrapolation algorithm of orders 2 through 24 and is designed to integrate non-stiff systems. Presently, it is not possible to use MEXX for the integration of ordinary differential equations.

Dymosim reference

Model functions for Dymosim

When user defined C-functions are called from model equations a reference to the source code file for these functions needs to be put in a file `userfunc.c` in the working directory.

```
#include "myfunc.c"
```

In addition to the Modelica functions, the functions `min(,)`, `max(,)` are supported. PackShape and PackMaterial are two functions provided to support visual objects. The following functions are also provided for inclusion in Dymola models when Dymosim is used as simulator.

status = LogVariable(x)

`LogVariable` is a function for helping tracking run-time problems in models. An equation like

```
status = LogVariable(x)
```

It will give a message in the event log of the form

```
x = 5.5
```

status = ReadMatrix("name", M)

Read matrix with name “name” from data file “dsdata.mat” into parameter matrix M.

status = WriteMatrix("name", M)

Write matrix M into matrix with name “name” in a new data file with name “name”.mat.

The following random generator functions should only be used within instantaneous equations. The integration routines do not support stochastic differential equations.

RandomUniform(Time)

Uniformly distributed random number, range [0, 1].

RandomNormal(Time)

Normally distributed random number with mean 0 and variance 1.

status = RandomSeed(seed)

Initializes the random number generator with another seed.

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 directory \$DYMOLA/mfiles/traj on Unix machines and by directory %DYMOLA%\mfiles\traj on PC/Windows. The Dymosim m-files are based on a *trajectory datastructure*:

A trajectory is defined by two matrices:	
s	A numeric matrix where column 1 is the time vector (= monotonically increasing values) and the other columns are the corresponding signal values. Intermediate values are obtained by linear interpolation. If a trajectory is discontinuous, two successive time instants are identical. Example: s = [0 0 0 1 1 2 2 4 4 3 9 6];
n	A string matrix, where row “j” of “n” is the name of the signal of column “j” of “s”. Example: n = ['time' 't*t ' '2*t ']

The following m-functions are provided (as usual, you get detailed help for an m-file command within Matlab by command “`help <name>`”):

<i>Functions on trajectories (data + names):</i>	
Dymosim	perform time simulation of a Dymola model.
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 names as legend.
tplotm	plot trajectories in multiple diagrams and use signal names as legend.
trange	find trajectory index range of 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 row numbers.

<i>Other functions:</i>	
tload	load trajectory (e.g. dymosim simulation) into Matlab work space.
tsave	save trajectory on mat-file (e.g. to be used as input signals for dymosim).
tfigure	set meaningful default values for figures (e.g. white background and same colors on screen and printer,)

Dymosim command line arguments

Dymosim is a stand alone program which accepts command line arguments with the following syntax:

- Usage: `dymosim [options] [file_in [file_out]]`
simulate a DSblock-model with dymosim.
- Options:
- s simulate (default).
 - h list command summary.
 - d file use input defaults from „file“, i.e., read first all input data from „file“ and afterwards read a subset of the input data from the Dymosim input file „dsin.txt“ or from „[file_in]“.
 - i generate Dymosim input file (containing e.g. the names and default /initial values of all parameters and state variables).
 - ib generate Dymosim input file in (Matlab) binary format (mat-file).
 - it generate Dymosim input file using timer defaults (for realtime Dymosim).
 - p precede Dymosim output lines by program name (for realtime Dymosim, in order to identify the process which prints the line).

- v verify input, i.e., store complete input in file „dsinver.txt“.
- c cmd execute command cmd after simulation end. This can be useful if dymosim is started as a separate process in the background.

Basic file format

The input file as well as the result file of Dymosim have the same format. The data may be given in ASCII (human readable) or binary format. The binary format is identical to the Matlab binary format, i.e., the Dymosim input file can be generated by the Matlab “`save`” command, whereas the Dymosim result file can be loaded into Matlab by the Matlab “`load`” command.

The contents of a file consists of a set of named matrices of type `char`, `int`, `float`, or `double`. In ASCII-format, a matrix has the following structure:

```
<type> <name> (<row-dimension>, <column-dimension>) <row_1>
# elements separated by blanks; may span several
lines<row_2>
      ...
<last row>
```

Blank lines are ignored, as well as all characters which are present after the comment character “#”. The first matrix on every file is a “`char`” matrix with name “`class`”. The first row of this matrix contains the class-name of the data on file. This name allows the identification of the type of data stored in the file. The second row contains a class-specific version number. Finally the third row contains a description text of the data, which is an arbitrary string provided by the program which generated the file.

If desired, the ASCII form of files can be directly generated by hand. With the provided program “`alist`” the ASCII and binary form can be transformed into each other. E.g. “`alist file`” lists file “`file`” on standard output in ASCII format, independent of the previous format of the file. Command “`alist -h`” produces the following short description:

Usage: *alist [options] files*

transform Dymosim files and write them to standard output.

Options:

- a write in ASCII format (default).
- b write in Matlab binary format.
- h help (list this description).
- d transform single precision matrices into double precision.
- s transform double precision matrices into single precision.

Example:

alist -as dsres.mat > dsres.txt

transform file „`dsres.mat`” from binary into ASCII format, change the precision to float and store the result on file „`dsres.txt`“.

Dymosim input file “dsin.txt”

A default version of the Dymosim input file is generated by command “dymosim -i” in ASCII format. The Dymosim input file currently have the following structure (input file of a simple circuit model). The dymosim program is also able to read old versions of this format, and the file format is continuously extended with additional settings. Since the files are self-documenting the best documentation can be found by generating it by “dymosim -i” and reading the comments:

```
#1
char Aclass(3,24)
Adymosim
1.4
Modelica experiment file

# Experiment parameters
double experiment(7,1)
    0           # StartTime      Time at which integration starts
    #           (and linearization and trimming time)
    1           # StopTime       Time at which integration stops
    0           # Increment        Communication step size, if > 0
    500          # nInterval      Number of communication intervals, if > 0
    1.00000000000000E-004 # Tolerance     Relative precision of signals for
                           #           simulation, linearization and trimming
    0           # MaxFixedStep   Maximum step size of fixed step size
                           #           integrators, if > 0.0
    8           # Algorithm      Integration algorithm as integer (1...14)
    #
    #           | model|           |           | dense | state |
    # Algorithm | typ | stiff | order | output| event |
    # -----+-----+-----+-----+-----+-----+
    # 1 | deabm | ode | no | 1-12 | yes | no |
    # 2 | lsode1 | ode | no | 1-12 | yes | no |
    # 3 | lsode2 | ode | yes | 1-5 | yes | no |
    # 4 | lsodar | ode | both | 1-12,1-5 | yes | yes |
    # 5 | dopri5 | ode | no | 5 | no | no |
    # 6 | dopri8 | ode | no | 8 | no | no |
    # 7 | grk4t | ode | yes | 4 | no | no |
    # 8 | dassl | dae | yes | 1-5 | yes | yes |
    # 9 | odassl | hdae | yes | 1-5 | yes | yes |
    # 10 | mexx | hdae | no | 2-24 | no | no |
    # 11 | euler | ode | no | 1 | no | yes |
    # 12 | rkfix2 | ode | no | 2 | no | yes |
    # 13 | rkfix3 | ode | no | 3 | no | yes |
    # 14 | rkfix4 | ode | no | 4 | no | yes |
    # -----+-----+-----+-----+-----+-----+
    # euler and rkfix have fixed stepsize.

# Method tuning parameters
double method(27,1)
    1           # grid      type of communication time grid, defined by
```

```

#           = 1: equidistant points ("Increment/nInterval")
#           = 2: vector of grid points ("tgrid")
#           = 3: variable step integrator (automatically)
#           = 4: model (call of "increment" in Dymola, e.g.
#                  incr=Time > 2 then 0 else 0.1
#                  dummy=increment(incr))
#           = 5: hardware clock (functions "udstimerXXX")
#           grid = 1,3 is stopped by "StopTime"
#           grid = 2   is stopped by "tgrid(last)"
#           grid = 4   runs forever (stopped by model)
#           grid = 5   runs forever (stopped by udstimer-
Run)
1          # nt      Use every NT time instant, if grid = 3
3          # dense   1/2/3 restart/step/interpolate GRID points
0          # evgrid 0/1 do not/save event points in comm. time grid
1          # evu     0/1 U-discontinuity does not/trigger events
0          # evuord  U-discontinuity order to consider (0,1,...)
0          # error   0/1/2 One message/warning/error messages
0          # jac     0/1 Compute jacobian numerically/by BLOCKJ
0          # xd0c   0/1 Compute/set XDO
0          # f3      0/1 Ignore/use F3 of HDAE (= index 1)
0          # f4      0/1 Ignore/use F4 of HDAE (= index 2)
0          # f5      0/1 Ignore/use F5 of HDAE (= invar.)
0          # debug   0/1 do not/print debug information
100         # pdebug  priority of debug information (1...100)
0          # fmax   Maximum number of evaluations of BLOCKF, if > 0
0          # ordmax Maximum allowed integration order, if > 0
0          # hmax   Maximum absolute stepsize, if > 0
0          # hmin   Minimum absolute stepsize, if > 0 (use with
care!)
0          # h0      Stepsize to be attempted on first step, if > 0
1.0e-014   # teps   Bound to check, if 2 equal time instants
1.00e-10    # eveps  Hysteresis epsilon at event points
20         # eviter Maximum number of event iterations
1.000000000000000E-006 # delaym Minimum time increment in delay buffers
1          # fexcep 0/1 floating exception crashes/stops dymosim
1          # tscale  clock-time = tscale*simulation-time,if grid = 5
#           > 1: simulation too slow
#           = 1: simulation-time = real-time
#           < 1: simulation too fast
1          # shared   type of process communication, if grid = 5
#           = 0: no communication,(single process without clock)
#           = 1: no communication (single process with clock)
#           = 2: shared memory (multiple processes with clock)
2473       # memkey  key to be used for shared memory, if shared = 2

#      Output parameters
int settings(13,1)
0          # lprec   0/1 do not/store result data in double
1          # lx      0/1 do not/store x (state variables)
1          # lxd     0/1 do not/store xd (derivative of states)
0          # lu      0/1 do not/store u (input signals)
1          # ly      0/1 do not/store y (output signals)
0          # lz      0/1 do not/store z (indicator signals)

```

```

1                      # lw      0/1 do not/store w  (auxiliary signals)
1                      # la      0/1 do not/store a  (alias      signals)
0                      # lperf   0/1 do not/store performance indicators
0                      # levent  0/1 do not/store event point
1                      # lres    0/1 do not/store results on result file
0                      # lshare  0/1 do not/store info data for
shared memory on dsshare.txt
1                      # lform   0/1 ASCII/Matlab-binary storage format of
results
#                      # (for simulation/linearization; not for trimming)

#      Names of initial variables
char initialName(9,8)
C.C
R.R
C.v
C.der(v)
G.p.v
G.p.i
C.n.v
C.n.i
R.p.v

double initialValue(9,6)
-1 1 0 0 1 0  # C.C
-1 1 0 0 1 0  # R.R
-1 1 0 0 2 0  # C.v
0 0 0 0 3 0  # C.der(v)
0 0 0 0 6 0  # G.p.v
0 0 0 0 6 0  # G.p.i
0 0 0 0 6 0  # C.n.v
0 0 0 0 6 0  # C.n.i
0 0 0 0 6 0  # R.p.v

# Matrix with 6 columns defining the initial value calculation
# (columns 5 and 6 are not utilized for the calculation but are
# reported by dymosim via dymosim -i for user convenience):
#
# column 1: Type of initial value
#           = -2: special case: for continuing simulation (column 2 = value)
#           = -1: fixed value (column 2 = fixed value)
#           =  0: free value, i.e., no restriction (column 2 = initial value)
#           >  0: desired value (column 1 = weight for optimization
#                   column 2 = desired value)
#           use weight=1, since automatic scaling usually
#                   leads to equally weighted terms
# column 2: fixed, free or desired value according to column 1.
# column 3: Minimum value (ignored, if Minimum >= Maximum).
# column 4: Maximum value (ignored, if Minimum >= Maximum).
#           Minimum and maximum values are used for scaling purposes
#           both in the simulator and for the initial value calculation.
#           Furthermore, they restrict the search range in initial
#           value calculation.

```

```

# column 5: Category of variable.
#      = 1: parameter.
#      = 2: state.
#      = 3: state derivative.
#      = 4: output.
#      = 5: input.
#      = 6: auxiliary variable.
# column 6: Data type of variable.
#      = 0: real.
#      = 1: boolean.
#      = 2: integer.
#
# Initial values are calculated according to the following procedure:
#
# - If parameters, states and inputs are FIXED, and other variables
#   are FREE, no special action takes place (default setting).
#
# - If there are only FIXED and FREE variables and the number of
#   FREE parameters, states and inputs is IDENTICAL to the number of
#   FIXED state derivatives, outputs and auxiliary variables, a non-linear
#   equation is solved to determine a consistent set of initial conditions.
#
# - In all other cases the following optimization problem is solved:
#   min( sum( weight(i)*( (value(i) - DESIRED(i))/scale(i) )^2 ) )
#   under the constraint that the differential equation is fulfilled
#   at the initial time. In most cases weight(i)=1 is sufficient, due
#   to the automatic scaling (if DESIRED(i) is not close to zero,
#   scale(i) = DESIRED(i). Otherwise, the scaling is based on the
#   minimum and maximum values given in column 3 and 4. If these values
#   are zero, scale(i)=1 is used).
#
char initialDescription(9,53)
Capacitance [F]
Resistance [Ohm]
Voltage drop between the two pins (= p.v - n.v) [V]
Voltage drop between the two pins (= p.v - n.v) [V/s]
Potential at the pin [V]
Current flowing into the pin [A]
Potential at the pin [V]
Current flowing into the pin [A]
Potential at the pin [V]

```

The matrices may be in any order on the file. If a matrix is missing, a default matrix is used. If a matrix has dimension “ $(0,0)$ ”, the matrix is just ignored. It is allowed to supply only a subset of the parameters or initial values of state variables. If a parameter or state variable name is provided which is not present in the model, Dymosim prints a warning message and ignores this variable. This feature is useful, e.g., when a simulation run should be performed with an input file from a previous run of a slightly different model.

Simulation result file “dsres.mat”

On the Dymosim result file the results of a simulation are stored, as well as the corresponding signal names. The amount of output on the result file is defined by matrix “settings” in the input file. An example of the result file in ASCII format is given in the following table. The file format is also self-describing, explaining how aliases and constant variables are stored compactly

```
#1
char Aclass(3,11)
Atrajectory
1.1

char name(16,8)
Time
C.v
G.p.v
G.p.i
C.n.v
C.n.i
R.p.v
C.der(v)
C.i
C.p.v
C.p.i
R.v
R.i
R.p.i
R.n.v
R.n.i

char description(16,53)
Time in [s]
Voltage drop between the two pins (= p.v - n.v) [V]
Potential at the pin [V]
Current flowing into the pin [A]
Potential at the pin [V]
Current flowing into the pin [A]
Potential at the pin [V]
Voltage drop between the two pins (= p.v - n.v) [V/s]
Current flowing from pin p to pin n [A]
Potential at the pin [V]
Current flowing into the pin [A]
Voltage drop between the two pins (= p.v - n.v) [V]
Current flowing from pin p to pin n [A]
Current flowing into the pin [A]
Potential at the pin [V]
Current flowing into the pin [A]

# Matrix with 4 columns defining the data of the signals:
#
```

```

# dataInfo(i,1)=      j: name i data is stored in matrix "data_j".
#                           (1,1)=0, means that name(1) is used as abscissa
#                           for ALL data matrices!
#
# dataInfo(i,2)=      k: name i data is stored in column abs(k) of matrix
#                           data_j with sign(k) used as sign.
#
# dataInfo(i,3)=      0: Linear interpolation of the column data
#                           = 1..4: Piecewise convex hermite spline interpolation
#                           of the column data. Curve is differentiable upto
#                           order 1..4. The spline is defined by a polygon.
#                           It touches the polygon in the middle of every segment
#                           and at the beginning and final point. Between such
#                           points the spline is convex. The polygon is also the
#                           convex envelope of the spline.
#
# dataInfo(i,4)=      -1: name i is not defined outside of the defined time range
#                           = 0: Keep first/last value outside of time range
#                           = 1: Linear interpolation through first/last two points outside
#                           of time range.
#
int dataInfo(16,4)
    0  1  0 -1  # Time
    2  2  0 -1  # C.v
    1  2  0  0  # G.p.v
    2  3  0 -1  # G.p.i
    1  3  0  0  # C.n.v
    2  4  0 -1  # C.n.i
    1  4  0  0  # R.p.v
    2  5  0 -1  # C.der(v)
    2 -4  0 -1  # C.i
    2  2  0 -1  # C.p.v
    2 -4  0 -1  # C.p.i
    2 -2  0 -1  # R.v
    2 -4  0 -1  # R.i
    2 -4  0 -1  # R.p.i
    2  2  0 -1  # R.n.v
    2  4  0 -1  # R.n.i

float data_1(2,4)
    0                 0                 0                 0
    1                 0                 0                 0

float data_2(6,5)
0.0000000E+000 1.0000000E+000 0.0000000E+000 1.0000000E+000 -1.0000000E+000
2.0000000E-001 8.1863302E-001 0.0000000E+000 8.1863302E-001 -8.1882674E-001
4.0000001E-001 6.7018896E-001 0.0000000E+000 6.7018896E-001 -6.6999793E-001
6.0000002E-001 5.4872990E-001 0.0000000E+000 5.4872990E-001 -5.4875731E-001
8.0000001E-001 4.4920427E-001 0.0000000E+000 4.4920427E-001 -4.4958425E-001
1.0000000E+000 3.6774468E-001 0.0000000E+000 3.6774468E-001 -3.6768275E-001

