OpenModelica Users Guide

Version 0.5, November 2005

Preliminary Draft, 2005-11-28
for OpenModelica 1.3.1

PELAB – Programming Environment Laboratory
Department of Computer and Information Science
Linköping University, Sweden

Table of Contents

Preface

This users guide provides documentation and examples on how to use the OpenModelica system, both for the Modelica beginners and advanced users.

Chapter 1

Introduction

The OpenModelica system described in this document has both short-term and long-term goals:

- The short-term goal is to develop an efficient interactive computational environment for the Modelica language, as well as a partial but rather complete implementation of the language. It turns out that with support of appropriate tools and libraries, Modelica is very well suited as a computational language for development and execution of both low level and high level numerical algorithms, e.g. for control system design, solving nonlinear equation systems, or to develop optimization algorithms that are applied to complex applications.
- The longer-term goal is to have a complete reference implementation of the Modelica language, including simulation of equation based models and additional facilities in the programming environment, as well as convenient facilities for research and experimentation in language design or other research activities. However, our goal is not to reach the level of performance and quality provided by current commercial Modelica environments that can handle large models requiring advanced analysis and optimization by the Modelica compiler.

The long-term *research* related goals and issues of the OpenModelica open source implementation of a Modelica environment include but are not limited to the following:

- Development of a *complete formal specification* of Modelica, including both static and dynamic semantics. Such a specification can be used to assist current and future Modelica implementers by providing a semantic reference, as a kind of reference implementation.
- *Language design*, e.g. to further *extend the scope* of the language, e.g. for use in diagnosis, structural analysis, system identification, etc., as well as modeling problems that require extensions such as partial differential equations, enlarged scope for discrete modeling and simulation, etc.
- *Language design* to *improve abstract properties* such as expressiveness, orthogonality, declarativity, reuse, configurability, architectural properties, etc.
- *Improved implementation techniques*, e.g. to enhance the performance of compiled Modelica code by generating code for parallel hardware.
- *Improved debugging* support for equation based languages such as Modelica, to make them even easier to use.
- *Easy-to-use* specialized high-level (graphical) *user interfaces* for certain application domains.
- *Visualization* and animation techniques for interpretation and presentation of results.
- *Application usage* and model library development by researchers in various application areas.

The OpenModelica environment provides a test bench for language design ideas that, if successful, can be submitted to the Modelica Association for consideration regarding possible inclusion in the official Modelica standard.

The current version of the OpenModelica environment allows most of the expression, algorithm, and function parts of Modelica to be executed interactively, as well as equation models and Modelica functions to be compiled into efficient C code. The generated C code is combined with a library of utility functions, a run-time library, and a numerical DAE solver. An external function library interfacing a LAPACK subset and other basic algorithms is under development.

1.1 System Overview

The OpenModelica environment consists of several interconnected subsystems, as depicted in Figure 1-1 below.

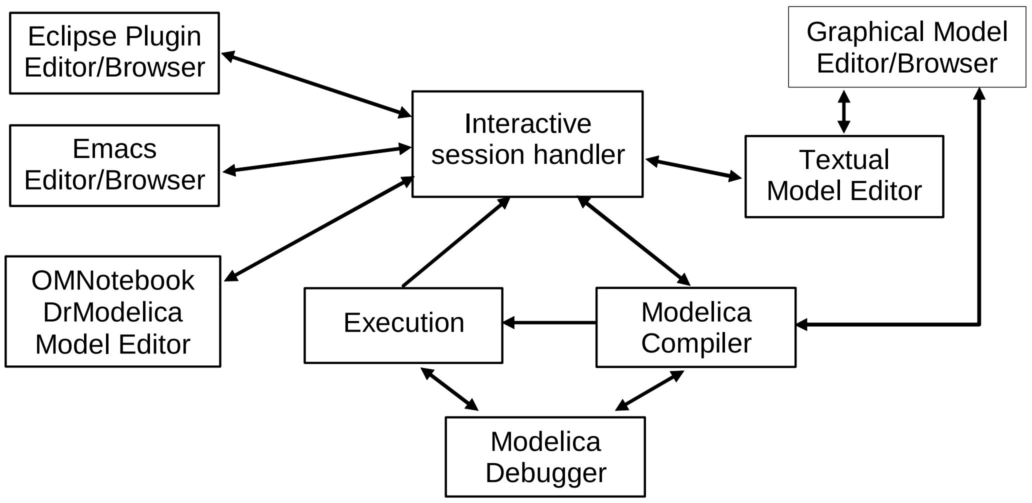

Eclipse Plugin Editor/Browser
Emacs Editor/Browser
OMNotebook DrModelica Model Editor
Interactive session handler
Graphical Model Editor/Browser
Textual Model Editor
Execution
Modelica Compiler
Modelica Debugger

Figure 1-1. The architecture of the OpenModelica environment. Arrows denote data and control flow. The interactive session handler receives commands and shows results from evaluating commands and expressions that are translated and executed. Several subsystems provide different forms of browsing and textual editing of Modelica code. The debugger currently provides debugging of an extended algorithmic subset of Modelica. The graphical model editor is not really part of OpenModelica but integrated into the system and available from MathCore without cost for academic usage.

The following subsystems are currently integrated in the OpenModelica environment:

- *An interactive session handler*, that parses and interprets commands and Modelica expressions for evaluation, simulation, plotting, etc. The session handler also contains simple history facilities, and completion of file names and certain identifiers in commands.
- *A Modelica compiler subsystem*, translating Modelica to C code, with a symbol table containing definitions of classes, functions, and variables. Such definitions can be predefined, user-defined, or obtained from libraries. The compiler also includes a Modelica interpreter for interactive usage and constant expression evaluation. The subsystem also includes facilities for building simulation executables linked with selected numerical ODE or DAE solvers.
- *An execution and run-time module*. This module currently executes compiled binary code from translated expressions and functions, as well as simulation code from equation based models, linked with numerical solvers. In the near future event handling facilities will be included for the discrete and hybrid parts of the Modelica language.
- *Emacs textual model editor/browser*. In principle any text editor could be used. We have so far primarily employed Gnu Emacs, which has the advantage of being programmable for future extensions. A Gnu Emacs mode for Modelica has previously been developed. The Emacs mode hides Modelica graphical annotations during editing, which otherwise clutters the code and makes it hard to read. A speedbar browser menu allows to browse a Modelica file hierarchy, and among the class and type definitions in those files.

- *Eclipse plugin editor/browser*. The Eclipse plugin provides file and class hierarchy browsing and text editing capabilities, rather analogous to previously described Emacs editor/browser. Some syntax highlighting facilities are also included. The Eclipse framework has the advantage of making it easier to add future extensions such as refactoring and cross referencing support.
- *OMNotebook DrModelica model editor*. This subsystem provides a lightweight notebook editor, compared to the more advanced Mathematica notebooks available in MathModelica. This basic functionality still allows essentially the whole DrModelica tutorial to be handled. Hierarchical text documents with chapters and sections can be represented and edited, including basic formatting. Cells can contain ordinary text or Modelica models and expressions, which can be evaluated and simulated. However, no mathematical typesetting or graphic plotting facilities are yet available in the cells of this notebook editor.
- *Graphical model editor/browser*. This is a graphical connection editor, for component based model design by connecting instances of Modelica classes, and browsing Modelica model libraries for reading and picking component models. The graphical model editor is not really part of OpenModelica but integrated into the system and provided by MathCore without cost for academic usage. The graphical model editor also includes a textual editor for editing model class definitions, and a window for interactive Modelica command evaluation.
- *Modelica debugger*. The current implementation of debugger provides debugging for an extended algorithmic subset of Modelica, excluding equation-based models and some other features, but including some meta-programming and model transformation extensions to Modelica. This is conventional full-feature debugger, using Emacs for displaying the source code during stepping, setting breakpoints, etc. Various back-trace and inspection commands are available. The debugger also includes a data-view browser for browsing hierarchical data such as tree- or list structures in extended Modelica.