```

Bibliography

- [Bren89] Brenan K.E., Campbell S.L. and Petzold L.R.: Numerical Solution of Initial--Value Problems in Differential--Algebraic Equations. Elsevier Science Publishers, 1989
- [Fueh88] Führer C.: Differential-algebraische Gleichungssysteme in mechanischen Mehrkörpersystemen. Ph.D. dissertation, Mathematisches Institut, Technische Universität München, 1988.
- [Gaus91] Gaus N. and Otter M.: Dynamic Simulation in Concurrent Control Engineering. IFAC Symposium on Computer Aided Design in Control Systems, Swansea, UK, Preprints pp. 123-126, July 15-17, 1991.
- [Gay93] Gay D., Feldman S., Maimone M. and Schryer N.: A Fortran-to-C Converter. Computing Science Technical Report No. 149, AT&T Bell Laboratories, Murray Hill N.J. 07974. The f2c compiler and the documentation is available via anonymous ftp from [netlib.att.com](ftp://netlib.att.com), cd netlib/f2c.
- [Hind80] Hindmarsh A.C.: LSODE and LSODI, Two New Initial Value Ordinary Differential Equation Solvers. ACM-Signum Newsletter, vol. 15, no. 4, pp. 10-11, 1980.
- [Hind83] Hindmarsh, A.C.: ODEPACK, a systematized collection of ODE solvers. Scientific Computing, edited by R.S. Stepleman et. al., North-Holland, Amsterdam, 1983.
- [Hair87] Hairer E., Norsett S.P. and Wanner G.: Solving Ordinary Differential Equations. Nonstiff Problems. Springer-Verlag, Berlin, 1987.
- [Lubi91] Lubich C.: Extrapolation integrators for constrained multibody systems. IMPACT Comp. Sci. Eng., No. 3, pp. 213-234, 1991.
- [Kaps79] Kaps P. and Rentrop P.: Generalized Runge-Kutta Methods of Order Four with Step-size Control for Stiff Ordinary Differential Equations. Numer. Math. 33, pp. 55-68, 1979.
- [Otte91] Otter M. and Gaus N. ANDECS-DSSIM: Modular Dynamic Simulation with Database Integration. User's Guide, Version 2.1. Technical Report TR R51--91, DLR, Institut für Robotik und Systemdynamik, June 1991.
- [Petz82] Petzold L.R.: A description of DASSL: A differential/algebraic system solver. Proc. 10th IMACS World Congress, Montreal, August 8-13, 1982.
- [Sham75] Shampine L.F. and Gordon M.K.: Computer Solution of Ordinary Differential Equations. Freeman, San Francisco, 1975.
- [Sham80] Shampine L.F. and Watts H.A.: DEPAC -- Design of a User Oriented Package of ODE Solvers. Sandia National Laboratories, Albuquerque, New Mexiko, SAND79--2374, 1980.

OTHER SIMULATION ENVIRONMENTS

Other simulation environments

Using the Dymola-Simulink interface

Before starting make sure to have a working Matlab/mex configuration by trying to compile and link an example Mex file, e.g.: matlab\extern\examples\mex\yprime.c

Please include

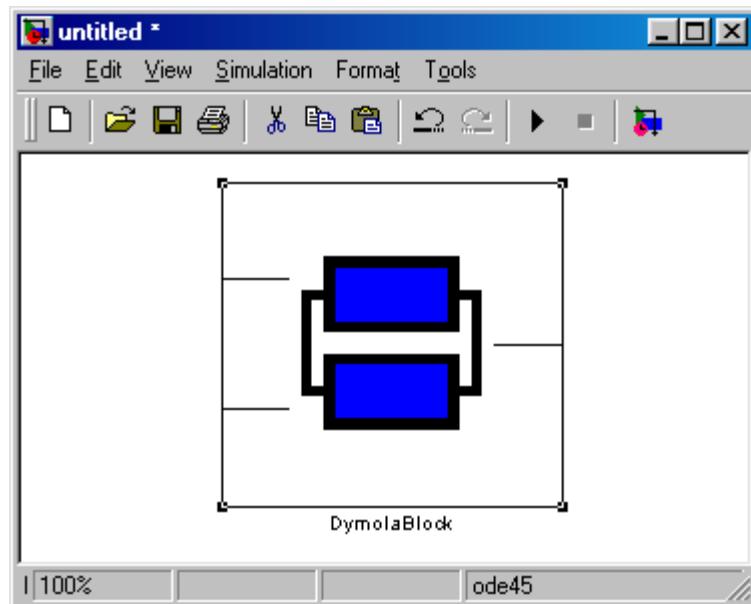
- dymola\mfiles
- dymola\mfiles\traj

in the Matlab path. These must be present every time you want to use blocks constructed with Dymola, a good idea is to do it once and then store the path in Matlab.

Graphical interface between Simulink and Dymola

Dymola has a new interface to Simulink. You find it in Simulink's library browser as Dymola Block/DymolaBlock (if it does not appear you have probably not included dymola\mfiles and dymola\mfiles\traj in your Matlab-path or is using a version prior to Matlab 5.3/Simulink 3). You click once to open the library and you then drag the DymolaBlock to other models.

Dymola block before compiling.



The DymolaBlock block, with the Dymola logo, represents the Modelica model. It can be connected to other Simulink blocks, and also to other DymolaBlocks. The lines at the right and left of the block will be replaced by ports after compiling the model.

The DymolaBlock is a shield around a S-function MEX block, i.e. the interface to the C-code generated by Dymola for the Modelica model. Double-clicking on the DymolaBlock open a form where you can set the name of the model and optionally the file name.

A simple way of setting the model name is to start Dymola and double-click on the correct model and then press “Select from Dymola>”. This automatically gives the model-name.

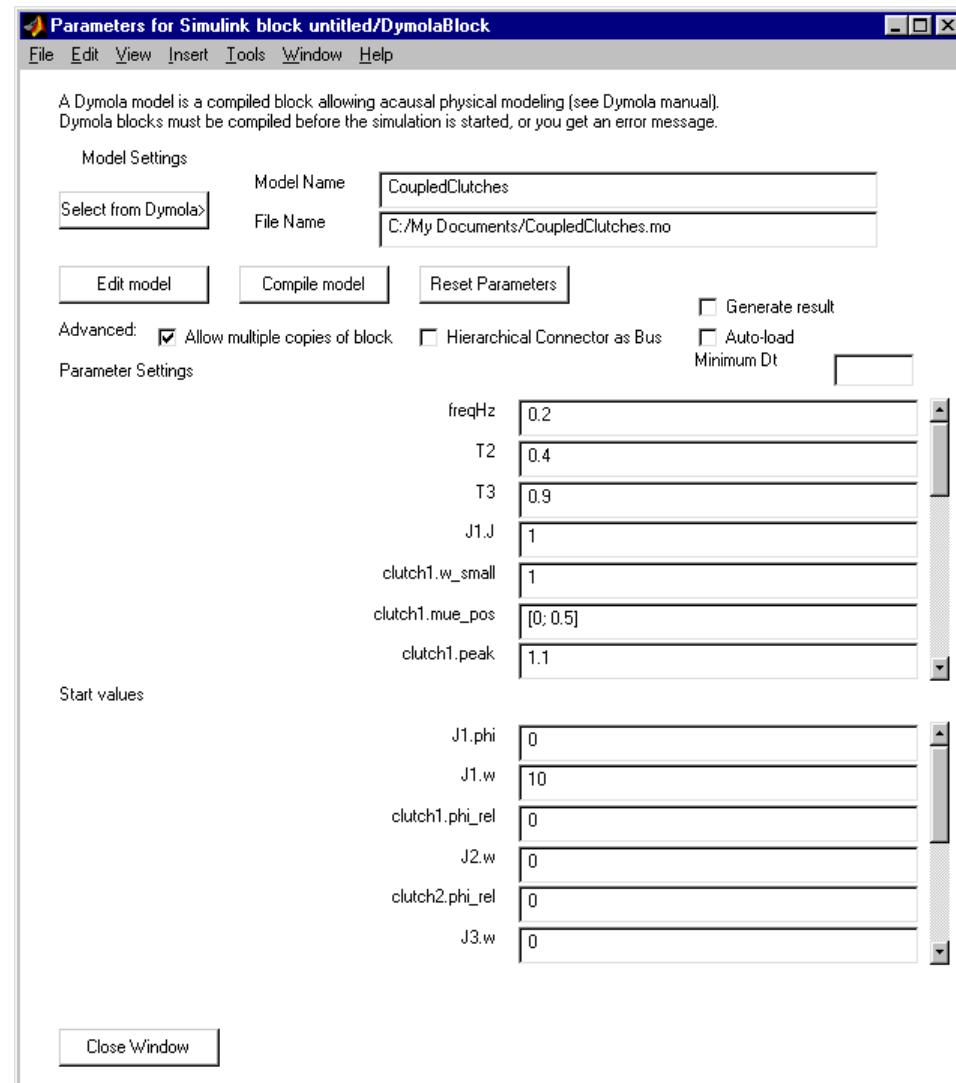
Note that the model name may contain a dotted name, e.g. Modelica.Blocks.Sources.ExpSine and the file name is optional, but is used both to find the model and to ensure that the Simulink model is up to date.

When associating a Modelica model the first time or when editing the Modelica model, click on “Edit Model” (after selecting the correct current directory at the Matlab prompt). This launches Dymola and cd's to the correct directory in Dymola and opens a window with the model. Edit the model in Dymola and verify it by making a test simulation in Dymola. After that click on “Compile Model”.

Ignore the check-boxes “Allow multiple copies of block” and “Hierarchical Connectors as Buses” (only available in Matlab 6.1 and later), and the boxes for results for now.

The form is now rebuilt as shown with parameters and start values for states (depending on the model). If the numbers of parameters or states if sufficiently large a scrollbar is added.

**Model dialog for a
Dymola model in
Simulink.**

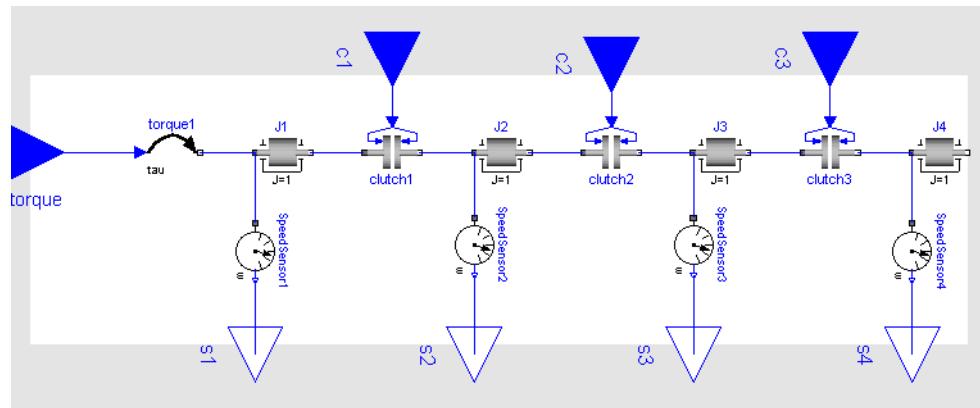


It is now possible to change parameters and initial values by entering their values in the new fields. Note that for array variables it is not possible to change the size of the array.

The parameter settings are kept even if you press 'Compile' (assuming they match). To reset parameters and start-values to the default in Dymola press 'Reset Parameters'.

In order to make the Dymola model useful as block in Simulink you need external inputs to the Dymola block and external outputs from the Dymola block. You change this in your model in Dymola, and it can in either be accomplished by declaring variables as input or output in the top-level model or graphically by adding input (filled) and output (non-filled) connectors, e.g. from Modelica.Blocks.Interfaces.

**Dymola model for use
in Simulink with
external inputs and
outputs.**



As an example consider a rewrite of the coupled clutches example where each source in Modelica has been replaced by an input connector, and speed-sensors have been added and connected to output connector.

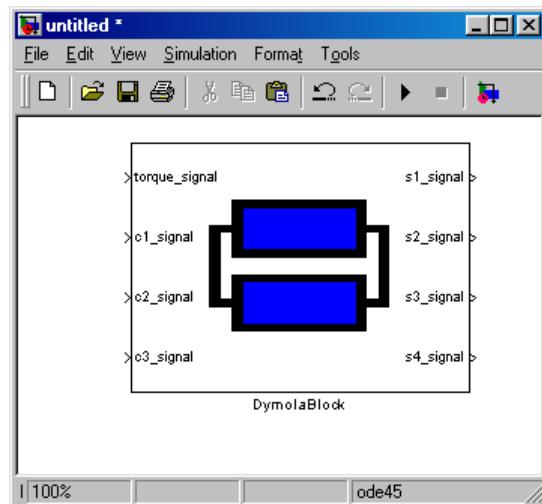
You can then close this parameter screen and observe that input and output ports been added to the DymolaBlock, corresponding to the inputs and outputs connectors in Dymola. The name of the ports are the names of the Modelica input/output variables. The order of the graphical inputs and outputs correspond to the order in which they were added to the model.

Simulation is made in the usual way after connecting the input and output ports to other Simulink blocks, and setting suitable tolerances and integration algorithm in Simulink.

If you want to recreate the example above you should also be aware that the parametrizations differ between Modelica standard library models and Simulink blocks, e.g. the frequency of Simulink's Sine-block is measured in rad/s, which is commonly known as angular frequency and should thus be 2π times the frequency in the corresponding source in Modelica.

If you have not compiled all Dymola models used in the model you will get an error message. In that case, double click on those blocks to bring up the forms and click on "Compile Model".

Dymola block after compiling. With named ports.



Hierarchical Connectors as Buses

Sometimes it is desirable to use buses for signals in Simulink. Provided you have Matlab 6.1 or later, it is possible to map hierarchical connectors in Modelica to buses in Simulink. This is accomplished by checking “hierarchical Connectors as Buses” and recompiling.

In order to get an understanding of the structure an easy way is to construct hierarchical output connectors in Modelica and examine the corresponding bus-signals in Simulink. The rules for the buses are:

- All hierarchical input/outputs components are transformed into buses, except the signal-element of block-connectors.
- Connector members of simple types (Real/Integer/Boolean) are mapped to real signals in Simulink with the same name as in Dymola. Non-scalar elements are mapped into non-scalar signals in Simulink.
- Connector members that are scalar records/connectors are mapped to buses in Simulink with the same name as in Dymola.
- For connector members that are arrays of records/connectors, each element is mapped to buses in Simulink with the same name as in Dymola (except that Simulink requires that any “,” for matrix elements is replaced by “;”).

The mapping is fully hierarchical (i.e. connectors can contain sub-connectors that contain sub-connectors etc), and applied to all input/outputs.

Result files

Sometimes it is useful to generate result files in Dymola’s standard format when running in Matlab/Simulink, e.g. to:

- Animate 3d-objects
- Investigate intermediate variables in the model
- Generate an input file for further simulation studies in Dymola (this requires an extra step: save the generated result file (Context menu in plot-selector in Dymola) as ‘dsu.txt’).

All of this is accomplished by selecting “Generate Result”.

If you also want the result file to be automatically loaded in Dymola select “Auto-load”.

If the result files become very large it might be useful to reduce the size by introducing a “Minimum Dt”, which is a minimal output interval length, e.g. 0.1 (similar, but not identical to “Interval length” in Dymola). The choice “Minimum Dt” requires Matlab 6.1 or later.

Any modifications of these choices require a new compilation.

Implementation notes

The DymolaBlock only works from Matlab 5.3/Simulink 3.0.

Temporary files for compilation are created in \dymola\tmp and reused from one compilation to the next. When changing mex-compiler in Matlab you have to remove the contents of this directory.

The DymolaBlock rely on a number of callbacks and tags in order to allow copying, renaming, etc. This allows us to store settings for model name, parameters and initial values in Simulink’s mdl-format. However, manually editing mdl-files containing DymolaBlocks is not supported, neither is changing the tag of the DymolaBlock or any of the callbacks associated with the DymolaBlock.

If “Allow multiple copies of block” is unchecked you should **not copy** the block. Unchecking it **should only** be done if you have dSPACE multi-processor hardware, since its build command internally makes copies of the model that should share the original dll-name. Normally copying of DymolaBlock takes special care to create unique identifications for the block such that the copies use different dll’s even if they are multiple copies of the same Dymola model. This is necessary in order to allow you to connect two copies of the same model to each other, but does not work together with dSPACE’s build command.

DymolaBlock is found in DymolaBlockMaster.mdl, and you can also get DymolaBlock by running DymolaBlockMaster. In this case you must copy DymolaBlock to another model.

Simulation in Matlab

There is also a low level M-file, Dymsim.m allows you to use Modelica models from Dymola as building blocks in Simulink, and simulate them from within Matlab and Simulink.

To use dymsim.m:

- Select File/Translate Model in the Modelica model’s window.
- Start Matlab and change to the same directory.

- Make sure that Dymola's mfiles are in the Matlab-path.
- Run the command 'dymsim' in Matlab; this compiles the model and simulates it.
- To simulate it again without recompiling use simulate on the Matlab command line.

In Matlab you can change parameters, p, and initial values, x0, as well as StartTime and StopTime.

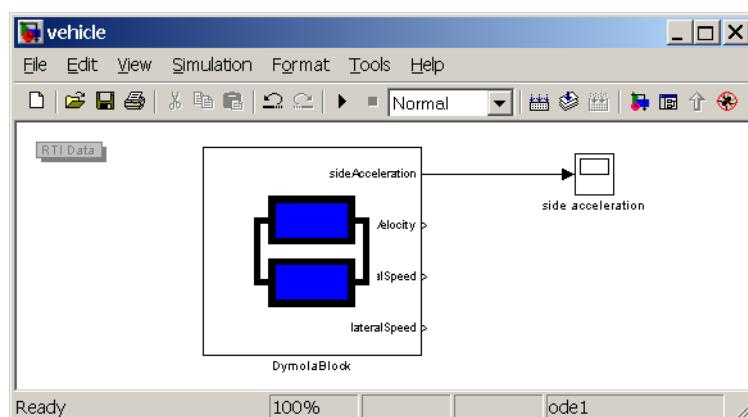
Real-time simulation

Dymola provides support for real-time simulation of DymolaBlock on several platforms: dSPACE, Real-Time Workshop and RT-LAB.

dSPACE systems

The DymolaBlock allows you to run your Dymola models on dSPACE systems. The Simulink model is created as described in the dSPACE documentation. In many cases offline simulation in Simulink can be used to verify correct operation.

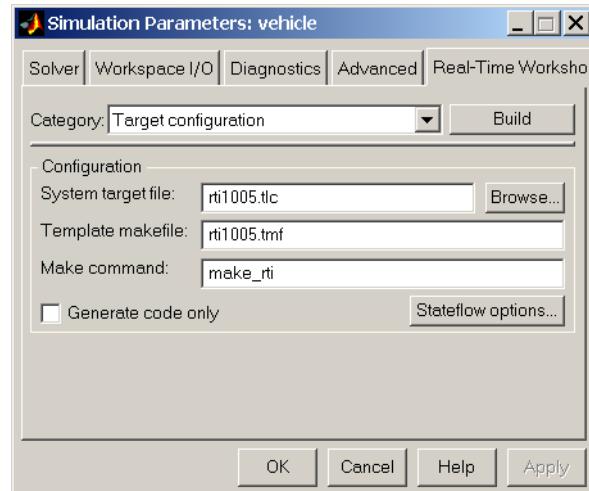
The model configured to simulate with RTI.



There can be only one DymolaBlock when generating code for a multi-processor dSPACE system (RTI-MP). If you have checked “Allow multiple copies of block” you will get a warning about the block not being up-to-date. In the DymolaBlock model dialog in Simulink uncheck “Allow multiple copies of block”. See “Implementation notes” on page 224 for reasons.

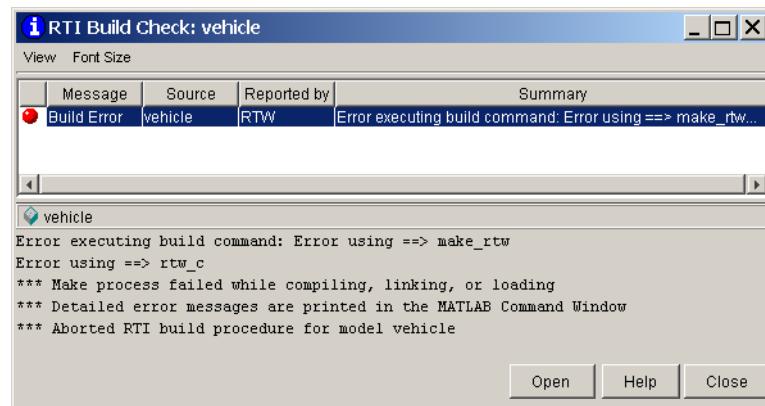
The appropriate target configuration must be selected in Tools/Real-Time Workshop/Options.

**Target configuration
for DS1005.**



In order to compile the model for dSPACE you also have to modify the makefiles. The recommended practice for dSPACE is to change the <model>_usr.mk. The easiest way to create a <model>_usr.mk is to build the application according to the instructions in the dSPACE documentation, which will fail because additional files from the Dymola distribution are unavailable.

**Initial failed build (to
create makefile).**



After modifying the generated <model>_usr.mk, the build process should succeed. The vital settings are additional source files and directories. For a DS1005 system the changes are:

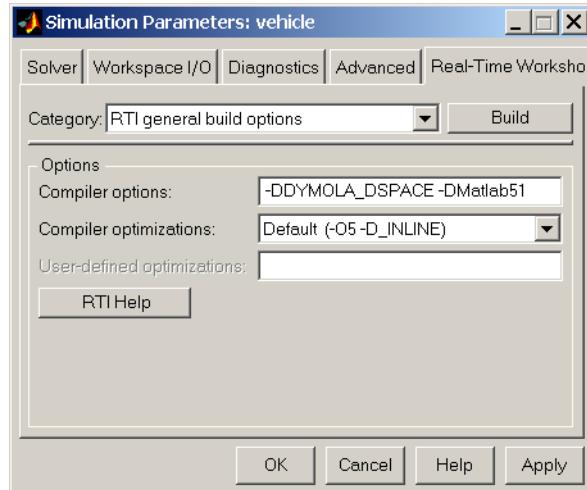
```
# Additional C source files to be compiled.  
USER_SRCS = dymc.c dymf.c amat.c usertab.c  
  
# Directories for C source files.  
USER_SRCS_DIR = \dymola\source  
  
# Path names for user include files.  
USER_INCLUDES_PATH = \dymola\source
```

For dSPACE release 3.4 one more option must be set in the user makefile. For your convenience \$DYMOLA/mfiles/tmf/appl_usr.mk contains an example for dSPACE release 3.4.

```
# Additional compiler definitions.  
# The MATLAB option defines "MATLAB 5.1 or later".  
OPTS = -O5 -D_INLINE -DDYMOLA_DSPACE -DMatlab51
```

For dSPACE release 4.0 compiler options are set in the options dialog of Real-Time Workshop. The following definitions are needed to compile Dymola models on dSPACE.

Compiler options for dSPACE RTI.



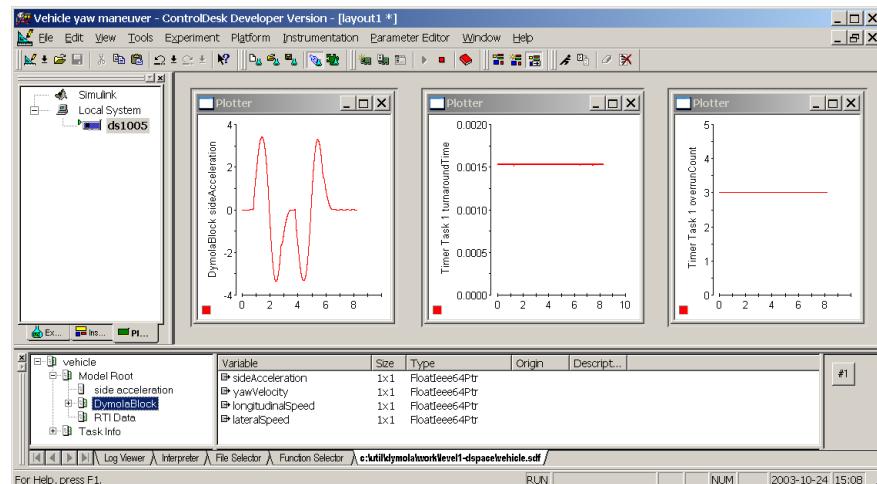
This completes the setup required to build for simulation on dSPACE. The remaining setup issue is handling of overruns, see page 228.

Simulation

After completion of the build step, the model has been downloaded to the dSPACE hardware. Starting the simulation runs in realtime, but signals are plotted in the Simulink scopes.

A more extensive user interface can be built using dSPACE ControlDesk, which provides a range of tools for both input and output. ControlDesk also allows plotting of important system variables, such as, the models turnaround time (model calculation time plus overhead) and the number of overruns.

**Simulation using
dSPACE ControlDesk.**



Overrun

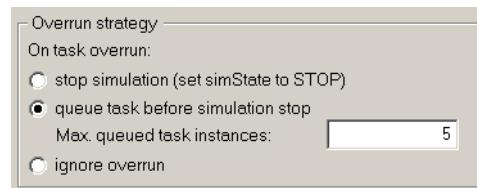
An overrun situation may arise for several reasons, as documented in the dSPACE manual. Two additional cases are worth pointing out:

- Event iteration in a model generated by Dymola may require additional CPU resources.
- When the model is downloaded to the dSPACE hardware, the first time step must also perform the initialization of global data and the C runtime library.

In these cases it is usually helpful to plot the turnaround time, a signal provided by the dSPACE system. The turnaround time will show when additional resource are needed. In many cases occasional overrun situations are harmless.

The RTI setup enables you to specify a number of instances of a task that may be queued before the realtime kernel issues an overrun error. This is done in the Real-Time Workshop options dialog by selecting the category “RTI simulation options” and then pressing “Task Configuration”. The appropriate setting can be determined by trial-and-error.

**RTI task configuration
(overrun strategy).**



xPC and Real-Time Workshop

Using Real-Time Workshop, and in particular xPC, it is possible to use the DymolaBlock for Hardware in the loop simulation (HILS). The setup is a bit different from dSPACE in order to work-around some problems in the compilation procedures in Simulink. The positive side is that is easy to handle, all you have to do is to append

```
"USER_INCLUDES=-Ic:\dymola\source"
```

to the make line in the real-time workshop build setup. Note the quotes and adapt the dymola path as necessary.

This works well with e.g. xPC (both Watcom and Visual Studio compiler) under Matlab 6. However, it has not been tested with all Real-Time Workshop targets, and thus there might be some potential problems for some targets.

Real-time simulation using RT-LAB

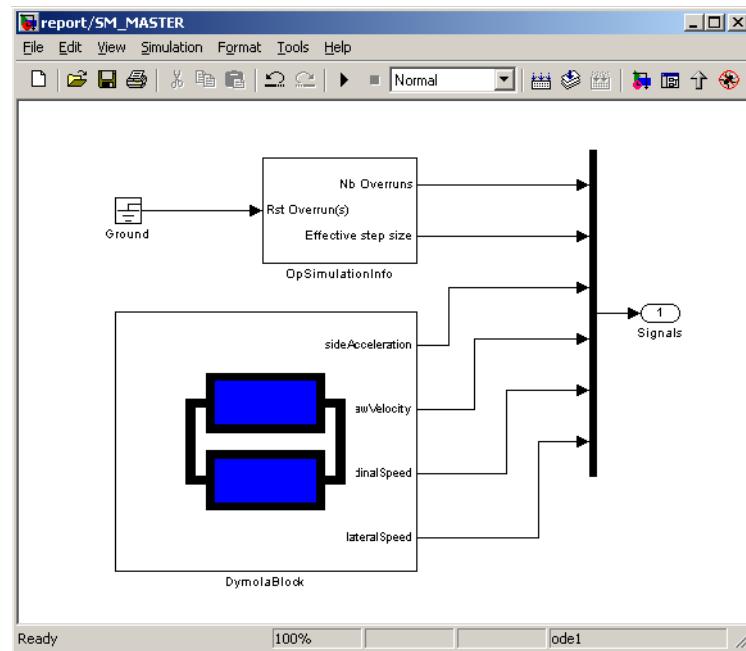
The RT-LAB development environment is based on Simulink and Real-Time Workshop. The generated code is executed on one or more target systems running the QNX real-time operating system. Consequently, the Modelica model developed in Dymola is exported to Simulink, using a pre-defined DymolaBlock (S-function block). When the model has been translated by Dymola and compiled, the block is automatically customized according to the model's input, output and top-level parameters.

Model re-organization

The Simulink model must then be slightly modified to fit into the RT-LAB framework. Blocks that should run in real-time must be grouped according to computational node, and input and output blocks (e.g., scopes) are grouped into a “console” block. This procedure is described in more detail in the RT-LAB documentation.

The figure below shows a “master” block, which is the primary block of the real-time computation. In addition to the DymolaBlock, which contains the entire plant model, an OpSimulationInfo block has been added to make signals for effective step size (computation time + communication time) and a counter of overruns. In this example all the signals are multiplexed into a single channel to the console.

An RT-LAB master block.

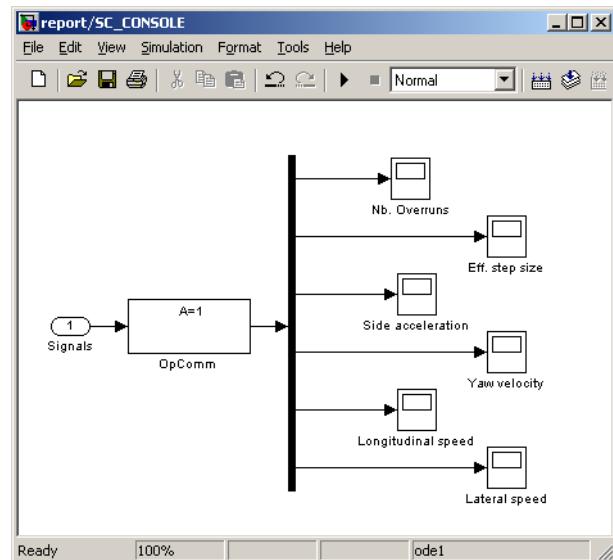


In a cluster with several computational nodes, one node is designated as master, and other real-time nodes are designated as slaves. Structurally the slave blocks are similar to the master block.

The controlling workstation, which does not run in real-time, is represented by the console block, shown below. In this case, several scopes are used to watch the output signals. Input blocks could also have been used. A special OpComm block must be inserted as well, which represents data communication from the real-time computers

Even after these manipulations, the Simulink model can be simulated in offline mode. This helps verify that the model is correct before going to real-time simulation.

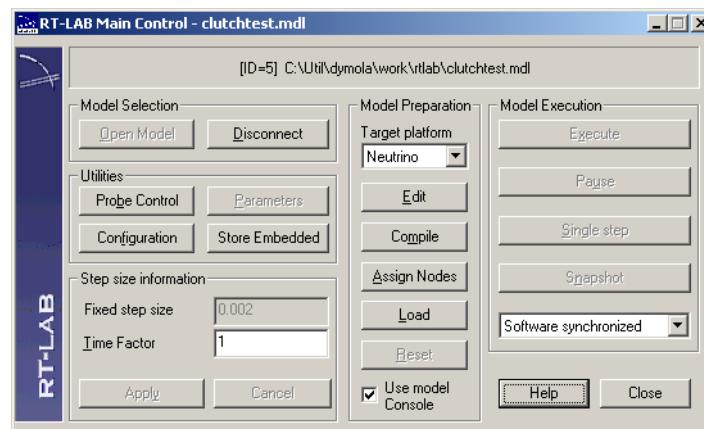
An RT-LAB console block.



RT-LAB setup

The model is now ready for processing by RT-LAB. Most of the work is done through the RT-LAB Main Control window. The figure below shows the main control window after the Simulink model has been opened.

The RT-LAB Main Control window.



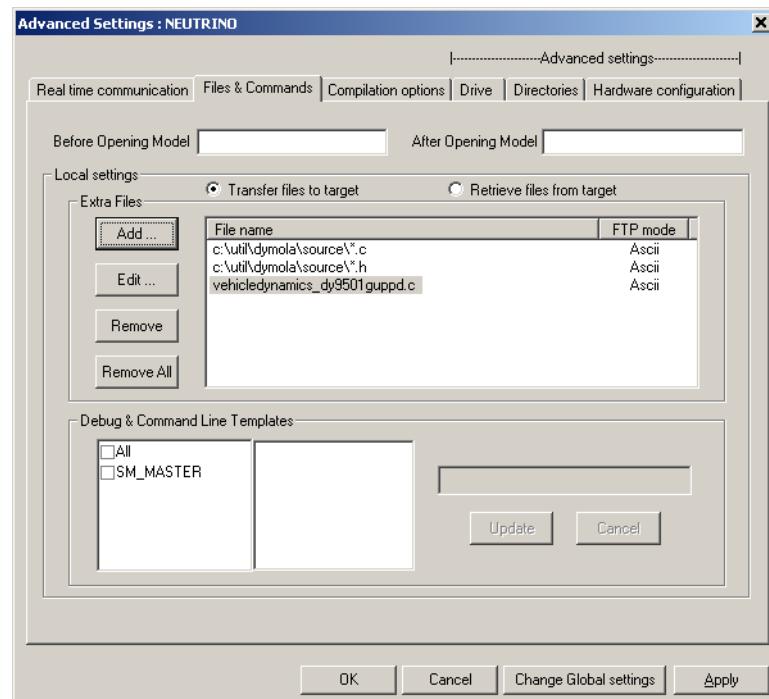
The most important operations are located at the center of the RT-LAB Main Control. The Edit button opens the model in Simulink, should any changes be required. The Compile button processes the Simulink model and builds the target code with the help of Real-Time Workshop. The Load button initializes the QNX target(s) with the executable code and establishes communication channels. The Execute button is used to run the model in real time. As a side effect, the Execute button starts Simulink with the console block active and receiving data from the real-time nodes.

During model separation, RT-LAB creates new MDL files with new names and in a new directory. These new files cannot be used to edit or compile the DymolaBlock, press the Edit button in RT-LAB Main Control to edit the original model.

File transfer settings

Configuration settings are needed to set up processing and compilation of the model. One important step is to setup the transfer of files from the console to the target compiling the model. This is done by pressing the Configuration button and then selecting Advanced/Files&Commands.

RT-LAB file transfer settings.



The first two lines describe required files from the Dymola distribution; note the use of wildcards. Also, compilation of the DymolaBlock creates a model source file with a machine-generated name (this file can be found using Windows Explorer, in this case it was vehicledynamics_dy9501guppd.c) and the filename must be recorded.

Compilation options

Experiments have shown that for large models moderate optimization (-O1) yields faster codes than maximum optimization (-O2) on the QNX platform. Under the “Compilation options” tab, the field for “User compiler option” should contain the optimization option -O1.