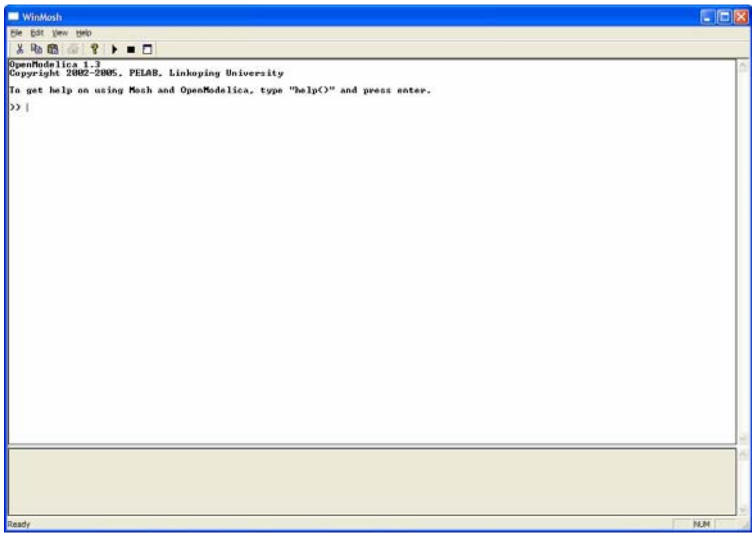
1.1.1 Implementation Status

In the current OpenModelica implementation version 1.3 (October 2005), not all subsystems are yet integrated as well as is indicated in Figure 1-1. Currently there are two versions of the Modelica compiler, one which supports most of standard Modelica including simulation, and is connected to the interactive session handler, the notebook editor, and the graphic model editor, and another meta-programming Modelica compiler version which is integrated with the debugger and Emacs, supports meta-programming Modelica extensions, but does not allow equation-based modeling and simulation. Those two versions are currently being merged into a single Modelica compiler version.

1.2 Interactive Session with Examples

The following is an interactive session using the interactive session handler in the OpenModelica environment. (Also called WinMosh.exe (under Windows) or mosh (under Linux) – the Modelica Shell).

The Windows version which at installation is made available in the start menu as `OpenModelica->OpenModelica Shell` responds with an interaction window:



```
OpenModelica 1.3
Copyright 2002-2005. PELAB. Linkoping University

To get help on using Mosh and OpenModelica, type "help()" and press enter.

>> |
```

We enter an assignment of a vector expression, created by the range construction expression `1:12`, to be stored in the variable `x`. The value of the expression is returned.

```
>> x := 1:12
 {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}
```

Load the function `bubblesort`, either by using the pull-down menu `File->Load Model`, or by explicitly giving the command:

```
>> loadFile("C:/OpenModelica13/testmodels/bubblesort.mo")

true
```

The function `bubblesort` is called below to sort the vecto `x` in descending order. The sorted result is returned together with its type. Note that the result vector is of type `Real[:]`, instantiated as `Real[12]`, since this is the declared type of the function result. The input `Integer` vector was automatically converted to a `Real` vector according to the Modelica type coercion rules. The function is automatically compiled when called if this has not been done before.

```
>> bubblesort(x)
{12.0,11.0,10.0,9.0,8.0,7.0,6.0,5.0,4.0,3.0,2.0,1.0}
```

Another call:

```
>> bubblesort({4,6,2,5,8})
{8.0,6.0,5.0,4.0,2.0}
```

It is also possible to give operating system commands via the `system` utility function. A command is provided as a string argument. The example below shows the `system` utility applied to the UNIX command `cat`, which here outputs the contents of the file `bubblesort.mo` to the output stream. However, the cat command does not boldface Modelica keywords – this improvement has been done by hand for readability.

```
>> cd("C:/OpenModelica13/testmodels")
>> system("cat bubblesort.mo")

function bubblesort
  input Real[:] x;
  output Real[size(x,1)] y;
protected
  Real t;
algorithm
```

```
   y := x;
   for i in 1:size(x,1) loop
     for j in 1:size(x,1) loop
       if y[i] > y[j] then
             t := y[i];
             y[i] := y[j];
             y[j] := t;
       end if;
     end for;
   end for;
 end bubblesort;
```

Note: Under Windows the output emitted into stdout by `system` commands is put into the `winmosh` console windows, not into the `winmosh` interaction windows. Thus the text emitted by the above cat command would not be returned. Only a success code (0 = success, 1 = failure) is returned to the `winmosh` window. For example:

```
>> system("dir")
0

>> system("Non-existing command")
1
```

Another built-in command is `cd`, the *change current directory* command. The resulting current directory is returned as a string.

```
>> cd()
"C:\OpenModelica13\testmodels"

>> cd("..")
"C:\OpenModelica13"

>> cd("C:\\OpenModelica13\\testmodels")
"C:\OpenModelica13\testmodels"
```

We load a model, here the whole Modelica standard library, which also can be done through the `File->Load Modelica Library` menu item:

```
>> loadModel(Modelica)
true
```

We also load a file containing the dcmotor model:

```
>> loadFile("C:/OpenModelica13/testmodels/dcmotor.mo")
true
```

It is simulated:

```
>> simulate(dcmotor,startTime=0.0,stopTime=10.0)

record
    resultFile = "dcmotor_res.plt"
end record
```

We list the source code of the model:

```
>> list(dcmotor)

"model dcmotor
  Modelica.Electrical.Analog.Basic.Resistor r1(R=10);
  Modelica.Electrical.Analog.Basic.Inductor i1;
  Modelica.Electrical.Analog.Basic.EMF emf1;
  Modelica.Mechanics.Rotational.Inertia load;
  Modelica.Electrical.Analog.Basic.Ground g;
  Modelica.Electrical.Analog.Sources.ConstantVoltage v;
```

```
equation
  connect(v.p,r1.p);
  connect(v.n,g.p);
  connect(r1.n,i1.p);
  connect(i1.n,emf1.p);
  connect(emf1.n,g.p);
  connect(emf1.flange_b,load.flange_a);
end dcmotor;
"
```

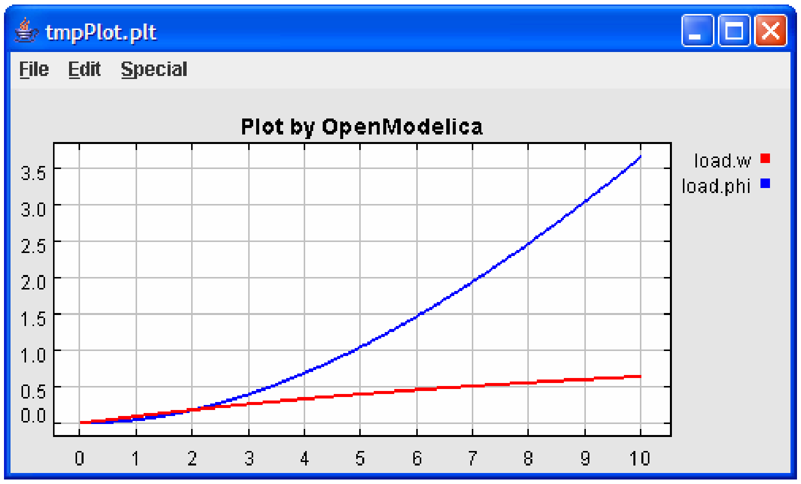
We test code instantiation of the model to flat code:

```
>> instantiateModel(dcmotor)
```

```
"fclass dcmotor
Real r1.v "Voltage drop between the two pins (= p.v - n.v)";
Real r1.i "Current flowing from pin p to pin n";
Real r1.p.v "Potential at the pin";
Real r1.p.i "Current flowing into the pin";
Real r1.n.v "Potential at the pin";
Real r1.n.i "Current flowing into the pin";
parameter Real r1.R = 10 "Resistance";
Real i1.v "Voltage drop between the two pins (= p.v - n.v)";
Real i1.i "Current flowing from pin p to pin n";
Real i1.p.v "Potential at the pin";
Real i1.p.i "Current flowing into the pin";
Real i1.n.v "Potential at the pin";
Real i1.n.i "Current flowing into the pin";
parameter Real i1.L = 1 "Inductance";
parameter Real emf1.k = 1 "Transformation coefficient";
Real emf1.v "Voltage drop between the two pins";
Real emf1.i "Current flowing from positive to negative pin";
Real emf1.w "Angular velocity of flange_b";
Real emf1.p.v "Potential at the pin";
Real emf1.p.i "Current flowing into the pin";
Real emf1.n.v "Potential at the pin";
Real emf1.n.i "Current flowing into the pin";
Real emf1.flange_b.phi "Absolute rotation angle of flange";
Real emf1.flange_b.tau "Cut torque in the flange";
Real load.phi "Absolute rotation angle of component (= flange_a.phi = flange_b.phi)";
Real load.flange_a.phi "Absolute rotation angle of flange";
Real load.flange_a.tau "Cut torque in the flange";
Real load.flange_b.phi "Absolute rotation angle of flange";
Real load.flange_b.tau "Cut torque in the flange";
parameter Real load.J = 1 "Moment of inertia";
Real load.w "Absolute angular velocity of component";
Real load.a "Absolute angular acceleration of component";
Real g.p.v "Potential at the pin";
Real g.p.i "Current flowing into the pin";
Real v.v "Voltage drop between the two pins (= p.v - n.v)";
Real v.i "Current flowing from pin p to pin n";
Real v.p.v "Potential at the pin";
Real v.p.i "Current flowing into the pin";
Real v.n.v "Potential at the pin";
Real v.n.i "Current flowing into the pin";
parameter Real v.V = 1 "Value of constant voltage";
equation
  r1.R * r1.i = r1.v;
  r1.v = r1.p.v - r1.n.v;
  0.0 = r1.p.i + r1.n.i;
  r1.i = r1.p.i;
  i1.L * der(i1.i) = i1.v;
  i1.v = i1.p.v - i1.n.v;
  0.0 = i1.p.i + i1.n.i;
  i1.i = i1.p.i;
  emf1.v = emf1.p.v - emf1.n.v;
  0.0 = emf1.p.i + emf1.n.i;
  emf1.i = emf1.p.i;
  emf1.w = der(emf1.flange_b.phi);
  emf1.k * emf1.w = emf1.v;
  emf1.flange_b.tau = -(emf1.k * emf1.i);
  load.w = der(load.phi);
  load.a = der(load.w);
  load.J * load.a = load.flange_a.tau + load.flange_b.tau;
  load.flange_a.phi = load.phi;
  load.flange_b.phi = load.phi;
  g.p.v = 0.0;
  v.v = v.V;
  v.v = v.p.v - v.n.v;
  0.0 = v.p.i + v.n.i;
  v.i = v.p.i;
  emf1.flange_b.tau + load.flange_a.tau = 0.0;
  emf1.flange_b.phi = load.flange_a.phi;
  emf1.n.i + v.n.i + g.p.i = 0.0;
  emf1.n.v = v.n.v;
  v.n.v = g.p.v;
  i1.n.i + emf1.p.i = 0.0;
  i1.n.v = emf1.p.v;
  r1.n.i + i1.p.i = 0.0;
  r1.n.v = i1.p.v;
  v.p.i + r1.p.i = 0.0;
  v.p.v = r1.p.v;
  load.flange_b.tau = 0.0;
end dcmotor;
"
```