DDE communication

Dymola DDE commands

Dymola accepts commands through a Windows DDE connection. The DDE connection should be opened with an application name of "dymola"; the topic name is not used.

The following commands are recognized:

- All Dymola commands as entered in the command input or read from a script file.
- These commands in the graphical editor:

DDE Command	Corresponding command in graphical editor
open filename	File/Open
clear	File/Clear All

All Modelica script features are also supported.

Explorer file type associations

It is possible to associate new commands with the ".mo" file type. These commands are accessible through the right-mouse button menu in Explorer. We suggest this setup:

1. Start Windows Explorer and select View/Options. Select the File Type tab.
2. Click on "New Type" to create a new association. Enter the following general information:
 - a. Change icon: browse for the Dymola program, typically C:\Dymola\bin\dymola.exe.
 - b. Description of type: Modelica model
 - c. Associated extension: .mo
 - d. Content type (MIME): text/plain
 - e. Default extension for content type: .txt
3. Associate new menu commands with the file type. Click on "New...", then enter:
 - a. Action: Open
 - b. Application used to perform action: browse for the Dymola program, typically C:\Dymola\bin\dymola.exe.
 - c. Use DDE: check
 - d. DDE message: open %1
 - e. Application: dymola

Leave the remaining fields empty.

In a similar way you can associate several commands with the extensions .mo and .mos:

Action	Application to perform action	DDE message	File extension
Edit	C:\windows\notepad.exe %1 or C:\winnt\notepad.exe %1		all
Open	C:\Dymola\bin\dymola.exe	open %1	*.mo
Run	C:\Dymola\bin\dymola.exe	@%1	*.mos

Dymosim Windows application

Dymosim can be compiled as a Windows application using an additional library with Windows interface routines. The advantage of running Dymosim as Windows application (instead of a console application) is that Dymosim can act as a realtime DDE server.

Without any action from the user, Dymosim will be built and executed in the usual way, i.e., without realtime synchronization. The noticeable differences are that the current simulation time is shown in the minimized Dymosim window, and that the dymosim application has a Stop command.

Realtime simulation

If the environment variable DYMOSIMREALTIME is defined, Dymosim will start in realtime mode, where the simulation time is synchronized with real (or wall-clock) time. The time display will also include the time the simulation is delayed at each accepted output point, in order to keep synchronized with real time. A negative value indicates that the simulation is faster than real time, i.e., that there is spare time for additional computations.

Dymosim DDE server

Dymosim compiled as a Windows application will act as a DDE server, allowing some other application to retrieve data values or set parameters. Dymosim must be started before it can accept DDE operations; MATLAB's ddeinit will not start Dymosim automatically, for example.

A DDE connection is established by sending a WM_DDE_INITIATE messages with the application name "dymosim" (any topic can be used).

MATLAB example: channel=ddeinit('dymosim','xxx')

After a Stop command or at the end of simulation, Dymosim will send a WM_DDE_TERMINATE message.

MATLAB example: ddeterm(channel)

Note that all transactions between the Dymosim DDE server and the DDE client are logged in the Dymosim window. They can also be logged on file.

Simulator commands

The following commands can be sent to Dymosim using WM_DDE_EXECUTE messages:

"run"	Start simulation (if simulation is not started automatically), or resume simulation after a pause command.
"stop"	Stop simulation.
"pause"	Pause simulation. The simulation is temporarily halted until a "run" command is given. Note that DDE requests are handled while pausing.
"logon"	Enables logging to file, if logging is off.
"logoff"	Disables logging to file, if logging is on.

MATLAB example: `ddeexec(channel, 'run')`

If the environment variable DYMOsimLOGDDE is defined, a file logging all DDE communication to/from the simulator is created when the program starts.

Setting parameters

Parameters may be set by sending a WM_DDE_POKE message with the name of the parameter and its new value (the string representation of a number).

There are four special variables:

realtime_	Set to "1" to enable realtime mode, or to "0" to disable realtime mode.
tscale_	The scale factor between simulated time and real time. In realtime mode the simulator will maintain the relationship $\text{real-time} = \text{tscale}_ * \text{simulated-time}$
abstol_	Absolute tolerance for hot linked variables (default 0).
reltol_	Relative tolerance for hot linked variables (default 0).

MATLAB example: `ddepoke(channel, 'tscale_', 2.0)`

Requesting variables

The value of variables at the last accepted output point are available by sending a WM_DDE_REQUEST message with the name of the variable. Dymosim will then return a message with a current value of the variable (the string representation of the number), or zero if no such variable exists.

It is also possible to request the value of the following special variables:

delayed_	Returns the time the simulation was delayed at the last accepted output point.
status_	Returns the state of the DDE server. The state is composed of the following parts: 1 Simulation started (running). 10 Simulation paused. 100 Current simulation time 0.
time_	Returns the current simulation time.
tscale_	Returns the current simulation time scale factor.

abstol_	Absolute tolerance for hot linked variables.
reldtol_	Relative tolerance for hot linked variables.

MATLAB example: `ddereq(channel, 'time_')`

Hot linking variables

Variables can be "hot linked" using message WM_DDE_ADVISE. The linked variables will be sent to the client at output points when a significant change has occurred.

A significant change of a variable is determined from absolute and relative tolerances (set-table by the DDE client) as follows (x_0 = value last sent to client, x = current value):

```
absmax = max (abs (x0) , abs (x)) ;
absdiff = abs (x - x0) ;

changed = absmax < 1 ? absdiff > abstol_
: absdiff/absmax > reltol_;
```

The variable is sent to the client when "changed" is true. The variable is also sent at the first output point following the hot-link operation.

MATLAB example: `ddeadv(channel, 'time_', 't=[t x];', 'x')`

Dymosim counts the number of links to a variable, and any corresponding "unlink" messages. Updates for the variable are sent while the link count is greater than zero.

Extended graphical user interface

If the environment variable DYMOSIMGUI is defined, Dymosim will start with a slightly extended user interface. The Operations menu then also contains commands to manually Run/Pause the simulation or toggle realtime mode. The Toggle Log command enables logging of all DDE communication on the file ddelog.txt.

Another difference is that simulation does not start automatically when the program is executed; instead the user must give a Run command.

APPENDIX — MODELICA

Appendix — Modelica

Modelica syntax specification

The Modelica syntax specification is taken from the Modelica Language Specification.

Lexical conventions

The following syntactic meta symbols are used (extended BNF):

[] optional
{ } repeat zero or more times

The following lexical units are defined:

```
IDENT = NONDIGIT { DIGIT | NONDIGIT }

NONDIGIT = "_" | letters "a" to "z" | letters "A" to "Z"

STRING = "" { S-CHAR | S-ESCAPE } ""

S-CHAR = any member of the source character set except double-quote "", and
backslash "\"

S-ESCAPE = "\' " | "\" " | "\?" | "\\" |
           "\a" | "\b" | "\f" | "\n" | "\r" | "\t" | "\v"

DIGIT = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

UNSIGNED_INTEGER = DIGIT { DIGIT }
```

```

UNSIGNED_NUMBER = UNSIGNED_INTEGER [ "." [
UNSIGNED_INTEGER ] ]
[ ( e | E ) [ "+" | "-" ] UNSIGNED_INTEGER ]

```

Note: string constant concatenation "a" "b" becoming "ab" (as in C) is replaced by the "+" operator in Modelica.

Modelica uses the same comment syntax as C++ and Java. Inside a comment, the sequence <HTML> </HTML> indicates HTML code which may be used by tools to facilitate model documentation.

Keywords and built-in operators of the Modelica language are written in bold face. Keywords are reserved words and may not be used as identifiers.

Grammar

Model definition

```

model_definition:
{ [ final ] class_definition ";" }
```

Class definition

```

class_definition :
[ partial ]
( class | model | record | block | connector | type |
package | function )
IDENT class_specifier

class_specifier :
string_comment composition end IDENT
| "=" name [ array_subscripts ] [ class_modification ] comment

composition :
element_list
{ public element_list |
protected element_list |
equation_clause |
algorithm_clause
}
[ external [ language_specification ]
[ external_function_call ] ";" [ annotation ";" ] ]

language_specification :
STRING

external_function_call :
[ component_reference "=" ]
IDENT "(" [ expression { "," expression } ] ")"
```

```

element_list :
{ element ";" | annotation ";" }

element :
[ inner | outer ]
( [ replaceable ] class_definition | extends_clause
| [replaceable] component_clause )

Extends
extends_clause :
  extends name [ class_modification ]

Component clause
component_clause:
  type_prefix typeSpecifier [ array_subscripts ] component_list

type_prefix :
[ flow ]
[ discrete | nondiscrete | parameter | constant ] [input | output]

typeSpecifier :
  name

component_list :
  component_declaration { "," component_declaration }

component_declaration :
  declaration comment

declaration :
  IDENT [ array_subscripts ] [ modification ]

Modification
modification :
  class_modification [ "=" expression ]
  | "=" expression
  | ":=" expression

class_modification :
  "(" { argument_list } ")"

argument_list :
  argument { "," argument }

```

```

argument :
    element_modification
| element_redeclaration

element_modification :
    [ final ] component_reference modification

element_redeclaration :
    redeclare
    ([ replaceable ] class_definition | extends_clause |
[replaceable] component_clause1 )

component_clause1 :
    type_prefix type_specifier component_declaration

Equations

equation_clause :
    equation { equation ";" | annotation ";" }

algorithm_clause :
    algorithm { algorithm ";" | annotation ";" }

equation :
    ( simple_expression "=" expression
    | conditional_equation_e
    | for_clause_e
    | when_clause_e
    | connect_clause
    | assert_clause )
comment

algorithm :
    ( component_reference ( ":=" expression | function_call )
    | "(" expression_list ")" ":=" function_call
    | conditional_equation_a
    | for_clause_a
    | while_clause
    | when_clause_a
    | assert_clause )
comment

conditional_equation_e :
    if expression then
    { equation ";" }

```

```

{ elseif expression then
  { equation ";" }
}
[ else
  { equation ";" }
]
end if

conditional_equation_a :
if expression then
  { algorithm ";" }
{ elseif expression then
  { algorithm ";" }
}
[ else
  { algorithm ";" }
]
end if

for_clause_e :
for IDENT in expression loop
  { equation ";" }
end for

for_clause_a :
for IDENT in expression loop
  { algorithm ";" }
end for

while_clause :
while expression loop
  { algorithm ";" }
end while

when_clause_e :
when expression then
  { equation ";" }
end when

when_clause_a :
when expression then
  { algorithm ";" }
end when

connect_clause :

```

```

connect "(" connector_ref "," connector_ref ")"
connector_ref :
    IDENT [ array_subscripts ] [ "." IDENT [ array_subscripts ] ]

assert_clause :
    assert "(" expression "," STRING { "+" STRING } ")"
    terminate "(" STRING { "+" STRING } ")"

Expressions
expression :
    simple_expression
    | if expression then expression else expression

simple_expression :
    logical_expression [ ":" logical_expression [ ":" logical_expression ] ]

logical_expression :
    logical_term { or logical_term }

logical_term :
    logical_factor { and logical_factor }

logical_factor :
    [ not ] relation

relation :
    arithmetic_expression [ rel_op arithmetic_expression ]

rel_op :
    "<" | "<=" | ">" | ">=" | "==" | "<>"

arithmetic_expression :
    [ add_op ] term { add_op term }

add_op :
    "+" | "-"

term :
    factor { mul_op factor }

mul_op :
    "*" | "/"

```

```

factor :
    primary [ "^" primary ]

primary :
    UNSIGNED_NUMBER
    | STRING
    | false
    | true
    | component_reference [ function_call ]
    | "(" expression_list ")"
    | "[" expression_list { ";" expression_list } "]"
    | "{" expression_list "}"

name :
    IDENT [ "." name ]

component_reference :
    IDENT [ array_subscripts ] [ "." component_reference ]

function_call :
    "(" function_arguments ")"

function_arguments :
    expression_list
    | named_arguments

named_arguments: [named_argument { "," named_argument }]

named_argument: IDENT "=" expression

expression_list :
    expression { "," expression }

array_subscripts :
    "[" subscript { "," subscript } "]"

subscript :
    ":" | expression

comment :
    string_comment [ annotation ]

string_comment :
    [ STRING { "+" STRING } ]

```

```
annotation :  
annotation class_modification
```

APPENDIX — ADVANCED MODELICA

Appendix — Advanced Modelica

Declaring functions

In Modelica it is possible to define and use functions, and the functions be used in equations of a model, in defining parameters, and called interactively from the command line.

Functions inherently reduce the possibility for symbolic manipulations, and should thus not be used unless necessary. In particular index-reduction requires that all functions are differentiated, which cannot be done in a straightforward way without help. This “help” is provided in form of an annotation giving the derivative of the function.

The functions themselves can either be written in Modelica or as external functions in C or FORTRAN 77. In most cases the external function is provided in a binary library with a corresponding header defining the function.

User-defined derivatives

In order to reduce the index it is necessary to differentiate arbitrary expressions with respect to time. This derivative must be very accurate in order to not introduce unnecessary numerical errors. The functions are seldom simple scalar functions from one scalar input, but in-

stead have vectors and/or records as input and several inputs and/or outputs. Often second order differentials are also needed.

For solving non-linear system of equations derivatives can also increase performance, by allowing us to compute an analytical Jacobian.

To allow this we have an annotation that defines the derivative of a one function, and Dymola can use this both to reduce the index and to compute Jacobians for non-linear system of equations.

This is defined as an annotation declaring the derivative-function for the given function, and it can be given both for functions written in Modelica as well as for external functions.

How to declare a derivative

The following define how to declaring the derivative to a function, and finally how to verify that the derivative is consistent with the function. It is strongly influenced by forward mode automatic differentiation and well-suited for differentiation with respect to one variable, as in index-reduction.

It can furthermore be used to efficiently compute all interesting derivatives in a straightforward way as will be explained later.

A function declaration can have an annotation derivative specifying the derivative function with an optional order-attribute indicating the order of the derivative (default 1), e.g.:

```
function f0 annotation(derivative=f1); end f0;
function f1 annotation(derivative(order=2)=f2); end f1;
function f2 end f2;
```

It is also necessary how to write the derivative function for a given function, this is described in a procedural form below, and with examples that make it more clear.

The lookup for the derivative annotation follow the normal lookup rules of Modelica.

First order derivative

The inputs to the derivative function of order 1 are constructed as follows:

- First are all inputs to the original function, and after all them we will in order append one derivative for each input containing reals.
- The outputs are constructed by starting with an empty list and then in order appending one derivative for each output containing reals.

As an example consider the following:

```
function foo0
    input Real x;
    input Boolean linear;
    input Real z;
    output Real y;
algorithm
```

```

if linear then
    y:=z+x;
else
    y:=z+sin(x);
end if;
annotation(derivative=foo1);
end foo0;

function foo1
    input Real x;
    input Boolean linear;
    input Real z;
    input Real der_x;
    input Real der_z;
    output Real der_y;
    annotation(derivative(order=2)=foo2);
algorithm
    der_y:=der_z+(if linear then der_x else cos(x)*der_x);
end foo1;

```

This implies that given the following equation

$$y(t) = \text{foo0}(x(t), b, z(t))$$

we know that

$$\dot{y}(t) = \text{foo1}(x(t), b, z(t), \dot{x}(t), \dot{z}(t))$$

A more complex example involving records and matrices is

```

record R
    Real M[2,2];
    Real x[2];
end R;

function recordFunction
    input R x;
    output Real y[2];
algorithm
    y:=x.M*x.x;
    annotation(derivative=recordFunction_d);
end recordFunction;

function recordFunction_d
    input R x;
    input R der_x;
    output Real der_y[2];
algorithm
    der_y:=x.M*der_x.x+der_x.M*x.x;
    // Since (A*B)'=A'*B+A*B' for matrices.
end recordFunction_d;

```

Thus if

```
y(t)=recordFunction(x(t));
```

we have

```
der(y(t))=recordFunction_d(x(t),der(x(t)));
```

Second and higher order derivatives

If the Modelica function is a nth derivative ($n \geq 1$) the derivative annotation indicates the $(n+1)$ th derivative, and order=n+1.

The input arguments are amended by the $(n+1)$ th derivatives, which are constructed in order from the nth order derivatives.

The output arguments are similar to the output argument for the nth derivative, but each output is one higher in derivative order.

Continue the example above with:

```
function foo1
  ...
  annotation(derivative(order=2)=foo2);
  ...
end foo1;

function foo2
  input Real x;
  input Boolean linear;
  input Real z;
  input Real der_x;
  input Real der_z;
  input Real der_2_x;
  input Real der_2_z;
  output Real der_2_y;
algorithm
  der_2_y:=der_2_z+(if linear then der_2_x else
    cos(x)*der_2_x-sin(x)*der_x^2);
end foo1;
```

This allows us to conclude that

$$\ddot{y}(t) = \text{foo2}(x(t), b, z(t), \dot{x}(t), \dot{z}(t), \ddot{x}(t), \ddot{z}(t))$$

Restrictions

An input or output to the function may be any predefined type (Real,Boolean,Integer and String) or a record, provided the record does not contain both reals and non-reals predefined types. Allowing mixed records would require that we automatically constructed a new record from the parts containing reals, which would be difficult to describe.

The function must have at least one input containing reals, since we must have something to take the derivative with respect to.

The output list of the derivative function may not be empty, since we otherwise have no derivative. This can occur if the function e.g. returns a boolean value.

Verifying Derivatives

In order to verify that a derivative is consistent with the function it is recommended to follow the following test-procedure. The basic idea is to compare the integral of the derivative with the original function.