We plot part of the simulated result:

```
>> plot({load.w,load.phi})
true
```



We load and simulate the BouncingBall example containing when-equations and if-expressions (the Modelica key-words have been bold-faced by hand for better readability):

```
>> loadFile("C:/OpenModelica13/testmodels/BouncingBall.mo")
```

```
true
```

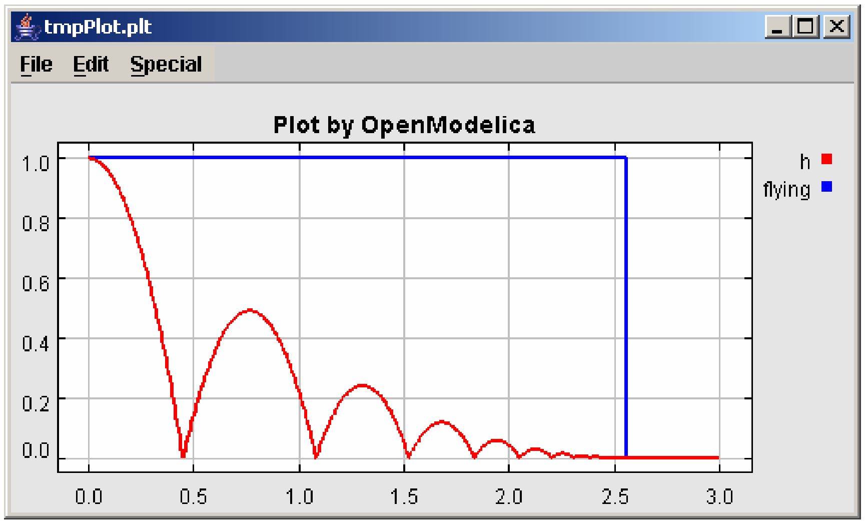
```
>> list(BouncingBall)
"model BouncingBall
  parameter Real e=0.7 "coefficient of restitution";
  parameter Real g=9.81 "gravity acceleration";
  Real h(start=1) "height of ball";
  Real v "velocity of ball";
  Boolean flying(start=true) "true, if ball is flying";
  Boolean impact;
  Real v_new;
equation
  impact=h <= 0.0;
  der(v)=if flying then -g else 0;
  der(h)=v;
  when {h <= 0.0 and v <= 0.0,impact} then
    v_new=if edge(impact) then -e*pre(v) else 0;
    flying=v_new > 0;
    reinit(v, v_new);
  end when;
end BouncingBall;
"
```

Instead of just giving a `simulate` and `plot` command, we perform a `runScript` command on a `.mos` (Modelica script) file `sim_BouncingBall.mos` that contains these commands:

```
loadFile("BouncingBall.mo");
simulate(BouncingBall, stopTime=3.0);
plot({h,flying});
```

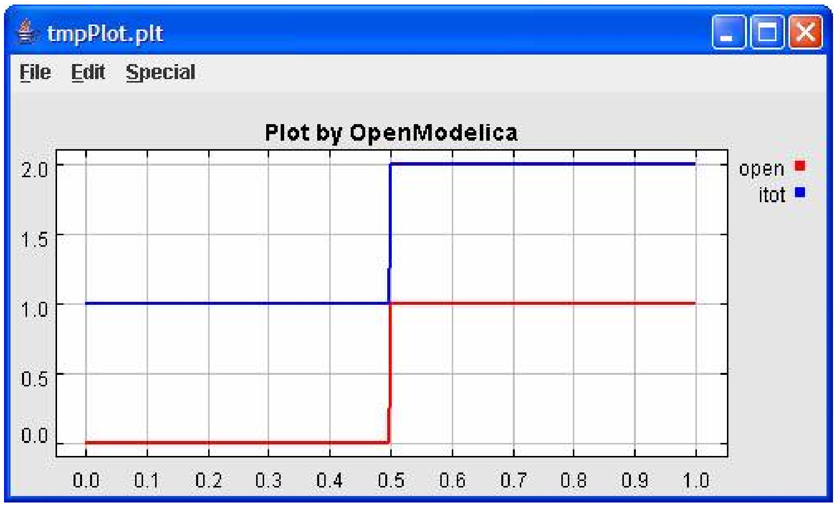
The `runScript` command:

```
>> runScript("sim_BouncingBall.mos")
"true
record
    resultFile = "BouncingBall_res.plt"
end record
true
true"
```



We enter a switch model, to test if-equations (e.g. copy and paste from another file and push enter):

```
>> model switch
  Real v;
  Real i;
  Real i1;
  Real itot;
  Boolean open;
equation
  itot = i + i1;
  if open then
    v = 0;
  else
    i = 0;
  end if;
  1 - i1 = 0;
  1 - v - i = 0;
  open = time >= 0.5;
end switch;
Ok
>> simulate(switch, startTime=0, stopTime=1);
>> plot({itot,open})
true
```



We note that the variable `open` switches from false (0) to true (1), causing `itot` to increase from 1.0 to 2.0.

Now, clear all loaded libraries and models:

```
>> clear()
true
```

List the loaded models – but nothing left:

```
>> list()
""
```

We load another model, the `VanDerPol` model (or via the menu `File->Load Model`):

```
>> loadFile("C:/OpenModelica13/testmodels/VanDerPol.mo"))
```

```
true
```

It is simulated:

```
>> simulate(VanDerPol)
record
    resultFile = "VanDerPol_res.plt"
end record
```

Assign a vector to a variable:

```
>> a:=1:5
{1,2,3,4,5}
```

Type in a function:

```
>> function MySqr input Real x; output Real y; algorithm y:=x*x; end MySqr;
Ok
```

Call the function:

```
>> b:=MySqr(2)
4.0
```

Look at the value of variable `a`:

```
>> a
{1,2,3,4,5}
```

Look at the type of `a`:

```
>> typeOf(a)
"Integer[]"
```

Retrieve the type of `b`:

```
>> typeOf(b)
"Real"
```

What is the type of `MySqr`? Cannot currently be handled.

```
>> typeOf(MySqr)
Error evaluating expr.
```

List the available variables:

```
>> listVariables()
{currentSimulationResult, a, b}
```

Do code instantiation to flat forrm of the `VanDerPol` model:

```
>> instantiateModel(VanDerPol)

"fclass VanDerPol
Real x(start=1.0);
Real y(start=1.0);
parameter Real lambda = 0.3;
equation
  der(x) = y;
  der(y) = -x + lambda * (1.0 - x * x) * y;
end VanDerPol;
"
```

Clear again:

```
>> clear()
true
```

The following is a small example (`ExternalLibraries.mo`) to show the use of external functions:

```
model ExternalLibraries
  Real x(start=1.0),y(start=2.0);
equation
  der(x)=-ExternalFunc1(x);
  der(y)=-ExternalFunc2(y);
end ExternalLibraries;

function ExternalFunc1
  input Real x;
  output Real y;
external
  y=ExternalFunc1_ext(x) annotation(Library="libExternalFunc1_ext.o",
                                    Include="#include \"ExternalFunc1_ext.h\"");
end ExternalFunc1;

function ExternalFunc2
  input Real x;
  output Real y;
  external "C" annotation(Library="libExternalFunc2.a",
                          Include="#include  \"ExternalFunc2.h\"");
end ExternalFunc2;
```

These C (.c) files and header files (.h) are needed:

```
/* file: ExternalFunc1.c */
double ExternalFunc1_ext(double x)
{
  double res;
  res = x+2.0*x*x;
  return res;
}

/* Header file ExternalFunc1_ext.h for ExternalFunc1 function */
double ExternalFunc1_ext(double);

/* file: ExternalFunc2.c */
double ExternalFunc2(double x)
{
  double res;
  res = (x-1.0)*(x+2.0);
  return res;
}

/* Header file ExternalFunc2.h for ExternalFunc2 */
double ExternalFunc2(double);
```

The following script file `ExternalLibraries.mos` will perform everything that is needed, provided you have gcc installed in your path:

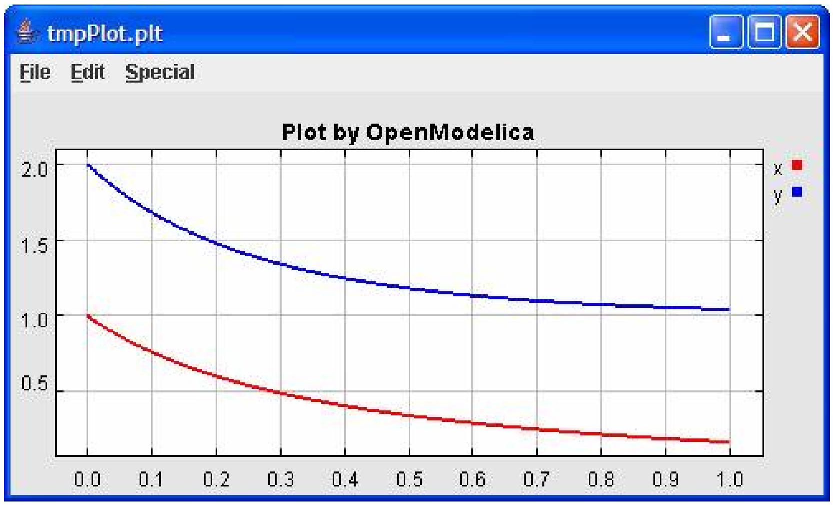
```
loadFile("ExternalLibraries.mo");
system("gcc -c -o libExternalFunc1_ext.o ExternalFunc1.c");
system("gcc -c -o libExternalFunc2.a ExternalFunc2.c");
simulate(ExternalLibraries);
```

We run the script:

```
>> runScript("ExternalLibraries.mos");
```

and plot the results:

```
>> plot({x,y});
```



Leave and quit OpenModelica:

```
>> quit()
```

1.3 Commands for the Interactive Session Handler

The following is the complete list of commands currently available in the interactive session hander.

`simulate(`*modelname*`)`	Translate a model named *modelname* and simulate it.
`simulate(`*modelname*`[,startTime=<`*Real*`>][,stopTime=<`*Real*`>][,numberOfIntervals =<`*Integer*`>])`	Translate and simulate a model, with optional start time, stop time, and optional number of simulation intervals or steps for which the simulation results will be computed. Many steps will give higher time resolution, but occupy more space and take longer to compute. The default number of intervals is 500.
`plot(`*vars*`)`	Plot the variables given as a vector or a scalar, e.g. `plot({x1,x2})` or `plot(x1)`.
`cd()`	Return the current directory.
`cd(`*dir*`)`	Change directory to the directory given as string.
`clear()`	Clear all loaded definitions.
`clearVariables()`	Clear all defined variables.
`instantiateModel(`*modelname*`)`	Performs code instantiation of a model/class and return a string containing the flat class definition.
`list()`	Return a string containing all loaded class definitions.
`list(`*modelname*`)`	Return a string containing the class definition of the named class.
`listVariables()`	Return a vector of the names of the currently defined variables.
`loadModel(`*classname*`)`	Load model or package of name *classname* from `MODELICAPATH`.
`loadFile(`*str*`)`	Load Modelica file (`.mo`) with name given as string argument *str*.
`readFile(`*str*`)`	Load file given as string *str* and return a string containing the file content.

`runScript(`*str*`)`	Execute script file with file name given as string argument *str*.
`system(`*str*`)`	Execute *str* as a system(shell) command in the operating system; return integer success value. Output into stdout from a shell command is put into the console window.
`timing(`*expr*`)`	Evaluate expression *expr* and return the number of seconds (elapsed time) the evaluation took.
`typeOf(`*variable*`)`	Return the type of the *variable* as a string.
`saveModel(`*str*`,`*modelname*`)`	Save the model/class with name *modelname* in the file given by the string argument *str*.
`help()`	Print this helptext (returned as a string).
`quit()`	Leave and quit the OpenModelica environment

Chapter 2

Getting Started with the Graphical Model Editor

This chapter gives a short introduction to graphical modeling. You will learn how to build your own model using the graphical model editor by using the drag-and-drop technique of already developed and freely available components from the Modelica Standard Library.

NOTE: This chapter is a preliminary description which in the near future will be replaced by a separate manual for the Graphical Model Editor.

The Modelica Standard Library is loaded into the OpenModelica environment when the model editor is started and can be browsed using the class browser visible at the left of Figure 2-1 below.

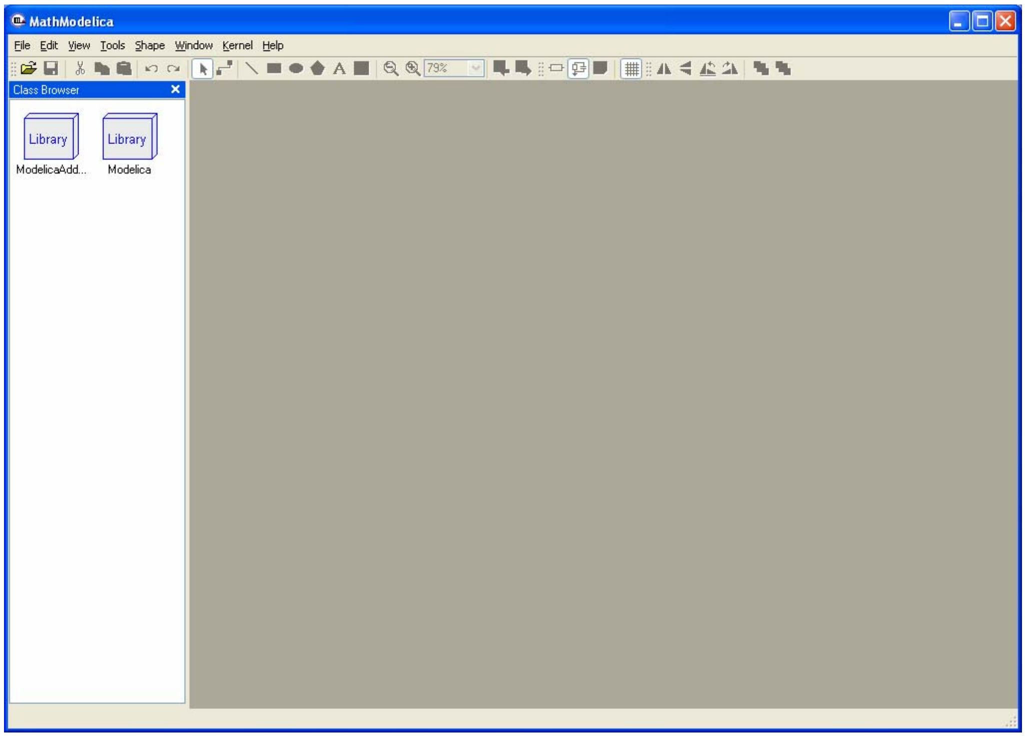


Figure 2-1. The Graphical Model Editor with the class browser to the left, showing icons for the ModelicaAdditions library and the Modelica Standard library.

To open the library, double click on the Modelica package icon in the class browser to the left. As shown by Figure 2-2, the Modelica Standard Library is hierarchically structured into sublibraries.