Assume one has a model using foo0:

```
model B
  Real x;
equation
  x=foo0(time, false, -0.1*time);
end B;
```

and we want to verify that the derivative of foo0 is correct. We do that by extending B as follows:

```
model VerifyFirstDerivative
  extends B;
  Real y;
equation
  der(y)=der(x);
initial equation
  y=x;
end VerifyFirstDerivative;
```

That the derivative is correct can be verified by comparing x (which is computed directly) and y (which is computed as an integral of the derivative). By setting second derivatives equal one can verify the second derivative as well. Note that this procedure does not modify the original model and can therefore be used even when the input arguments to the function are given internally.

External functions

In addition to functions written in Modelica, Dymola also allows external functions written in ANSI/ISO C. For each external function it is necessary to declare a Modelica interface. This declaration provides the required information needed to call the function from a Modelica model, and in some simple cases provide argument conversion.

Existing libraries may require wrapper functions.

External functions are declared as Modelica functions, but with a body that defines the interface. The Modelica specification defines the details of how function arguments in Modelica are mapped to similar data types in C, and how values returned from the functions are mapped back to Modelica types. When the interface of the external function does not match the Modelica specification, a wrapper function must be written in C to perform the required conversions.

In most cases the external function is provided in a binary library with a corresponding header file declaring the function. In order to support this one can specify a header, overriding the usual definition of the function, and a library that will automatically be linked with.

Including external functions

In simple cases it is possible to translate the C code of the function with the model itself. The main advantage of this approach is that it does not require any additional effort to build a library. The disadvantage is that definitions in the implementation of the C function may interfere with the generated model code and cause the compilation to fail.

As an example we will use the following trivial function that returns the sum of two real numbers. Its implementation in ANSI/ISO C is called `add2.c`.

```
#ifndef ADD2_C
#define ADD2_C
double add2(double x, double y)
{
    return x + y;
}
#endif
```

This function requires a declaration that provides a mapping between Modelica and C, and also specifies the name of the file containing the implementation.

```
function add2 "Sum of two numbers"
    input Real x, y;
    output Real sum;
    external "C";
    annotation(Include="#include <add2.c>") ;
end add2;
```

The first two declarations define the input and output arguments and their types in the Modelica context. The `external` declaration identifies this as an external function. The types of the parameters in the external function must be compatible with the Modelica specification.

The `Include` annotation is a string which in this case includes the implementation of the function. The contents of the `Include` string is inserted into the generated C code, hence it should contain valid C code. It can even contain several line-breaks in order to include several files or even preprocessor macros. There is no guarantee that the header will only be included once, and thus necessary to guard against multiple inclusion with `#ifndef` and `#endif` wrapping.

The code (`add2.c`) should be located either in the current directory, in a relative location (`#include <../source/add2.c>`), or in directory `dymola/source`.

Linking to external library

***This section refers only
to the Windows version
of Dymola.***

For most application it is best to build a library with function definitions using software development tools outside of the Dymola environment, and then link the library to the model. In this way libraries are easily shared between a variety of applications.

Dynamic libraries are not supported.

A major benefit compared to including function definitions is that the risk of interference between the code generated by Dymola and the code of the external functions is greatly reduced. The only parts included in the compilation of the model are header files declaring the external functions in C, and the implementation is compiled separately.

It is currently only possible to link static libraries, not DLLs.

Building an external library

Using an external library the C code consists of two parts: a header file declaring the function, and an implementation. It is common to use a single header file to declare a group of related functions. The header file for the example above would be:

```
#ifndef ADD2_H
#define ADD2_H
extern double add2(double x, double y);
#endif
```

The implementation is very similar to the code used for inclusion in the model code, but the header file should be included to ensure compatibility in the event of changes in the interface. The `#ifndef` and `#endif` wrappers are not needed.

```
#include <add2.h>
double add2(double x, double y)
{
    return x + y;
}
```

Using the default compiler (GCC), the function is compiled and included in the library with these DOS commands. On Windows, the directory of the GCC compiler (`dymola/egcs/bin`) must have been added to the PATH environment variable.

```
set PATH=c:\dymola\egcs\bin;%PATH%
```

The compile and link commands are:

```
gcc -c add2.c
ar rv libext.a add2.o
ranlib libext.a
```

The name of the library shall start with `lib` and the extension (for GCC) is `.a`. These conventions are assumed when the model is linked.

Library annotation

The Modelica interface uses two special annotations, `Include` and `Library`, to specify the header file and the name of the library:

```
function add2 "Sum of two numbers"
    input Real x, y;
    output Real sum;
    external "C";
    annotation(Include="#include <add2.h>", Library="ext");
end add2;
```

Prefix and extension are not given.

Note that the library name is “ext”; the “lib” prefix is added by the linker, and the extension depends on the used compiler (.a for GCC and .lib for Microsoft C). This ensures portability of the Modelica interface to different platforms and compilers.

As a more complex example consider an interface to National Instruments AI_VRead in its Ni-Daq library. A protected variable is used to pick up the status code returned from the function.

```
function AI_VRead "Analog in"
annotation (
    Include="#include <nidaqex.h>",
    Library={"nidaq32"});
    input Integer iDevice=1;
    input Integer iChannel=1;
    input Integer iGain=1;
    output Real dVoltage;
protected
    Integer iStatus;
    external "C" iStatus = AI_VRead(iDevice, iChannel, iGain,
        dVoltage);
end AI_VRead;
```

The Library annotation is either a single string or a vector or strings that name several binary libraries, and the compiler will link with all listed libraries.

Note that for this example to work the header and library files must be in the search path of the compiler. This could be accomplished by placing the header in %DYMOLA%/source and the library in the correct sub-directory of %DYMOLA%/bin or by placing both of them in the current directory.

Other languages

Functions written in C++ or FORTRAN 77 are supported provided the C compiler supports cross-linkage with C++ or FORTRAN. When using languages other than C, provisions must be made to ensure that the required runtime libraries are linked.

C++

Functions written in C++ must be declared as `extern "C"` to be linkage-compatible with C. Wrapper functions are needed for example to use virtual functions or other C++ features that are not present in C.

FORTRAN

FORTRAN code can be linked in two ways. Perhaps the most straight-forward approach is to convert the FORTRAN code to C using a tool called `f2c`. This tool translates the code into portable C code, and also includes libraries for common FORTRAN runtime routines. The alternative is to use a link-compatible FORTRAN compiler.

In either case, wrapper functions are most likely required to map argument types.

Means to control the selection of states

Dymola supports automatic state selection according to the specification of Modelica.

Variables being subtypes of Real has an attribute, `stateSelect`, to give hints or even imperatively control the selection of variables to use as continuous time state variables.

Note that, the state selection is separated from the specification of initial conditions. The `fixed` attribute should exclusively be used for specifying start conditions and it should not influence the selection of states at all.

Motivation

The general view is that selection of states ought to be done automatically. This is also possible and unproblematic in most models, and we thus clearly understand that manual state selection can easily be overused. However, there are several reasons for allowing model library developers as well as users to influence or control the state selection:

- **Accuracy:** There are often many sets of state variables that will work from a pure mathematical point of view. However, they may have drastically different numerical properties. For mechanical systems it is favourable to use relative positions as state variables. If absolute coordinates are used then accuracy is lost when taking differences to calculate relative positions. The effect is drastic in rotating machinery systems and power systems where angular positions are increasing with time, but relative positions are rather constant, at least in normal operation. Say that two rotating bodies are connected by a spring such that the relative distance between them are 1 and that their angular speed is 1000. If the positions are calculated with a relative accuracy of 0.001, after one second there is hardly any accuracy in calculating the distance by taking the difference. The difference behaves irregularly and gives an irregular torque. The simulation stops. It is very difficult for a tool to find this out without actually doing simulation runs. Model developers for mechanical systems and power systems know it very well. It would be easy for them to indicate that absolute positions are bad choices when selecting states.
- **Efficiency by avoiding inverting functions:** The relations between possible sets of state variables may be non-linear. For some choices it may be necessary to invert non-linear functions, while for another set it is straightforward to calculate others. A typical example is thermodynamic problems, where you have property functions. They often assume two variables to be inputs (for example pressure and enthalpy) and calculate other properties (such as temperature, density etc). Thus, if such variables are selected as state variables it is “simply” calling property functions to calculate other need variables. If not it is necessary to solve equation systems to calculate the input variables. A model library developer knows this and it is straightforward to him to indicate good choices when selecting dynamic states.
- **Selecting a less nonlinear representation:** Different sets, x , of states gives an ODE, $\text{der}(x) = f(x)$ where the right hand side f have different properties. In general, the problem is simpler to solve if f is a less nonlinear problem. The Park transformation for three-phase

power systems is a classical way of transforming a nonlinear time-varying ODE into a time-invariant linear ODE. For control design it is very favourable to have linear time-invariant models, because there are lot of analysis and design methods and tools for such models. When using linearized versions of Modelica models it is important that the set of state variables is insensitive to minor changes in the model.

- **Avoiding dynamic state selection:** When selecting states the problem consists of a set of algebraic state constraints that relate dynamic variables. It may be remarked that these constraints are equations that are differentiated by Pantelides's algorithm. The task when selecting states is actually to use the algebraic constraints to solve for some of the variables, which thus are deselected as states and the remaining dynamic variables become state variables. A subset of dynamic variables can be deselected locally if its Jacobian is non-singular. In the general case the state selection must be made dynamic, but in many real applications it is possible to make a static selection of states. If the Jacobian has constant elements it is straightforward to make it automatically. However, for non-linear problems such as closed kinematics loops it is difficult to establish that a time-varying Jacobian always is non-singular. For reasons of efficiency it would be favourable to avoid the overhead of dynamic state selection and allow a user to inform that a certain selection of states will always work. Tools can support such an explicit control. Using dynamic state selection and making off-line simulations one can find a fixed choice that will work for real-time simulation, where efficiency is really needed.
- **The reinit construct:** The construct `reinit(x)` requires that `x` is state.
- **Use auxiliary variables as states:** To avoid unnecessary differentiation, it is useful to consider only variables appearing differentiated in a model as candidates when selecting states. It means that if a user would like to see an auxiliary variable, `v`, as a state variable, he has today to introduce another variable, say `derv` and an equation `derv = der(v)` to make the derivative `der(v)` appear in the model. It would be convenient to have a simpler way to introduce a variable as a state candidate.
- **Sensors:** A sensor for measuring speed, `v`, makes a variable differentiated, `v = der(r)` and in most cases it is not desirable to have the variable of the sensor model as a state variable. Introduction of variable for just plotting should not influence the state selection.

The state select attribute

A variable being subtype of Real variable has an attribute `stateSelect` to indicate its possible use as state variable. Its value can be

- *never*: Do not use as a state at all.
- *avoid*: Avoid it as state in favour of those having the *default value*
- *default*: If the variable does not appear differentiated in the model this means *no*.
- *prefer*: Prefer it as state over those having the *default value*.
- *always*: Do use it as a state.

The values of the stateSelect attribute are to given as

```
Real y(stateSelect = StateSelect.never);  
Real y(stateSelect = StateSelect.avoid);  
Real y(stateSelect = StateSelect.default);  
Real y(stateSelect = StateSelect.prefer);  
Real y(stateSelect = StateSelect.always);
```

The two extreme values *never* and *always* have clear and context independent meanings. If stateSelect is *always*, the variable will be a state. If such a variable does not appear differentiated in the model, the index reduction procedure will differentiate equations in order to be able to calculate the derivative. A model with two variables, x and y, with attribute *stateSelect* being *always* and being algebraically constrained, is thus erroneous. It is compulsory for variables appearing as arguments in reinit expressions. It supports explicit control of the selection of states and gives the user full control. It eliminates use of dynamic state selection. A dynamic state selection problem should only include variables having *stateSelect* being *prefer*, *default* or *avoid*.

The value *never* forbids the variable to be used as a state and it solves the sensor problem:

```
Real r(stateSelect = StateSelect.never);  
Real v = der(r);
```

The value *prefer* indicates that the variable should be used as a state when possible. The ambiguity lies in that there may be several candidates with *prefer* when selecting states. It solves the problem of giving preference to relative positions in mechanical problems. It is also useful for thermodynamic problems to avoid nonlinear equation systems. However, here the value *never* may be useful to rule out other candidates as well.

The value *default* means *never* for algebraic variables of the model. The index reduction procedure may introduce derivatives of algebraic variables when differentiating equations. However, this should not make them candidates for being state variables. Neither should higher order derivatives make derivatives candidates for being state variables. For example in mechanics we have

```
der(r) = v;  
m*der(v) = F;
```

The index procedure may introduce the second order derivative of r, but we should then not consider der(r) as candidate for being state variable.

The priorities for state selection are thus *always*, *prefer*, *default* and *avoid* for variables appearing differentiated.

Using noEvent

Note that this is an advanced section, and in most cases one should not use noEvent. This section describes the exceptions, and how to correctly use noEvent.

Background: How events are generated

By default Dymola generates events for the relational operators ($>$, \geq , $<$, \leq) and certain built-in functions: abs, sign, ceil, floor, div, mod, and rem. A simple optimization ensures that events are only generated if the arguments are varying continuously. Events are generated *after* the boolean expression have changed value, and it is thus necessary that expressions involving relations are valid and smooth a certain amount past the actual event.

Events are generated for code for equations and algorithms outside of functions. For algorithms there are currently some minor limitations for events in for-loops and severe limitations inside while-loops.

The problem of using events for *all* relations is that one cannot use an expression to guard against errors, e.g. square-root of a negative number, since the boolean guard would keep the value from the previous event. Furthermore, the events can lead to undesirable degradation of performance, if the derivatives are sufficiently smooth. On the other hand, if the derivatives are not smooth removing events would degrade performance even more.

Guarding expressions against evaluation

Certain numerical operations have a limited range of allowed input values, e.g. one cannot divide by zero and one cannot take the square root of a negative number. To guard against this one must use noEvent surrounding the guard condition.

As an example consider guarding against taking the square root of a negative number. An idealized model of a tank that is emptied through a hole in the bottom is:

$$\dot{h} = \begin{cases} -c\sqrt{h}, & \text{if } h \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

The model is best written as

```
Real h "Height of water in tank";
equation
  der(h)=if noEvent(h>=0) then -c*sqrt(h) else 0;
```

As an alternative we could in this particular case rewrite it using the max-function resulting in equations that are more compact, but less readable.

```
Real h "Height of water in tank";
equation
  der(h)=-c*sqrt(max({0,h}));
```

Note that we guard against h being negative even if the exact solution to the differential equation has $h(t) \geq 0$. The reason is that the numerical solution generates an approximate $h(t)$ which will be slightly negative. In general one must not only guard against the possible values for the exact solution, but also for all nearby approximate values.

Another example concern guarding against division by zero.

```
Real x;
Real sinc_x;
```

```

equation
  sinc_x=
    if noEvent(abs(x)<=Modelica.Constants.eps) then
      1
    else
      sin(x)/x;

```

In these two examples the noEvent is necessary in order to make it possible to evaluate the model for all values. Note that in the second example we do not compare abs(x) with 0 but with the slightly larger Modelica.Constants.eps. This provides a guard against roundoff errors. The extra error is insignificant since the error in the sinc-approximation it is proportional to the square of x, and thus lost in round-off.

How to use noEvent to improve performance

In some cases the derivatives are sufficiently smooth that events need not be generated, as an example we have piece-wise interpolation polynomials.

```

parameter Real cpos,cneg;
Real x,y;
equation
  der(x)=noEvent(if x>0 then cpos*x else cneg*x)
  +if time>=1 then y else 0;

```

For this to be useful the right hand side must be a sufficiently smooth function of x. In this case we note that the switch between the branches occurs for x=0 in which case both branches are zero. Note that as for all performance optimizations one should measure the performance to verify that the optimization is beneficial.

Additionally noEvent only surrounds the expression involving x and not the second term, which will generate a time event exactly at the time 1.

A note on style

When using the noEvent-operator to improve performance we are implicitly stating that the expression is sufficiently ‘smooth’. Dynasim has initiated work in the Modelica design process to introduce an operator ‘smooth’ for this purpose, and this thinking explains why noEvent is used surrounding the entire expression in the examples above and not only the actual relational operator. When viewing it as smooth it only makes sense to view a real-valued expression as smooth. This also guards against accidentally introducing events in one of the sub-expressions.

Combined example for noEvent

A more subtle example is if the noEvent is necessary in order to guarantee that we can solve for all algebraic variables. This is more complex than merely being able to evaluate the model. A simple example occurs for the turbulent flow equation

$$\Delta P_{\text{loss}} = c \dot{m} |\dot{m}|$$

Real mdot,Ploss;

```

parameter Real c;
equation
  Ploss = c* mdot*noEvent(abs(mdot));

```

Remember that `abs(mdot)` is expanded into an if-expression, in this case leading to:

```
Ploss = c* mdot*noEvent(if mdot>=0 then mdot else -mdot);
```

The `noEvent` allows us to use this equation to implicitly solve `mdot` from `Ploss`. Without the `noEvent` it would not be possible to solve the equation past the point where `mdot` changes sign (since it would be tantamount to taking the square root of a negative number).

We can also manually solve this equation for `Ploss` resulting in

```

Real mdot,Ploss;
parameter Real c;
equation
  mdot = noEvent(if Ploss/c>=0 then sqrt(Ploss/c) else -
sqrt(-Ploss/c));
  // Or: mdot = noEvent(sign(Ploss/c)*sqrt(abs(Ploss/c)));

```

We have here not considered the possibility that `c` might be zero.

The right hand side is here continuous when `Ploss` passes through zero, and thus `noEvent` can be seen as a performance improvement. Additionally we guard against taking the square of a negative number by using `noEvent`.