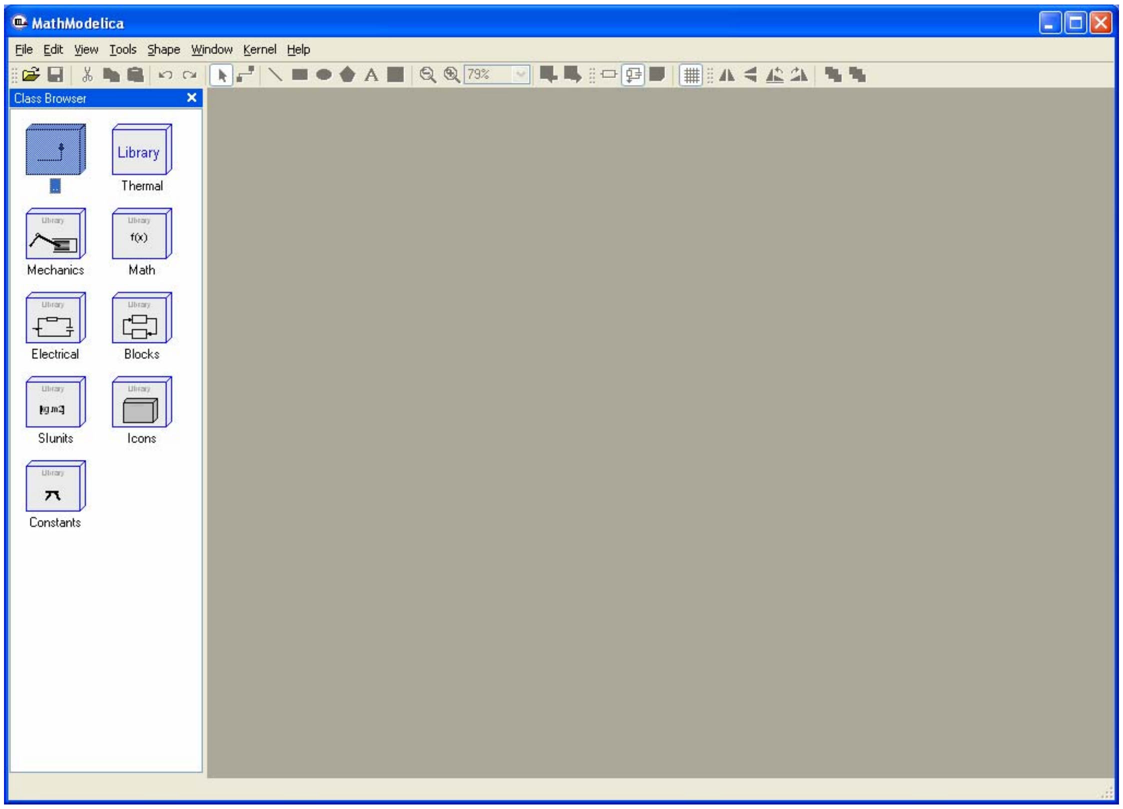


Figure 2-2. The Graphical Model Editor with the class browser showing the Modelica Standard library opened up into sublibraries.

The following list briefly describes the most important sublibraries in the Modelica standard library:

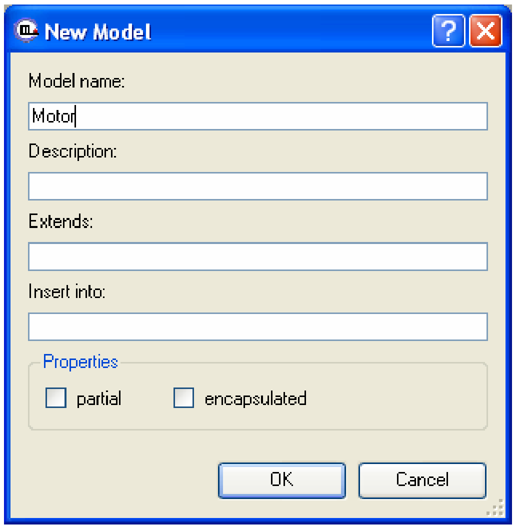
`Thermal`	Components for thermal systems.
`Mechanics`	Mechanical rotational and translational components.
`Math`	Definitions of common mathematical functions, such as sin, cos, and log.
`Electrical`	Common electrical components, such as resistors and transistors.
`Blocks`	Continuous and discrete input/output blocks for use in block diagrams.
`SIunits`	Type definitions with SI standard names and units.
`Icons`	Graphical layout for many component icons
`Constants`	Common constants from mathematics, physics, etc.

2.1 Your First Model

We will introduce the model editor by showing how to build a model of a simple DC motor. Since the DC motor includes both electrical and rotational mechanical components the example also illustrates multi-domain modeling.

2.1.1 Creating a New Model

To create a new model, select `New Model` in the `File` menu. A dialog box will appear, in which you will be able to specify a name of the new model. Enter `Motor` as `Model name`.



Model name: Motor
Description:
Extends:
Insert into:
☐ partial ☐ encapsulated

Figure 2-3. Dialog box for creating a new model.

When clicking on the `OK` button of the dialog box a new window will appear. This window presents different views of the model. A model has two graphical views (Icon and Diagram), and one text view (ModelicaText).

Your new `Motor` model will also appear at the top package level in the class browser. Since no icon for the model has yet been created it is assigned a default icon (a question mark).

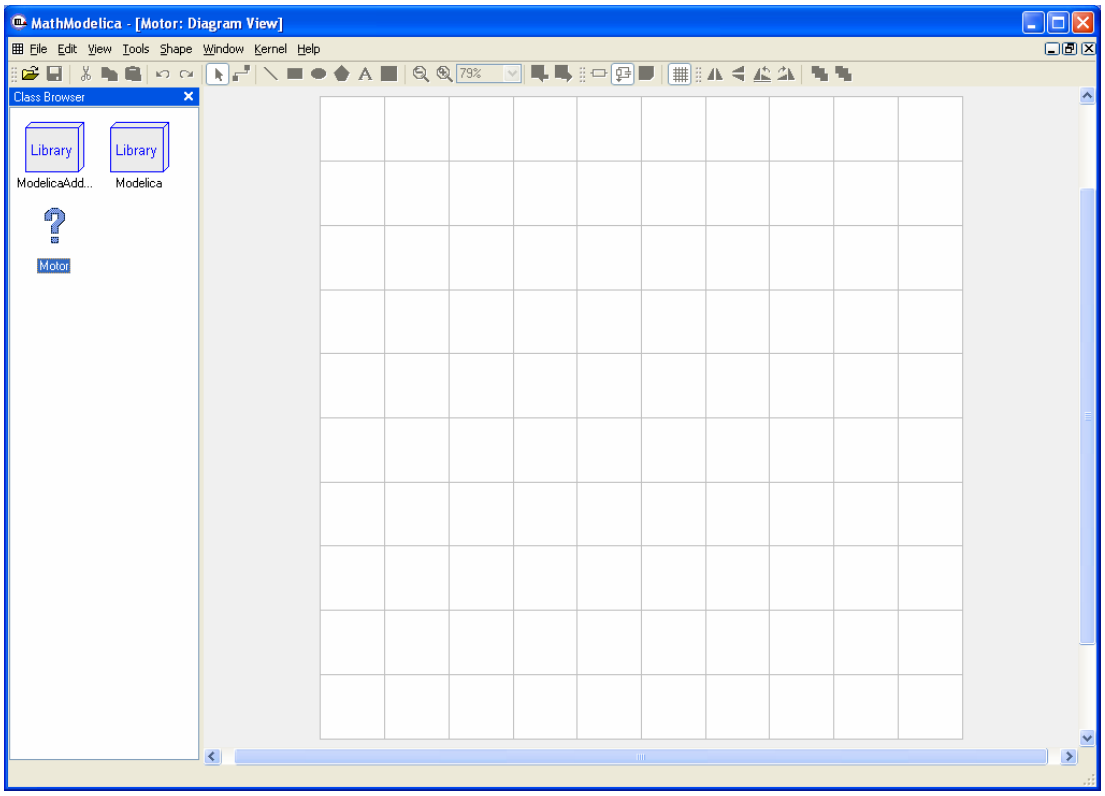

MathModelica - [Motor: Diagram View]
File Edit View Tools Shape Window Kernel Help
79%
Class Browser
Library
ModelicaAdd...
Library
Modelica
Motor

Figure 2-4. The Graphical Model Editor with the new `Motor` model appearing as a question mark icon in the class browser window to the left.

Now you can assemble the DC motor by drag-and-drop of components from the class browser to the diagram view window to the right. The constant voltage source component can be found in the `Modelica.Electrical.Analog.Sources` package whereas the rotational mass representing the motor shaft is located in the `Modelica.Mechanics.Rotational` package. The other electrical components needed are located in the `Modelica.Electrical.Analog.Basic` package.

Components placed in the diagram layer window can be graphically transformed using the mouse and keyboard. To move a component, select it and hold down the left mouse button while moving the mouse. The component will follow the mouse cursor. Release the mouse button when the component is located at the desired position. If more than one component is selected, all of them will be moved simultaneously.

Scaling of components is done using the handles that are visible when a component is selected. Place the mouse cursor over one of the handles, click and hold down the left mouse button while moving the mouse.

Components can also be rotated freely using the handles visible when a component is selected. Place the mouse cursor over one of the handles, click and hold down the left mouse button and the shift button on the keyboard while moving the mouse. The mouse cursor will change its appearance while rotating the component.

Pressing the right mouse button when the mouse cursor is placed over a component brings up a menu with suitable operations.

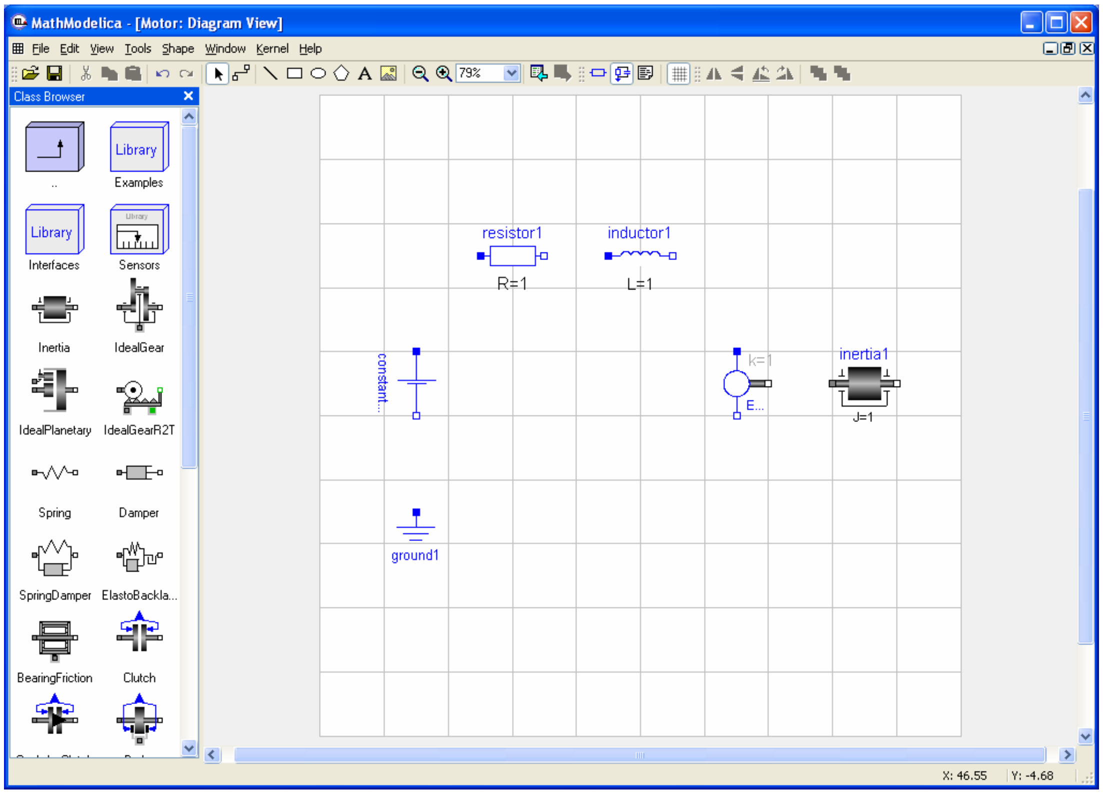
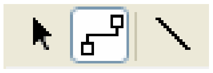


Figure 2-5. The Graphical Model Editor with several components dragged into the diagram view, and several sublibraries and model classes visible in the class browser window to the left.

When the components have been placed on the drawing area, similar to the figure above, you have to draw the lines that connect the components. This is done using the connector tool from the toolbar:



To connect two components, select the connector tool and place the mouse cursor over a connector, i.e., the square symbol on either side of the component. When you are close enough, the mouse cursor will change into a cross. Click and a hold down the left mouse button, drag the cursor to the other connector and then release the mouse button when the mouse cursor turns into a cross. Continue to connect all components until the model diagram resembles the one in Figure 2-6 below.

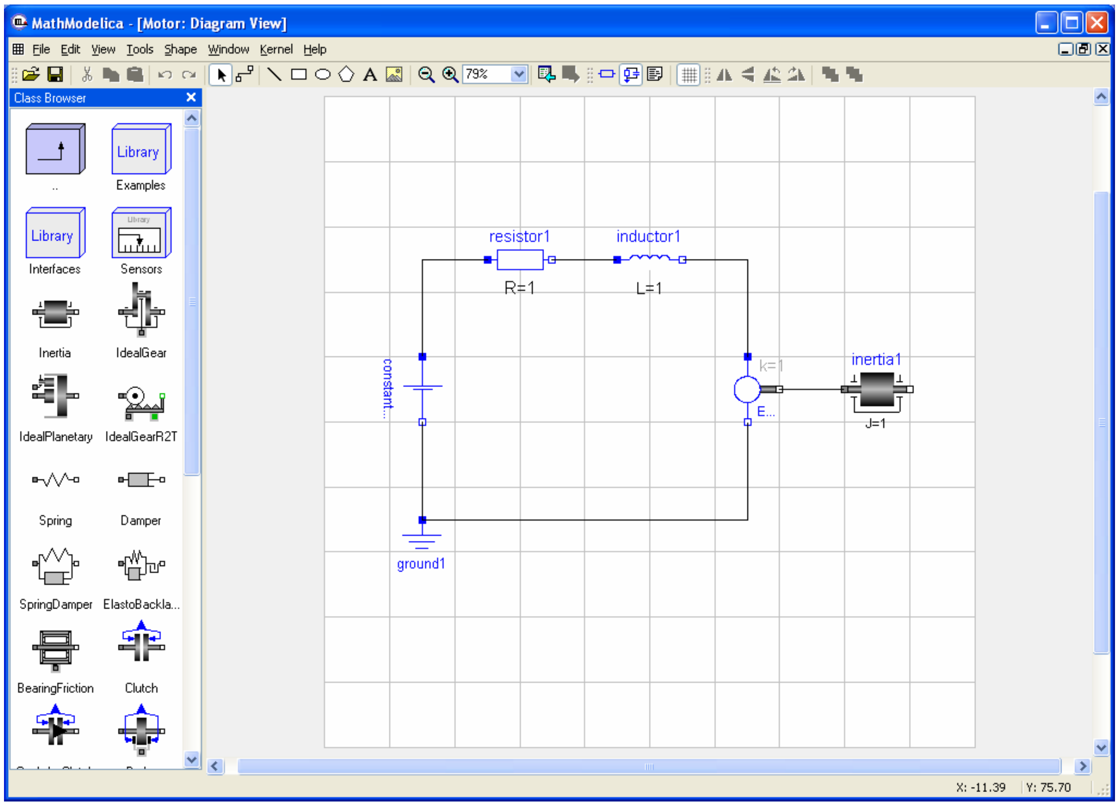


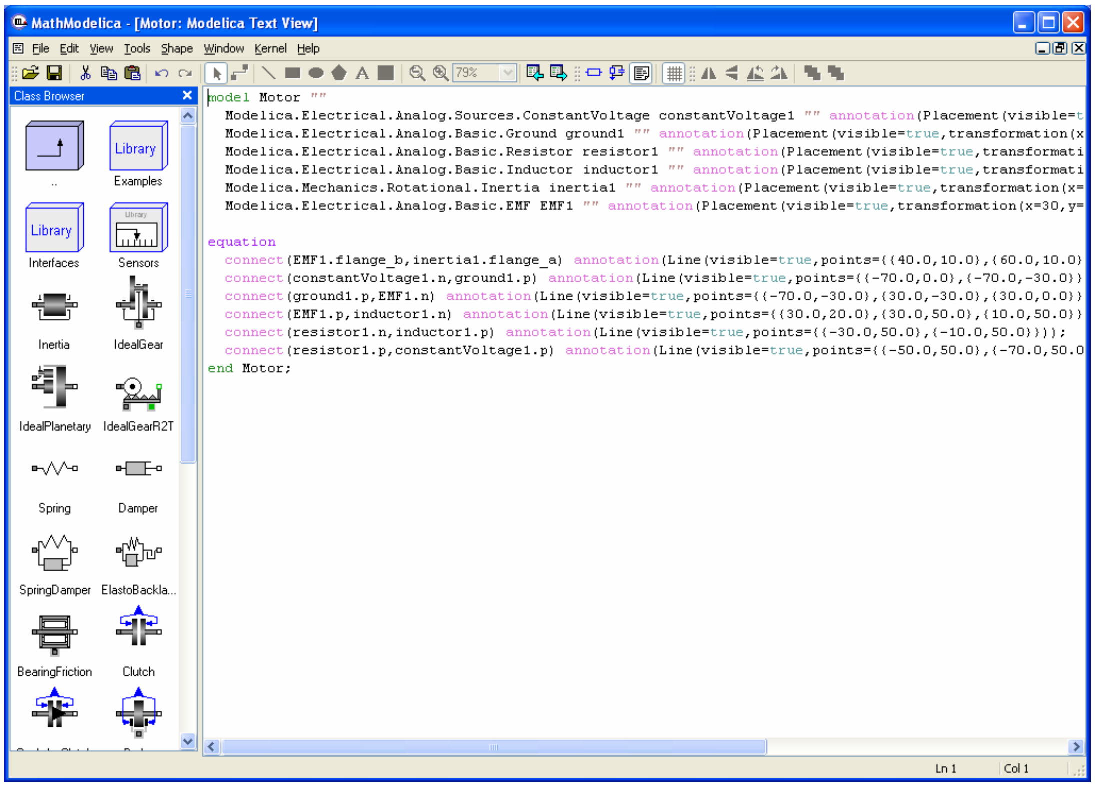
Figure 2-6. The Graphic Model Editor with components connected into a simple DC motor model.