An additional note on style

In this example we note an additional reason for using `noEvent` around the entire expression: if one of the branches of the if-expression contain any relations these relations should also be inside `noEvent`.

As an example consider the out-commented part of the previous example where we use `abs` and `sign` for `Ploss/c`. When using that form it is easy to forget that not only `abs` but also `sign` generates events and only have `noEvent` surrounding `sqrt`. Although it does generate errors, we unnecessarily lose performance due to the `sign`-events. By having `noEvent` around the entire expression this is automatically handled.

Mixing `noEvent` and events in one equation

There are no restrictions on using variables computed using `noEvent` to trigger events in other equations, or to applying `noEvent` to expressions involving variables computed using events. This allows a modeler to combine models independently of whether they internally use events or have turned them off using `noEvent`.

Sometimes it makes sense to mix `noEvent` and events in *one* equation. This is, however, only allowed under certain restrictions since the number of active relations generating events can only change at events. We would otherwise not be able to determine if the expression had changed value or not, and thus would be unable to correctly determine when to trigger the event.

Thus if the condition of an if-expression, if-clause, while-clause, or the indices of a for-loop is a relation where events have been turned off by using noEvent the bodies should not contain relations generating events. They can, of course, depend on variables that are computed using events. As a concrete example consider the following:

$$\dot{x} = \begin{cases} -1, & \text{if } x \geq 1 \\ -x, & \text{if } x \geq 0 \wedge x \leq 1 \\ 2x, & \text{if } x \leq 0 \end{cases}$$

Assuming that we do not want events when x passes through 1 an illegal example would be:

```
model ILLEGAL
    Real x(start=2);
equation
    der(x) =
        if noEvent(x<1) then
            if x>0 then -x else 2*x
        else -1;
end ILLEGAL;
```

In order to explain why this is illegal consider what happens after one second when x becomes less than one and $x>0$ is evaluated for the first time (without triggering an event because of the noEvent). The next if-expression should trigger an event if $x>0$ does not have the value from the last event. However, we did not evaluate it at the previous event and thus we do not know if it has changed value or not and thus we do not know when to trigger an event.

We are not allowed to evaluate the relations inside the wrong branch of the if-expression since that might involve unsafe operations, such as square roots of negative number, indexing outside of bounds, etc.

To solve this problem we can either surround the entire right-hand side by noEvent or introduce an auxiliary variable for ‘if $x>0$ then $-x$ else $2*x$ ’.

Conditional use of events

An extreme example of mixing noEvent and events is to have one variable control whether events should be generated or not. This is only applicable if noEvent is introduced for performance reasons.

We revisit the interpolation example from “How to use noEvent to improve performance” on page 261, and let a boolean parameter control whether we generate events or not.

```
Real x;
parameter Boolean generateEvents;
parameter Real cpos,cneg;
equation
    der(x) =
        (if (if generateEvents then x>0 else noEvent(x>0))
         then cpos*x
```

```

    else cneg*x)
+if time>=1 then y else 0;

```

If generateEvents is not a boolean parameter, but changes continuously we have to be more careful.

```

Real x;
Real level;
parameter Real eventLimit;
parameter Real cpos,cneg;
equation
level=noEvent(abs(der(x-time))*2+abs(x-time));
der(x)=
(if (if level>eventLimit then x>0 else noEvent(x>0))
then cpos*x
else cneg*x)
+if time>=1 then y else 0;

```

Here level must be computed using noEvent since we do not want to introduce extra events every time der(x) or x changes sign, but level is used without any noEvent in the if-expression since the number of relations generating events would otherwise change between events.

It is not possible to store the expression in parenthesis in the if-expression in a boolean variable, since it can change its value between events. To re-use it introduce an extra Real variable as follows.

```

Real x;
Real level;
parameter Real eventLimit;
Real xIsPositive;
parameter Real cpos,cneg;
equation
level=noEvent(abs(der(x-time))*2+abs(x-time));
xIsPositive=if
(if level>eventLimit then x>0 else noEvent(x>0)) then
1
else
-1;
der(x)=(if noEvent(xIsPositive>0) then cpos*x else cneg*x)
+if time>=1 then y else 0;

```

Note the noEvent on the last line. Without it we would always have an event when x changed sign. If level>eventLimit the expression x>0 will introduce events and thus xIsPositive need not introduce an additional event.

Constructing anti-symmetric expressions

Quite a few expressions are naturally anti-symmetric in some variable. In most cases this requires no extra thought, and only requires one to write the expression in the natural form, and it will be anti-symmetric.

However, in some cases it is known that a formula is anti-symmetric, and it straightforward to give a formula for positive values, but more difficult to give a simple formula valid for both positive and negative values. The natural idea in these cases is to extend the expression for positive values to an anti-symmetric formula valid for all values.

As a generic example consider

$$\dot{x} = \begin{cases} g(x, p), & x \geq 0 \\ -g(-x, p), & x \leq 0 \end{cases}$$

As a model this is written as:

```
Real x;
parameter Real p;
equation
  der(x)=noEvent(if x>=0 then g(x,p) else -g(-x,p));
  assert(noEvent(abs(g(0,p))<=1e-10),
         "Formula requires g(0,p)=0");
```

In most cases the assert-statement would be removed for efficiency reasons, and the function g replaced by an expression involving x , p , and perhaps other variables.

There are several details worth explaining in this example.

First and foremost neither abs nor sign are used; the reason is that by having only one test for the sign of x we guarantee that Dymola can correctly differentiate the expression (provided $g(0,p)=0$), and use it to reduce the index and to compute Jacobians for non-linear system of equations involving this equation. If we had used sign as defined in Modelica 1.4 the Dymola program would be unable to correctly differentiate the expression.

Second noEvent is used. The reason is that we assumed that $g(x,p)$ was not valid for negative x , either because it cannot be evaluated or because it generated incorrect results. Thus using noEvent guarantees that $g(x,p)$ is only evaluated for positive values of x .

Thirdly, the formula is only valid if $g(0,p)=0$. There are anti-symmetric expressions that do not obey this, such as friction force depending on relative velocity. In those cases one should introduce an extra locked state, as described in the friction logic models, and under no circumstances use noEvent . Using the above formula without thinking would for friction lead to a sliding mode, and many unnecessary events.

Although one has to be careful one can construct a variant of this model, where we introduce an auxiliary variable for the sign. In this form it is also possible to have expressions generating events in the expanded form of $g(x,p)$, provided they do not depend on x .

```
Real x,sign_x;
parameter Real p;
equation
  sign_x=noEvent(if x>=0 then 1 else -1);
  der(x)=sign_x*g(sign_x*x,p);
  assert(noEvent(abs(g(0,p))<=1e-10),
        "Formula requires g(0,p)=0");
```

Similar remarks as for the first example apply to this example.

Equality comparison of real values

Following the Modelica specification Dymola does not allow you to compare two real values for equality. The reason is not that it would be difficult to allow it, but that the desired result depends on circumstances and there is not *one* correct way of re-writing it as legal code. Instead of automatically generating a result that would only work for some cases, you are required to manually select the desired result.

Type of variables

In many cases it does not matter whether variables are declared as Integer or Real. However, only integers may be used as indices and compared for equality. Thus some equality comparisons between real expression can be removed by replacing Real variables by Integer variables (and perhaps a fixed scaling).

In some cases all variables and constants appearing in the expression are integers, but some operations generates a real valued result, e.g. division. If the result is known to be an integer one can replace these by integer division (div) or use integer to convert a real-valued expression to an integer-valued one.

Trigger events for equality

In some cases one want to perform some special action triggered when two real expressions have the same value. This is not possible, but it is for continuous varying variables equivalent to triggering the condition as follows, which is a part of car model where we want to terminate the simulation when the velocity is equal to 100km/h.

```
Modelica.SIunits.Velocity velocity;
constant Modelica.SIunits.Velocity stopAt=100/3.6;
equation
  when {velocity>=stopAt,velocity<=stopAt} then
    if not initial() then
      terminate("Velocity is 100km/h");
    end if;
  end when;
```

Assuming this is a normal test of accelerating to 100km/h it is possible to remove the second triggering condition $\text{velocity} \leq \text{stopAt}$, because we know that the original velocity is less than 100km/h. This also allows us to remove if-statement. Similar reasoning apply to many similar cases.

Locking when equal

In some models, e.g. bouncing balls it is natural to enter another state when the relative velocity is zero in order to avoid chattering. By necessity this should not occur when the relative velocity is exactly zero, but when it is small enough. The first challenge is thus to guarantee that the ball stops bouncing in this case, the second is that if we apply a relative force it should start bouncing anew. A model demonstrating this is given below.

```
model BouncingBall
    import Modelica.SIunits;
    SIunits.Height x(start=1);
    SIunits.Velocity v;
    SIunits.Force f;
    parameter SIunits.Mass m=2;
    parameter Real ebounce=0.5;
    parameter SIunits.Velocity vsmall=1e-4;
    Real fext;if time<10 then 0 else 350*sin(time)/(1+time);
    Boolean locked(start=false);
equation
    der(x)=v;
    m*der(v)=if locked then 0 else f;
    f=-m*Modelica.Constants.g_n+fext;
    when {x<=0,locked} then
        reinit(x,0);
        reinit(v,if locked then 0 else -ebounce*v);
    end when;
    locked=if pre(locked) then f<=0 else x<=0 and
        abs(v)<=vsmall and f<=0;
end BouncingBall;
```

Note that the logic here is more complex due to the fact that we have an external force in addition to the inherent bouncing of the ball. When writing such state machine logic it is vital that the expression used to enter the locked state is still true in the state (in order to avoid chattering), and to be safer one can rewrite it as

```
locked = f<=0 and if pre(locked) then true else
    x<=0 and abs(v)<=vsmall;
```

A more advanced model would have these as relative quantities and allow both the ball and the surface to move.

Guarding against division by zero

The sole exception where comparison of two real expressions would make sense is when it is used to guard against a single exceptional value, e.g. division by zero. By careful analysis it is general possible to show that the guard can and should be applied to slightly larger values, and use the technique in “Guarding expressions against evaluation” on page 260.

In some cases only one exact value is exceptional, and it is not possible to apply the guard for other values. In those rare cases one can use the same technique without any extra epsilon, i.e. ‘noEvent(abs(x)<=0)’.

APPENDIX — MIGRATION

Appendix — Migration

Migrating to newer libraries

Dymola supports migration of models using one set of model libraries to another set of model libraries with model components of similar structure, but for example having other class names or different connector names or parameter names. Dymola has commands to build up internal translation tables which will take effect when a model library or a model component is loaded. These commands can be collected in a script allowing all models that use a specific library to upgrade to the new one.

How to migrate

Assume that we would like to migrate myModel that uses a package OldLibrary to exploit a NewLibrary. In many cases in particular when release a new version of a library, the library developer will provide a convert script, which specifies the translation from the old version to the new version of the library.

Assume that the model is stored in a file named myModel.mo. It is advisable to also have a backup copy of the file. Assume also that a specification of the translation when migrating from using OldLibrary to using NewLibrary is specified in the script file Convert.mos.

To migrate myModel proceed as follows.

1. Start with a fresh Dymola.

2. Load NewLibrary.
3. In the Dymola main window select File/Run File Script@ to run Convert.mos.
4. Load myModel.mo
5. In the model window “myModel” select File/Check.
6. Hopefully there is no error message and the model can be saved. The conversion is done.
7. In case of error message consult the next two subsections on specifying translation and building a script file.

After a migration, the model shall of course be tested and it shall be checked that it gives the same simulation result as the old one.

Basic commands to specify translation

The command to build the translation tables are

```
convertClass("oldClass", "newClass");
convertElement("oldClass", "oldElement", "newElement");
convertModifiers("oldClass", oldParameterBindings,
                 newParameterBindings);
convertClear();
```

convertClass

The command

```
convertClass("oldClass", "newClass");
```

builds conversion tables to be applied on extends clauses and type declarations. As an example consider:

```
convertClass("DriveTwoCut",
            "Modelica.Mechanics.Rotational.Interface.Compliant");
```

All components of type DriveTwoCut or classes that contain extends DriveTwoCut will be converted such that they refer to class Modelica.Mechanics.Rotational.Interface.Compliant instead.

Conversion is also applied on hierarchical names. For example

```
convertClass("Modelica.Rotational1D", "Modelica.Rotational")
```

will take effect on Modelica.Rotational1D.Clutch and give Modelica.Rotational.Clutch.

Modelica’s normal vectorization applies for convertClass, which means that it is possible to let the arguments be vectors of Strings. For example,

```
convertClass({"oldClass1", "oldClass2"},
            {"newClass1", "newClass2"});
```

is equivalent to

```
convertClass ("oldClass1", "newClass1");
convertClass ("oldClass2", "newClass2");
```

convertElement

The command

```
convertElement ("oldClass", "oldElement", "newElement");
```

converts references to elements (connectors, parameters and local variables of classes) in equations, connections, modifier lists etc. The class name of the component is still converted according to convertClass.

The conversion uses the model structure after conversion, thus it correctly detects base-classes among the models you convert. However, only the new library is used, thus any inheritance used in the old library is lost. It means, for example, if a connector was renamed in base-class conversion of that connector name must be specified for all models in the old library extending from base-class. By using vector-arguments to the conversion functions it is only necessary to list the classes once for each renamed element.

Let us illustrate by an example. Assume that we have a drive train library, where there is a partial class DriveTwoCut specifying two connectors pDrive and nDrive. The new library has a similar class TwoFlanges defining two connectors flange_a and flange_b. We thus give the commands

```
convertClass ("DriveTwoCut", "TwoFlanges");
convertElement ("DriveTwoCut", "pDrive", "flange_a");
convertElement ("DriveTwoCut", "nDrive", "flange_b");
```

Assume that the old library contains the models Shaft and Gear, which are to be converted as Inertia respectively IdealGear:

```
convertClass ("Shaft", "Inertia");
convertClass ("Gear", "IdealGear");
```

Assume that Shaft and Gear extend from DriveTwoCut. Unfortunately, there will be no translation of references in, for example, connect statements to their connectors pDrive and nDrive, since the conversion uses the model structure after conversion. To have a proper translation, we need also to specify

```
convertElement ({ "Shaft", "Gear"}, "pDrive", "flange_a");
convertElement ({ "Shaft", "Gear"}, "nDrive", "flange_b");
```

where the vectorization allows a compact definition.

The convertElement command can also be used when a parameter is renamed. For more complex reparameterizations the command convertModifiers is useful.

convertModifiers

The command

```
convertModifiers ("oldClass", oldParameterBindings,
newParameterBindings);
```

specifies how parameter bindings given in modifiers are to be converted. The argument oldParameterBindings is a vector of strings of the form “oldParameter=defaultValue”, and the argument newParameterBindings is a vector of strings of the type “newParameter=expression”. To use the value of an old parameter in the new expression use %oldParameter%

As an example, assume that in the old model Clutch that the viscous friction coefficient `mue` is given as

```
mue = mueV0 + mueV1*abs(wrel)
```

where `mueV0` and `mueV1` are parameters declared in Clutch as

```
parameter Real mueV0 = 0.6;
parameter Real mueV1 = 0;
```

The model Clutch of the new model library uses linear interpolation with respect to the relative velocity, `wrel`, with a parameter `mue_pos` to define the interpolation table

```
parameter Real mue_pos[:, :] = [0, 0.5];
```

The original `mue`-equation is linear and one way of specifying a linear interpolation table is to compute its value for two arbitrary velocities. The model requires that the first velocity is zero, with `mue=mueV0`, and for the other value velocity we use velocity one, with `mue=mueV0 + mueV1`. Thus at translation we would like to obtain a new modifier

```
mue_pos = [0, value-of-mueV0;
           1, value-of-mueV0 + value-of-mueV1];
```

This is obtained by

```
convertModifiers("Clutch", {"mueV0=0.6", "mueV1=0"}, 
                {"mue_pos=[0,%mueV0%;1,%mueV0%+%mueV1%]"});
```

Example, the declarations in the old model

```
Clutch c1(mueV0=0.4,mueV1=0.1)
Clutch c2(mueV0=2*p);
```

are converted to

```
Modelica.Mechanics.Rotational.Clutch
  c1(mue_pos = [0,(0.4); 1,(0.4)+(0.1)]);
Modelica.Mechanics.Rotational.Clutch
  c2(mue_pos=[0,(2*p); 1,(2*p)+(0)]);
```

Note that since `c2` did not specify a value for `mueV1` the conversion used the default value. The substitution automatically adds parenthesis for the substituted arguments, thus avoiding the need for parenthesis in macros that are familiar to a C-programmer. The parenthesis are sometimes redundant and can be removed by going to the parameter dialogs of the corresponding components.

Prefer `convertElement` over `convertModifiers` even for parameters, and only use `convertModifiers` when there is a need for more than a one-to-one conversion of parameters.

convertClear

The command convertClear() clears the translation tables.

How to build a convert script

In order to convert a model using one library to another it is recommended to begin construct a conversion script for the library. Even if the script is not complete after converting one model it can be reused for the next model, and only amend it with additional lines for the additional library components in that model. If a library developer restructures a Modelica library, it is recommendable to construct such a script.

Below it is explained how to construct a convert script and how to amend it for additional models. For clarity assume that we would like to convert a drive train model, myOldModel, that uses components in the model library "Drive Trains" to a model, myNewModel that instead uses the components in the library Modelica.Mechanics.Rotational.