2.1.2 Changing Parameter Values of Components

To change a parameter value of a component, e.g. the resistance of the resistor, we need to switch to the Modelica Text View of the class. Click on the `Modelica Text View` button in the toolbar to switch to text mode:

Whenever we switch from the graphical view to the Modelica Text View after making any changes to the model in the graphical layers we need to update the text view. Click on the `Refresh Class` button, also found in the toolbar:

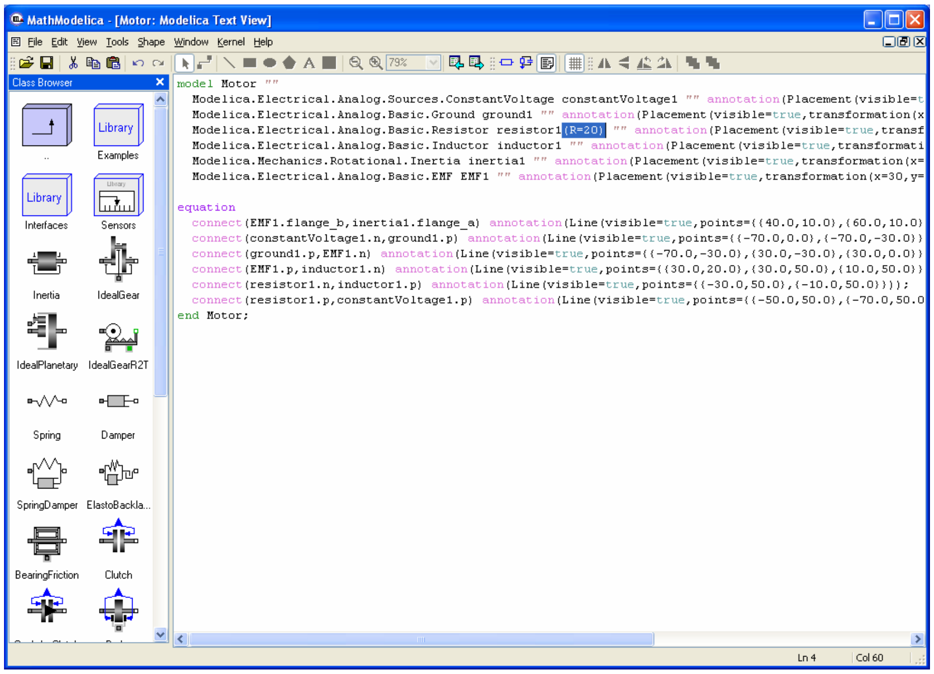
After the text view has been updated it should look similar to the one in Figure 2-7. The order of the component declarations and connection equations depend on in which order you placed them on the drawing area and connected them.



```
model Motor ""
  Modelica.Electrical.Analog.Sources.ConstantVoltage constantVoltage1 "" annotation(Placement(visible=t
  Modelica.Electrical.Analog.Basic.Ground ground1 "" annotation(Placement(visible=true,transformation(x
  Modelica.Electrical.Analog.Basic.Resistor resistor1 "" annotation(Placement(visible=true,transformati
  Modelica.Electrical.Analog.Basic.Inductor inductor1 "" annotation(Placement(visible=true,transformati
  Modelica.Mechanics.Rotational.Inertia inertia1 "" annotation(Placement(visible=true,transformation(x=
  Modelica.Electrical.Analog.Basic.EMF EMF1 "" annotation(Placement(visible=true,transformation(x=30,y=

equation
  connect(EMF1.flange_b,inertia1.flange_a) annotation(Line(visible=true,points={{40.0,10.0},{60.0,10.0}
  connect(constantVoltage1.n,ground1.p) annotation(Line(visible=true,points={{-70.0,0.0},{-70.0,-30.0}}
  connect(ground1.p,EMF1.n) annotation(Line(visible=true,points={{-70.0,-30.0},{30.0,-30.0},{30.0,0.0}}
  connect(EMF1.p,inductor1.n) annotation(Line(visible=true,points={{30.0,20.0},{30.0,50.0},{10.0,50.0}}
  connect(resistor1.n,inductor1.p) annotation(Line(visible=true,points={{-30.0,50.0},{-10.0,50.0}}));
  connect(resistor1.p,constantVoltage1.p) annotation(Line(visible=true,points={{-50.0,50.0},{-70.0,50.0
end Motor;
```

Figure 2-7. The Graphical Model Editor with the Modelica Text View mode showing the new `Motor` model. This view also allows text editing of the model.

Locate the line that starts with `Modelica.Electrical.Analog.Basic.Resistor`. Put the text cursor right after resistor1 and add the text (`R=20`). This text is a modification of the resistor component, changing its parameter `R` (resistance) from the default value of 1 to 20. The Modelica Text View should now look similar to the one found in Figure 2-8 below.



```
model Motor ""
  Modelica.Electrical.Analog.Sources.ConstantVoltage constantVoltage1 "" annotation(Placement(visible=t
  Modelica.Electrical.Analog.Basic.Ground ground1 "" annotation(Placement(visible=true,transformation(x
  Modelica.Electrical.Analog.Basic.Resistor resistor1(R=20) "" annotation(Placement(visible=true,transf
  Modelica.Electrical.Analog.Basic.Inductor inductor1 "" annotation(Placement(visible=true,transformati
  Modelica.Mechanics.Rotational.Inertia inertia1 "" annotation(Placement(visible=true,transformation(x=
  Modelica.Electrical.Analog.Basic.EMF EMF1 "" annotation(Placement(visible=true,transformation(x=30,y=

equation
  connect(EMF1.flange_b,inertia1.flange_a) annotation(Line(visible=true,points={{40.0,10.0},{60.0,10.0}
  connect(constantVoltage1.n,ground1.p) annotation(Line(visible=true,points={{-70.0,0.0},{-70.0,-30.0}}
  connect(ground1.p,EMF1.n) annotation(Line(visible=true,points={{-70.0,-30.0},{30.0,-30.0},{30.0,0.0}}
  connect(EMF1.p,inductor1.n) annotation(Line(visible=true,points={{30.0,20.0},{30.0,50.0},{10.0,50.0}}
  connect(resistor1.n,inductor1.p) annotation(Line(visible=true,points={{-30.0,50.0},{-10.0,50.0}}));
  connect(resistor1.p,constantVoltage1.p) annotation(Line(visible=true,points={{-50.0,50.0},{-70.0,50.0
end Motor;
```

Figure 2-8. The Modelica Text View of the Motor model, with a modifier `R=20` added for the `resistor1` component.

When you make any changes to the model in the Modelica Text View you need to click on the `Apply Class Definition` button in the toolbar to confirm the changes:

When you have done this, change class view back to the Diagram View and study the icon of the resistor component. It should show `R=20` instead of `R=1`, i.e., the resistance of the resistor is now 20 ohm.

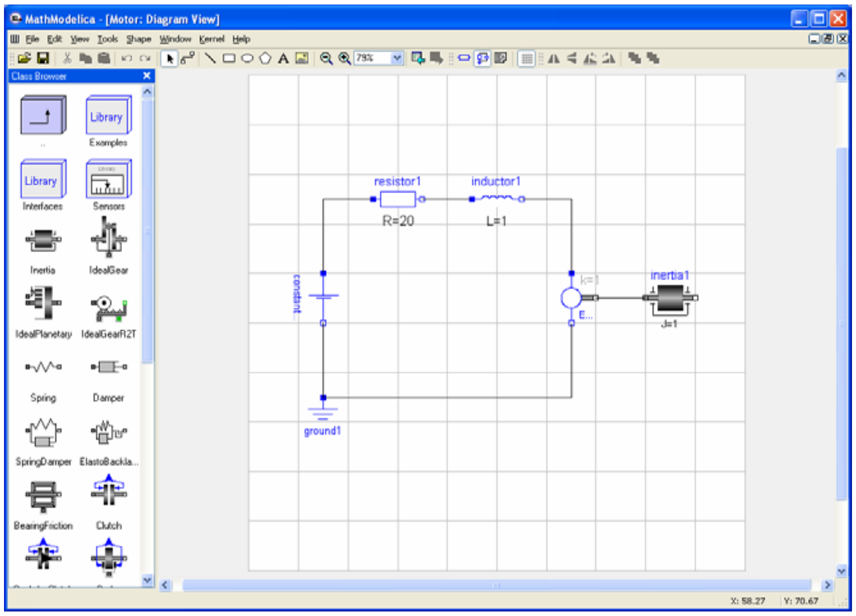
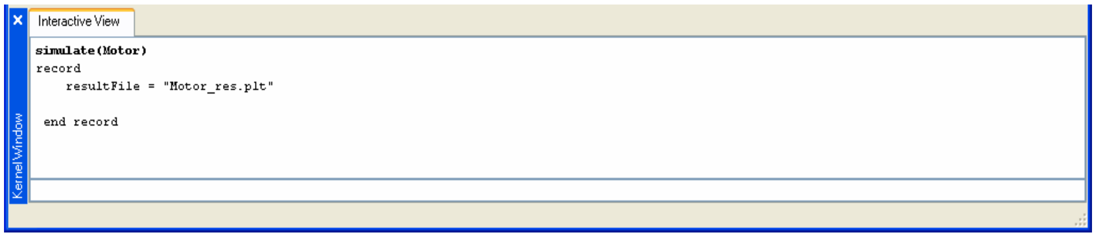


Figure 2-9. The Diagram View showing `R=20` for the `resistor1` component.