1. Copy myOldModel.mo to myNewModel.mo.
2. Make a local conversion script, say Convert.mos. Depending on what you are converting start as follows:
 - a. If you have a script for converting similar models: Use that as starting point, and at the end add the following lines (and remove similar ones):

```
openModel ("myNewModel.mo");
checkModel ("myModel");
```

- b. If you are starting from scratch, use the following as a template:

```
clear
// Start
//
// End
openModel ("myNewModel.mo");
checkModel ("myModel");
```

3. Converting class names.
 - a. In the Dymola main window select File/Clear Log and select File/Run File Script@ to run Convert.mos
 - b. There will be error messages such as
Error: Component type specifier Shaft not found
Go through all these message and list all model types that are missing.
 - c. For each missing type find the new one in Modelica.Mechanics.Rotational. You can do that by opening both libraries and comparing icons and reading documentation. For component Shaft, we select Modelica.Mechanics.Rotational.Inertia.

- d. Use a text editor to edit Convert.mos with a contents as

```

clear
// Conversion of not found Component type specifiers
convertClass("Shaft",
    "Modelica.Mechanics.Rotational.Inertia");
convertClass("Clutch",
    "Modelica.Mechanics.Rotational.Clutch");
convertClass("Gear",
    "Modelica.Mechanics.Rotational.IdealGear");
//
openModel("myNewModel.mo");
checkModel("myModel");

```

4. In the Dymola main window select File/Clear Log and select File/Run File Script@ to run Convert.mos

5. Error messages saying

Use of undeclared variable shaft1.pDrive

Use of undeclared variable shaft1.nDrive

indicate that the connectors of typical components have changed name. To fix that we include in Convert.mos before openModel("myNewModel.mo");

```

convertElement({ "Clutch", "Shaft", "Gear" },
    "pDrive", "flange_a");
convertElement({ "Clutch", "Shaft", "Gear" },
    "nDrive", "flange_b");

```

6. Error messages of the type

Error: Modifier 'fnMax' not found in Clutch.

indicate that a parameter has changed name. In simple cases when a simple renaming works use convertElement. Otherwise use convertModifiers.

7. Run the updated Convert.mos file (when asked if update myModel in the file myNewModel, answer No)

8. Keep a copy of the conversion script, Convert.mos, since it can be useful for converting similar models.

9. Now save the model.

In some rare cases it might be necessary to edit the model by hand or it is necessary develop model wrappers or a new model component.

Note, that default values for parameters are not translated. For example, if there is a model component m1 that has a parameter p declared as

```
parameter Real p = 1.0;
```

and the new model component also has a parameter p declared as

```
parameter Real p = 0;
```

then the old default value of 1.0 is lost and the new one being zero is used. If it is important to preserve the old parameter default values, this can be done by making a new model component that extends the new “m1” and modifies its parameter values according to the old m1.

APPENDIX — INSTALLATION

Appendix — Installation

Installation on Windows

This section refers only to the Windows version of Dymola.

Following installation the user may do additional setup. The installation of updates and removal of Dymola is also described below.

Holders of a floating Dymola license should install Dymola on the server first, and then proceed with the installation of the license server. After that, Dymola can be installed on all client computers, see “Installing on client computers” on page 290.

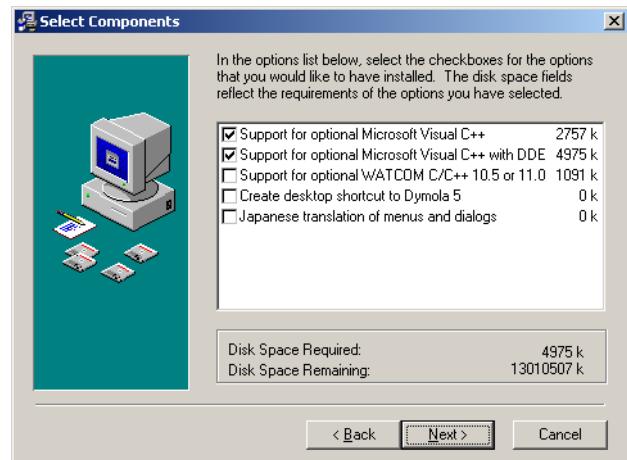
Installing Dymola

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 or Start/Run.

The first choice in the installation procedure is the name of the Dymola distribution directory. The default is `C:\Dymola`.

The second choice is to select optional components of the distribution. By unselecting components some space can be saved. The first three alternatives are related to compiler support, see below. Creating a desktop shortcut is a convenient way to access Dymola without using the Start/Programs menu.

Component selection.



Japanese translation requires special fonts.

The Japanese translation of menus and dialogs requires fonts that support all symbols. For that reason, the installation program will ask for confirmation before installing Japanese translations.



Multiple compiler support

Dymola uses a C compiler to translate the C code of the model into an executable simulation program. During the installation of Dymola, the user is given the chance to install libraries that support several different compilers under Windows:

1. Default compiler shipped with Dymola (minimum distribution gcc)
2. Microsoft Visual C/C++
3. Microsoft Visual C/C++ with Dymosim DDE support
4. WATCOM C/C++ 10.5 or 11.0

Dymola will start using the default compiler even if support for other compilers has been installed. The Microsoft compiler is faster and allows bigger models to be simulated. To

WATCOM C/C++ requires an update.

change the compiler Dymola uses to translate the model, select “Simulation/Setup...” and the “Compiler” tab, see page 154.

Users of WATCOM C/C++ 11.0 should download an updated runtime library, the file you need is called `rt1_d11.zip`. Please contact Dynasim if you have problems obtaining it.

The updated library is required if you use WATCOM C/C++ with MATLAB. Do not forget to delete the files in Dymola\tmp before you recompile Mex-files, because various object files are incompatible.

Microsoft compilers

Dymola supports several Microsoft compilers. The Dymola installation script tries to find which Microsoft compilers are installed, and prefers the latest compiler. Supported are:

- Microsoft Visual Studio 6
- Microsoft Visual Studio .NET 2002 and 2003
- Microsoft Visual C++ Toolkit 2003. This is a free compiler for Windows 2000 and Windows XP, which can be downloaded from

<http://msdn.microsoft.com/visualc/vctoolkit2003>

If additional compilers are installed after Dymola, or to use a different compiler, a setup file must be edited by hand. Examples of typical installation paths are shown in the comments in the file.

In file `dymola\bin\setvmdir.bat` modify the line `set MSVCDir=...` to the right path. The environment variable `MSVCDir` should point to the compiler’s installation directory. To find the right directory search for the file `vcvars32.bat` (e.g. `C:\Program Files\Microsoft Visual Studio\VC98\Bin\vcvars32.bat` for Visual Studio 6) and then strip off the trailing `\bin\vcvars32.bat` to get the right directory. The directory structure differs between compilers.

Optional libraries

Optional libraries (e.g. hydraulics) are distributed on separate CDs. They should be installed after installing Dymola as the files are installed in the Dymola distribution directory.

Dongle installation

This section applies only if you received a dongle with Dymola.

If your Dymola distribution contains a dongle (a small hardware device that attaches to the parallel or USB port of your computer), the appropriate dongle driver software must also be installed.

1. The driver installation file is stored on the Dymola distribution CD.
2. Double-click on `flexid.exe` to start the installation process. Please follow the instructions on the screen.
3. The dongle drivers are ready for use after the installation process reboots your computer.

4. Attach the dongle either to the parallel (printer) port or a USB port on your computer, depending on the type of dongle.

Please note that the dongle must be present every time the Dymola program is executed. If the dongle is not present, or the drivers have not been installed, Dymola will issue a message and terminate.

Additional setup

Creating shortcuts to Dymola

Shortcuts start Dymola in the right directory.

1. Click the right mouse button on the desktop.
1. Select New/Shortcut from the popup menu.
2. Browse for the Dymola program (`C:\Dymola\bin\dymola5.exe` by default).
3. Enter a suitable name and finish the creation of the shortcut.
4. Click the right mouse button on the newly created shortcut.
5. Select Properties from the popup menu.
6. Select the Shortcut tab of the dialog window.
7. Specify a working directory in the “Start in” field.

Shortcuts for Dymola language

Only for old models in the Dymola language.

The Dymola 4 program supports both the Modelica language and the Dymola language. The appropriate mode is chosen from the Start/Programs/Dymola menu under Microsoft Windows. However, if you create your own shortcuts, the switch `/Dymola` must be specified to start Dymola 4 in Dymola language mode.

1. Click the right mouse button on the shortcut to Dymola 4.
2. Select Properties from the popup menu.
3. Select the Shortcut tab of the dialog window.
4. Add `/Dymola` to the Target field.

Adding libraries and demos to the File menu

You can add your own libraries (or demo examples) to the File/Libraries menu. This is done by editing the file `dymola\insert\dymodraw.ini`, which contains further instructions.

HTML viewer for online help

Some alternatives of the Help menu display information stored in HTML. Under some circumstances Dymola will not be able to find the right viewer program; if so the user can define environment variable

DYMOLAHTMLVIEWER

which should contain the path of the viewer program. Dymola must be restarted. The “Info...” button (or menu choice) will also use HTML documentation if available, otherwise plain text.

Installing updates

Updated versions of Dymola are either distributed on CD, or can be downloaded from

<http://www.dynasim.se/update/>

The old version of Dymola should be removed before installing an update; you cannot install into an existing Dymola directory.

The downloaded update does not contain any license file. During installation of the update you are required to insert your original Dymola CD for authentication. You are also prompted for the license file, which typically is the license file on the CD (if you have obtained a newer license file by e-mail you should specify this one instead).

Removing Dymola

Under Windows, you can remove Dymola by clicking on the Start button in the Taskbar, select Settings/Control Panel and start Add/Remove Programs. Select Dymola and click on the Add/Remove button.

Note that files that you have created in the Dymola distribution directory, for example by running the demo examples, are not deleted. If so the remaining files and directories may be deleted through the Explorer.

Installation on UNIX

This section refers only to the UNIX version of Dymola.

Important information is available in `dymola\readme\unix.txt`. Please read this file for new information on the installation of Dymola.

Installing Dymola

Dymola for UNIX is normally distributed as a compressed tar-file. Extracting the file with

```
tar xvzf dymola.tar.gz
```

will create a subdirectory “`dymola`” with the entire distribution. The license file is normally copied from the distribution CD to `dymola/insert/dymola.lic`.

The Linux distribution of Dymola is configured for RedHat Linux 7.x; to run Dymola under other versions of Linux with incompatible C compilers, another library is required to run simulations. For example, to compile on RedHat 6.2 execute the following commands:

```
cd dymola/bin  
rm libds.a  
cp -p libds-rh62.a libds.a
```

To perform a simulation, Dymola will create temporary files in the directory of the model. In order to test the examples shipped with Dymola, we strongly recommend that users copy the examples directories to a private area. If there is only one user the Dymola distribution directory can be made writable.

Environment variables and setup

The following environment variables must be defined in order to run Dymola:

DYMOLA Directory root of the distribution.

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**.

PATH Search path for executable programs.

The file **\$DYMOLA/bin/dymola.setup** contains C-shell commands to set these environment variables, but will need editing to match the installation directory (if other than the user's login directory).

Note that **dymola.setup** must be read with the **source** command in C-shell, not executed as a shell script, to be effective. We suggest that you put the command

```
source $HOME/dymola/bin/dymola.setup
```

in the **.login** file in your home directory. Users of sh, bash or ksh should use the following syntax to set environment variables:

```
set DYMOLA=$HOME/dymola ; export DYMOLA
```

Alternatively, the setting of **DYMOLA** can be done when starting the program. A small command script with this contents will usually do:

```
#!/bin/csh  
setenv DYMOLA $HOME/dymola  
$DYMOLA/bin/dymola5 $*
```

On some UNIX systems the default font is tiny and hard to read. The **dymola.setup** file can be modified to set up a reasonable font for Dymola.

Additional setup

Adding libraries and demos to the File menu

You can add your own libraries (or demo examples) to the File/Libraries menu. This is done by editing the file **dymola/insert/dymodraw.ini**, which contains further instructions.

Compilation of model code

Dymola produces C code which must be compiled in order to generate a simulation model. On Windows we supply a free compiler as part of the distribution (GCC). On UNIX systems we rely on an ANSI/ISO C compiler already installed on the computer.

On UNIX systems the compilation is performed by a shell script, `dymola/insert/ds-build.sh`. If necessary this script can be modified to provide special options to the compiler, add application-specific libraries etc. For example, to compile with Sun's "ucbcc" you would have to add the `-xa` flag to the compile command to specify the ANSI compatibility option.

Removing Dymola

Remove the Dymola distribution directory and remove any settings in `.login` that you have made.

Dymola License Server

Background

These are instructions for manually installing a FLEXIm license server for Dymola. They only apply to users with a floating license. For non-floating licenses (the common case), installation of the license file is automatic.

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 all software from Dynasim.
- The license daemon (called `lmgrd.exe`) sends requests from application programs to the right vendor daemon on the right machine. The same license daemon can be used by all applications from all vendors, as this daemon processes no requests on its own, but forwards these requests to the right vendor daemon.

If you are already running an application that uses FLEXIm, you most likely already have a running license daemon. In this case only the vendor daemon (`dynasim.exe`) is required.

GLOBEtroter recommends that you use the latest version of the FLEXIm `lmgrd.exe` at all times as it includes bug fixes, enhancements, and assures the greatest level of compatibility with all of your FLEXIm licensed applications. GLOBEtroter guarantees that it will work correctly with all earlier versions of FLEXIm. The latest available license daemon can be downloaded from the website of GLOBEtroter:

<http://www.globetrotter.com/flexlm/lmgrd.shtml>

**Old license daemons
cannot be used!**

Dymola requires support of FLEXlm version 8.0d or later. A recent version of `lmgrd.exe` is part of the Dymola distribution. Note that some older versions distributed with MATLAB are not compatible.

Installing the license server

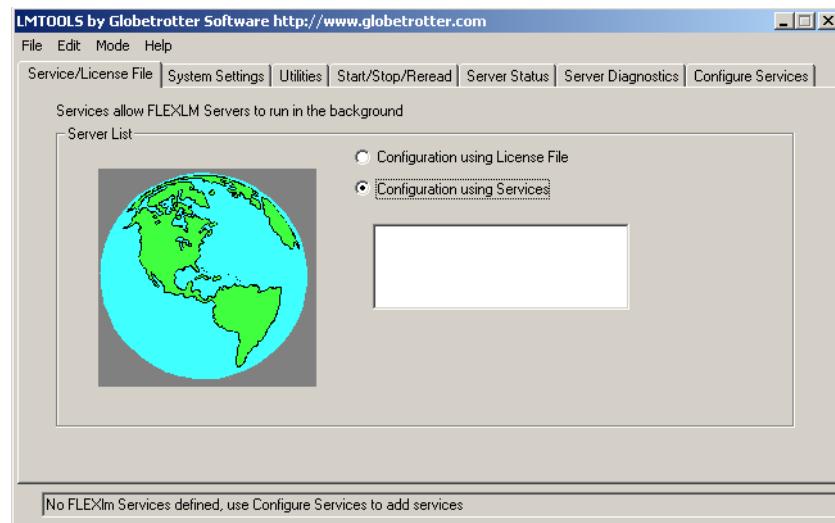
This section describes the simple case where we assume there are no other FLEXlm license daemons.

Start the utility program `dymola\bin\lmttools.exe`.

In the “Service/License File” tab:

- Select the radiobutton “Configuration using Services”.

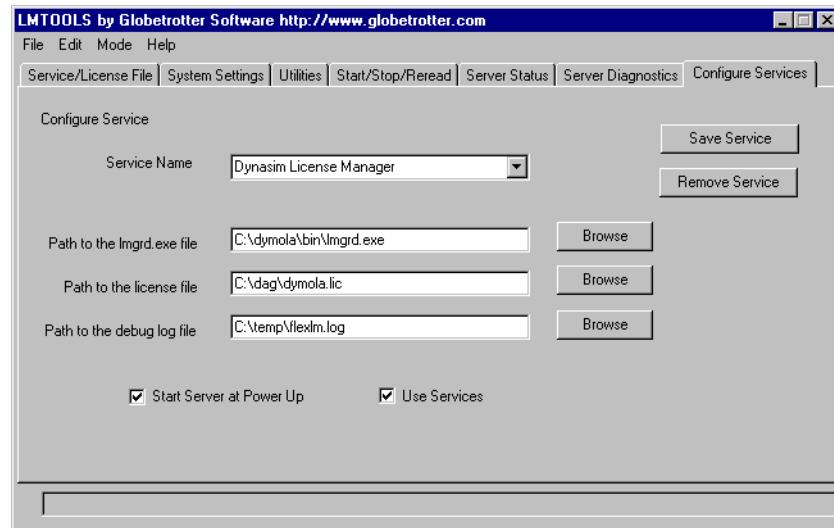
License server setup.



In the “Configure Services” tab:

- Enter a new service name, e.g. “Dynasim License Server”.
- Enter the path to the license daemon, `dymola\bin\lmgrd.exe`.
- Enter the path to your server license file, `dymola\insert\dymola.lic`.
- Enter the path to a debug log file (anywhere you want).
- Enable “Start Server at Power Up” and “Use Services”.
- Click on “Save Service”.

Configuration of the license server.



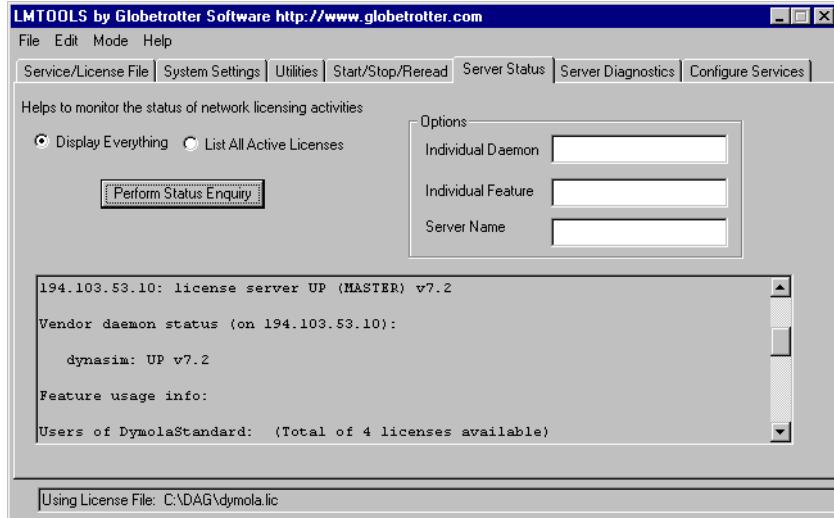
In the “Start/Stop/Reread” tab:

- Select the Dynasim license server.
- Click on “Start Server”.

In the “Server Status” tab:

- 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 (e.g. DymolaStandard or DymolaLight).

Checking the operation of the license server.



Also check the logfile to verify that the server has started and that Dymola features can be checked out. The following is an example:

The FLEXlm logfile.

```
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) DymolaSimulinkDymolaAnimation 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 FLEXlm logfile (see above) should contain additional lines showing what features were checked out. You can also do “Perform Status Enquiry” to check how many license are currently checked out.

Installing on client computers

Installation without CD.

After Dymola and all optional libraries have been installed on one computer, a simplified process can be used to install Dymola on all client computers using the license server. The following procedure should be performed on each client.

1. Open the Dymola distribution directory on a computer where Dymola has already been installed (over the network).
2. Execute the installation program `dymola\bin\Dymola5client.exe` and follow the instructions. The installation will ask for a directory to install Dymola. The default is `C:\Dymola`.

The installation program will then copy files from the server to the client computer and set up the Start/Programs menu. No CD is required to install Dymola on the clients.

Troubleshooting

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. In most cases, these problems are specific to the Windows environment.

License file

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.

- If your Dymola distribution included a dongle, drivers for the dongle must be installed and the dongle must be present. See also “Dongle installation” on page 283.
- The license file is locked to your computer system, which means that you cannot execute Dymola from another computer.
- You cannot copy the Dymola distribution directory from one computer to another; installation must be performed using the Dymola distribution CD or from the downloaded file. To install Dymola on client computers when a license server is used, see “Installing on client computers” on page 290.
- The license file on the Dymola CD for Windows is not complete. You cannot copy the license file from the CD.

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 FLEXIm error codes, is produced if Dymola is started with the command line option /FLEXImDiag. Start a command (DOS) window and issue the following commands:

```
cd \dymola\bin  
dymola5.exe /FLEXImDiag
```

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 288. The FLEXIm logfile provides additional information about the day-to-day operation of the server.

Always using the latest version of the FLEXIm license daemon lmgrd.exe is strongly recommended. It is guaranteed to be compatible with all earlier versions of FLEXIm.

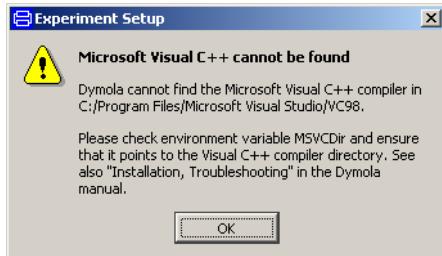
Compiler problems

The compiler used to compile the C code generated by Dymola into executable code for simulation is set in the Compiler tab, see “Simulation/Setup” on page 150.

Several potential problems can be found by pressing the “Verify compiler setup” button. Any warning messages indicate problems that need to be resolved before translating a model.

If you have setup your system to use Microsoft Visual C++ you may get the following error message:

Compiler not found.



This most likely reason is that the compiler is installed, but not in the location Dymola expects (this can happen if the compiler is installed after Dymola). You then have to manually set the location, see “Microsoft compilers” on page 283.

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 (but included in the demo version).
- You have included the directories `dymola\mfiles` and `dymola\mfiles\traj` 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 you have a working Matlab/Mex configuration 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 “Graphical interface between Simulink and Dymola” on page 219.
- 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 2π times the frequency in the corresponding source in Modelica).

The default compiler used by Matlab/Mex (called lcc) is very slow for large files. Models generated by Dymola may compile slowly using lcc, and we suggest using Microsoft Visual C/C++ instead (if available).

dSPACE systems

Older dSPACE systems (dSPACE release 2.0-3.3) may not support MAP files for Dymola models. The error message displayed by ControlDesk is

Encountered an error in the MAP file.

This problem is fixed by upgrading to dSPACE release 3.4, or by requesting an updated DLL from dSPACE.

Other Windows-related problems

Starting the installation

The installation normally starts automatically 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 or Start/Run.

Application fails to initialize properly

This is most likely caused by incorrect system DLLs. Dymola uses two system DLLs for OpenGL support. They are called `opengl32.dll` and `glu32.dll`. If these DLLs are incorrect, e.g. do not recognize the graphics card or fail initialization in some other way, you get an error message like this.

Note that with Windows NT 4.0 you get a more specific error message that mentions a problem with DLLs. In Windows 2000 it just says that the application failed.

Windows XP requires Service Pack 1 or later to run Dymola. Otherwise, XP displays the message “Windows encountered a problem with Dymola5.exe” and suggests sending a problem report.

Space characters in filenames

In some cases Dymola cannot handle filenames which contain space characters, notably the directory “Program Files”. This bug will be fixed in a future release.

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 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.

Windows environment space

Under Windows 95, 98 and ME, Dymola may under exceptional circumstances generate the error message “Out of environment space” during compilation.

The remedy is to modify two system configuration files to increase environment space: system.ini and config.sys.

1. Open \windows\system.ini with notepad. Search for a variable called CommandEnvSize. This variable should be set to at least 6000. If it does not exist in system.ini, add the following two lines:

```
[NonWindowsApp]  
CommandEnvSize=6000
```

2. On Windows 95 and 98, open \config.sys with notepad. The command environment space must also be set to 6000. Locate the line defining SHELL. The /E switch is used to define command environment space, for example:

```
SHELL=C:\COMMAND.COM C:\ /P /E:6000
```

Please keep other switches on the line unchanged. Windows must be restarted after these changes.

Index

Symbols

- = • 174
- @ • 149, 176

A

- about • 143
- absolute tolerance • 201
- acausal modeling • 76
- algorithm • 81
- animated GIF • 137
- animation • 166
 - menu • 169
 - visual shapes • 166
- arrow keys • 115, 143
- assignment • 81
 - interactive • 174
- automatic manhattanize • 121, 143
- AVI (Audio-Video Interleaved) • 137

B

- backstep animation • 169
- base class • 110, 112, 113
- BMP
 - graphical object • 124
- bound parameter • 61
- bring to front • 139
- browser
 - component • 109
 - package • 109

C

- chattering • 184
- check • 109, 139
- checkModel • 179
- class
 - description • 76, 113
 - documentation • 112
 - parameter • 79, 118
- clear all • 134

clear log • 137
clipboard • 136
closeModel • 179
color • 124
commands
 convertClass • 272
 convertClear • 275
 convertElement • 273
 convertModifiers • 273
comment • 240
 attribute see class description • 76
 initialization file • 143
compiler
 GNU C • 154
 Microsoft Visual C++ • 154, 283, 292
 on UNIX • 154
 setup • 154
 troubleshooting • 154, 291
 WATCOM C/C++ • 154
component • 17
 attributes • 120
 browser • 109
 library • 48
 size • 113, 115, 140
computer algebra • 78
connection • 75, 115, 138
 creating • 121
connector • 75
 nested • 122
 protected • 111
 visibility • 111
context menu • 47, 109, 113, 114, 116, 122
continue line • 173
continue simulation • 150
continuous run of animation • 170
ControlDesk • 227
coordinate system • 113, 140
 specification • 113
copy • 138
copy to clipboard • 136
create script • 137
creating a model • 50
crossing function • 85
current selection • 114
cut • 138

D

DAE • 78, 204
 high index • 78
data flow modeling • 77
DC motor • 42
DDE
 Dymola commands • 233
 Dymosim DDE server • 234
 extended gui • 236
 hot linking variables • 236
 requesting variables • 235
 setting parameters • 235
 simulator commands • 235
debugging models • 153, 182
declarations • 35, 125
default graphics • 124
delete • 138
delete key • 115, 143
deleting objects • 115
demo • 24, 134
dense output • 202, 203, 204
der(...) operator • 34
description • 127
description of class • 76, 113
diagram layer • 111, 142
differential-algebraic equation • 78
discrete equations • 83
document • 178
documentation
 HTML • 113
 of class • 112, 126
documentation layer • 111, 142
dongle • 283
double-clicking • 114, 116, 121, 124
dragging • 115
dsin.txt • 207
dsmodel.c • 198
dSPACE • 225
 ControlDesk • 227
 overrun situation • 228
 turnaround time • 228
duplicate
 class • 110
duplicate objects • 138
DXF-files • 167
dymodraw.ini • 143

DymolaBlock • 220
DYMOLAGIFCON • 137
DYMOLAPATH • 143
Dymosim file • 197
 dsin.txt • 199, 200, 209
 dsindef.txt • 200
 dslog.txt • 198
 dsres.mat • 198, 199, 213
 dsu.txt • 199, 200
dymtools • 199

E

edit menu
 change attributes • 139
 check • 139
 copy • 138
 cut • 138
 delete • 138
 draw shapes • 139
 duplicate • 138
 flip horizontal • 139
 flip vertical • 139
 manhattanize • 139
 options • 141
 order • 139
 paste • 138
 redo • 138
 rotate • 139
 select all • 138
 undo • 138
edit window • 108, 114
ellipses • 123, 124
embedded HTML • 128
encapsulated • 127
Enhanced Metafile Format • 136
environment space, Windows • 294
environment variables • 137, 286
equality comparison • 266
equation • 34
equations • 125
eraseClasses • 180
errors • 182
event • 85
 logging • 183
examples
 ideal diode • 87

motor drive • 73, 77
motor model • 74
polynomial multiplication • 81
rectifier circuit • 87
sampled data system • 82
exit • 138
experiment setup • 37
export
 animation • 137
 HTML • 137
 image • 136, 160
 to clipboard • 136
expression
 interactive • 173
external function • 253
 annotations • 255
 including • 254
 linking to library • 254

F

file menu
 check • 139
 clear all • 134
 clear log • 137
 copy to clipboard • 136
 demos • 134
 duplicate class • 133
 exit • 137, 138
 export animation • 137
 export HTML • 137
 export image • 136
 export to clipboard • 136
 libraries • 134
 new • 131, 132
 open • 133
 print • 136
 recent files • 137
 save • 134
 save all • 134
 save as • 134
 save log • 137
 save total • 134
 search • 134
 setup HTML • 137
file type associations • 233
fill color • 124

find
 class or component • 109, 134
 text • 126, 127, 129
FLEXlm • 287
flow prefix • 75
function • 81
 derivative • 249
 external • 253
Furuta pendulum • 65

G

gccbuild • 198
GenerateBlockTimers • 187
GenerateTimers • 189
GIF
 export of image • 160
 graphical object • 124
GIF Construction Set • 137
global error • 201
GNU C Compiler • 255
GNU C compiler • 154
graphical attributes • 124
graphical objects • 123
 default • 124
grid • 113, 115, 121, 123, 140
gridlines • 115, 139

H

handles • 114
hardware in the loop • 225
help • 176
help menu
 about • 143
 documentation • 142
 Dymola support • 143
 Dynasim website • 143
 what's this • 142
high index DAE • 78
HTML • 137
 documentation • 113, 129
 escape sequence • 128
 export • 137
 external references • 129
 hyperlinks • 129
 online documentation • 130

options • 130
setup • 137
hybrid modeling • 81
hybrid system • 183

I

icon layer • 111, 142
if-expression • 85
information window • 116
initialization of models • 90
 over specified • 182
inline integration • 190
InlineIntegration • 192
InlineIntegrationExplicit • 192
instability • 183
installation
 clients of server • 290
 dongle • 283
 environment variables • 286
 license daemon • 287
 license server • 287
 multiple compiler support • 282
 troubleshooting • 290
 UNIX • 285
 windows • 281
integration method • 148, 190
integrator

DASSL • 203, 205
DEABM • 203, 204
DOPRI5 • 203, 204
DOPRI8 • 203, 205
GRK4T • 203, 205
LSODAR • 203, 204
LSODE1 • 203, 204
LSODE2 • 203, 204
MEXX • 203, 205
ODASSL • 203, 205

J

JPEG
 graphical object • 124

K

keyboard commands • 143

L

layer • 108, 111
diagram • 111
documentation • 111
icon • 111
Modelica text • 111
library • 48
documentation • 113
shortcut to • 134
library annotation • 255
library menu • 134
library migration • 271
library window • 110
license file
error message • 291
license server • 287
line • 123
color • 124
style • 125
linearize • 150
list • 180
listfunctions • 176
logging events • 183
LogVariable • 205

M

manhattanize • 121, 139, 141
material
visual objects • 167
Matlab
mex • 219
path • 219
using Dymola models • 224
see also Simulink
matrix equations • 79
menu
animation • 169
diagram • 160
m-file • 205, 206
tcomp • 207
tcut • 207
tder • 207
tdiff • 207
tfigure • 207
tinteg • 207

tload • 199, 207
tnhead • 207
tnindex • 207
tnlist • 199, 207
tplot • 199, 200, 207
tplotm • 207
trajectory • 206
trange • 207
tsame • 207
tsave • 200, 207
tzoom • 207
Microsoft Visual C++ • 154, 283, 292
migration to a new library • 271
mixed-mode integration • 190
mode
modeling • 107
simulation • 107
model • 73
documentation • 113, 126
replaceable • 80
model debugging • 182
model editor • 73
Modelica • 73
syntax • 239
Modelica Standard Library • 42, 43, 75, 100
Modelica text • 111, 125, 142

modeling • 73
hybrid • 81
modeling mode • 107
modifier window • 110, 113, 116
modifiers • 116, 117
mouse button
left • 114
right • 114
moving
class • 110
objects • 114

N

nested connectors • 122
new
model, connector etc. • 131
package • 132
noEvent(...) operator • 86, 190, 259

O

ODAE • 204
ODE • 78, 204
Open result • 169
openModel • 178
ordering of objects • 139
output interval • 148

P

package browser • 109
parameter • 118
 class- • 79, 118
 declaration • 76
 propagation • 61
partial models • 75
paste • 138
pause animation • 169
pendulum • 33
picking objects • 114
plot • 180
plot menu
 delete diagram • 160
 new diagram • 160
 new plot window • 160
 open result • 160
 rescale • 160
plot window • 156
plotArray • 180
PNG
 graphical object • 124
polygons • 123
pre(...) operator • 83
print • 136
printPlot • 180
profiling • 187
propagation of parameters • 61
protected • 127

Q

quantity • 74
quit • 138

R

RandomNormal • 206
RandomSeed • 206

RandomUniform • 206
ReadMatrix • 206
real-time simulation • 190, 225
Real-Time Workshop • 228
recent files • 137
recent models • 141
rectangles • 123
redeclare keyword • 80
redo • 138
relative tic mark • 159
relative tolerance • 201
rename • 110
replaceable model • 80
reshaping objects • 113, 115
reverse animation • 169
rewind animation • 169
robot demo • 24
root finder • 203, 204
root model • 109
rotate • 139
RT-LAB • 229
run animation • 169
run script • 32, 137, 149

S

save all • 134
save as • 134
 results • 199
save log • 137
save model • 138
save total • 134
scaling • 201
script • 32
 create • 137
 files • 176
 functions • 181
 run • 149
search
 class or component • 109, 134
 text • 126, 127, 129
selecting objects • 114, 138
send to back • 139
setting parameters • 40
S-function MEX block • 220
shell command
 alist • 208

dymosim • 200, 207

shift key • 114, 143

shortcut to library • 134

Show Log • 155

simulate • 150

simulateModel • 179

simulation • 32, 147

 efficiency • 184

 time • 148

simulation menu

 continue • 150

 linearize • 150

 run script • 149

 setup • 150

 simulate • 150

 stop • 150

 translate • 150

simulation mode • 107

Simulink

 DymolaBlock • 220

 external input and output • 221

 graphical interface • 219

 implementation notes • 224

 parameters and initial values • 221

 Real-Time Workshop • 228

 simulation • 222

 troubleshooting • 292

 xPC • 228

special keyboard commands • 143

stability boundary • 203

starting Dymola • 23

state event • 85, 259

state selection • 257

StateSelect • 257

step animation • 169

stiff • 203, 204

stop • 138, 150

store

 as one file • 127

 in model • 151, 152

string manipulation • 175

syntax of Modelica • 239

T

terminate • 138

text editor • 125, 126

text strings • 124

toolbar • 123, 124

tooltips

 plot window • 159

translate model • 37, 148, 150

translateModel • 179

U

undo • 138

unit

 display • 74

 of measure • 74

Unix

 compiler • 154

 installation • 285

V

variable

 in Modelica • 74

 interactive • 174

 predefined • 175

variable order • 202

variable step-size • 202

variable structure systems • 86

view menu

 diagram layer • 142

 documentation layer • 142

 equation layer • 142

 icon layer • 142

visual modeling • 166

visual reference • 170

VisualShape • 166

VRML • 137

W

WATCOM C/C++ • 154

what's this • 142

window

 edit • 108

 grid • 113

 library • 110

 modifier • 110, 113, 116

 plot • 156

window menu

 diagram layer • 142

documentation layer • 142
icon layer • 142
Modelica text • 142
modeling • 141
new Dymola window • 142
new library window • 142
next • 142
previous • 142
simulation • 141
window types • 107
Windows environment space • 294
WriteMatrix • 206

X

xPC • 228
xscale • 201

Z

zooming • 142