2.1.3 Translating and Simulating Using the Interactive View

Translating and simulating is performed using the Interactive View of the simulation kernel window. To open the kernel window, select `Open Kernel Window` in the File menu. Enter the command `simulate(Motor)` in the interactive view to translate and simulate the model.



```
simulate(Motor)
record
    resultFile = "Motor_res.plt"

 end record
```

Figure 2-10. The interactive view of the simulation kernel window, with the simulation command `simulate(Motor)`.

2.1.4 Plotting

After the model has been translated and simulated, any of its variables can be plotted using the `plot` command. Giving the command `plot({inductor1.w, inertia1.a})` will bring up the window below.

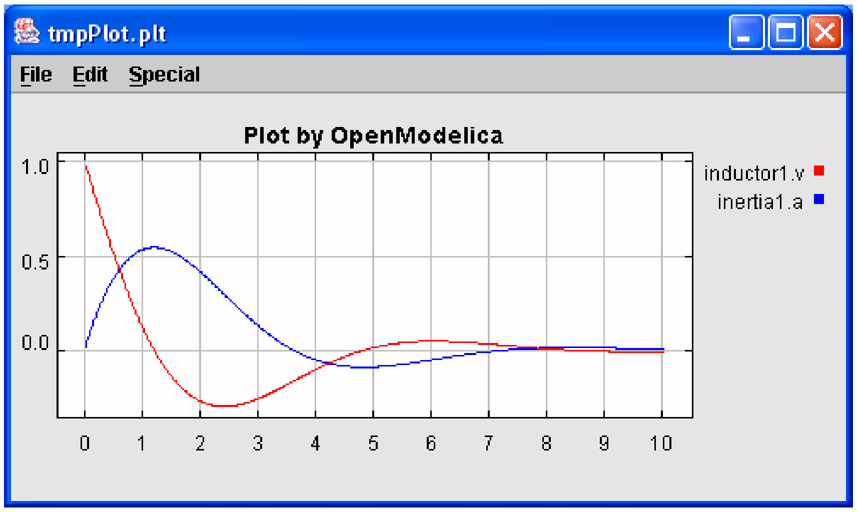


Figure 2-11. OpenModelica plot window created by the command: `plot({inductor1.w, inertia1.a})`, after a simulation of the `Motor` model.

2.1.5 Saving and Loading

It is possible to save a model from the model editor. Saving a model will create a Modelica 2.0 standard output file with the extension `.mo`. This file can later be loaded back into the model editor.

2.2 Keyboard Command Shortcuts

The following are keyboard shortcut commands available in the different windows of the Graphical model editor, including the text editing window.

2.2.1 Class Window – Common Shortcuts

`CTRL + O`	Open a Modelica (.mo) file.
`CTRL + Q`	Quit the model editor.

2.2.2 Class Browser Window

`LEFT ARROW`	Select the class to the left of the currently selected class.
`RIGHT ARROW`	Select the class to the right of the currently selected class.
`UP ARROW`	Select the class above the currently selected class.
`DOWN ARROW`	Select the class below the currently selected class.
`ENTER`	If the selected class is a package; open the package and view its contents, otherwise; open the class for editing.
`DELETE`	Delete the selected class.

2.2.3 Class Window – Graphical Layer View

`CTRL + S`	Save class present in active class window.
`CTRL + Z`	Undo last undo/redo supported operation.
`CTRL + Y`	Redo last undo/redo supported operation.
`CTRL + A`	Select all items.
`DELETE`	Delete the selected items.
`CTRL + W`	Zoom the view to fit the size of the window.
`F5`	Switch to a full screen view of the graphical layer.
`ESC`	Deselect all items, or if the Full Screen view is activated, return to normal view.
`SPACE`	Pointer tool.
`C`	Connector tool.
`L`	Line tool.
`R`	Rectangle tool.
`E`	Ellipse tool.
`P`	Polygon tool.
`T`	Text tool.
`B`	Bitmap tool.
`+`	Zoom In tool.

`-`	Zoom Out tool.
`LEFT ARROW`	Move selected items left, otherwise if no selection move view area left.
`RIGHT ARROW`	Move selected items right, otherwise if no selection move view area right.
`UP ARROW`	Move selected items up, otherwise if no selection move view area up.
`DOWN ARROW`	Move selected items down, otherwise if no selection move view area down.
`CTRL + LEFT ARROW`	Move view area to the left end of the diagram.
`CTRL + RIGHT ARROW`	Move view area to the right end of the diagram.
`CTRL + UP ARROW`	Move view area to the top of the diagram.
`CTRL + DOWN ARROW`	Move view area to the bottom of the diagram.
`SHIFT + LEFT ARROW`	Small move of selected items left; if no selection move view area left.
`SHIFT + RIGHT ARROW`	Small move of selected items right; if no selection move view area right.
`SHIFT + UP ARROW`	Small move of selected items up; if no selection move view area up.
`SHIFT + DOWN ARROW`	Small move of selected items down; if no selection move view area down.
`PAGE UP`	Move view area up.
`PAGE DOWN`	Move view area down.
`SHIFT + PAGE UP`	Move view area left.
`SHIFT + PAGE DOWN`	Move view area right.
`HOME`	Move view area to the upper left corner.
`END`	Move view area to the lower left corner.
`CTRL + HOME`	Move view area to the upper right corner.
`CTRL + END`	Move view area to the lower right corner.
`CTRL + L`	Rotate the selected items 90º to the left (anti-clockwise).
`CTRL + R`	Rotate the selected items 90º to the right (clockwise).
`CTRL + H`	Flip the selected items horizontally.
`CTRL + J`	Flip the selected items vertically.

2.2.4 Class Window – Modelica Text View

`CTRL + P`	Print the text.
`CTRL + A`	Select all text.
`CTRL + Z`	Undo last edit.
`CTRL + Y`	Redo last edit.
`CTRL + X`	Cut the selected text to the Clipboard.
`CTRL + C`	Copy the selected text to the Clipboard.
`CTRL + V`	Paste Clipboard contents into the text editor.
`LEFT ARROW`	Move one character left.
`RIGHT ARROW`	Move one character right.
`UP ARROW`	Move one line up.
`DOWN ARROW`	Move one line down.

`PAGE UP`	Move one page up.
`PAGE DOWN`	Move one page up.
`HOME`	Move to the beginning of the line.
`END`	Move to the end of the line.
`CTRL + HOME`	Move to the beginning of the text.
`CTRL + END`	Move to the end of the text.
`DELETE`	Delete the selected text, or if no text is selected, delete the character to the right.

2.2.5 Interative Kernel Window – Output View

`CTRL + C`	Copy the selected text to the Clipboard.
`CTRL + A`	Select all text.

2.2.6 Interactive Kernel Window – Input Field

`CTRL + X`	Cut the selected text to the Clipboard.
`CTRL + C`	Copy the selected text to the Clipboard.
`CTRL + V`	Paste Clipboard contents into the text field.
`CTRL + Z`	Undo the last operation.
`CTRL + Y`	Redo the last operation.
`DELETE`	Delete the selected text, or if no text is selected, delete the character to the right.
`CTRL + A`	Select all text.
`ENTER`	Evaluate given command.
`UP ARROW`	Step backward in the history of commands.
`DOWN ARROW`	Step forward in the history of commands.

Chapter 3

DrModelica Notebook and Model Editor

This chapter covers the OpenModelica electronic notebook and model editor subsystem, together with the DrModelica tutoring system for teaching Modelica, which is based on such notebooks.

However, the OpenModelica notebook facility is work in progress, which currently is only partially completed (see Section 0). For these reasons we first present the electronic notebook facility and DrModelica based on the MathModelica implementation. The OpenModelica electronic notebooks is a simplified version of those notebooks, that however still are able to handle the full DrModelica system.

3.1 Interactive Notebooks with Literate Programming

Interactive Electronic Notebooks are active documents that may contain technical computations and text, as well as graphics. Hence, these documents are suitable to be used for teaching and experimentation, simulation scripting, model documentation and storage, etc.

The Notebook facility is actually an interactive WYSIWYG (What-You-See-Is-What-You-Get) realization of Literate Programming, a form of programming where programs are integrated with documentation in the same document.

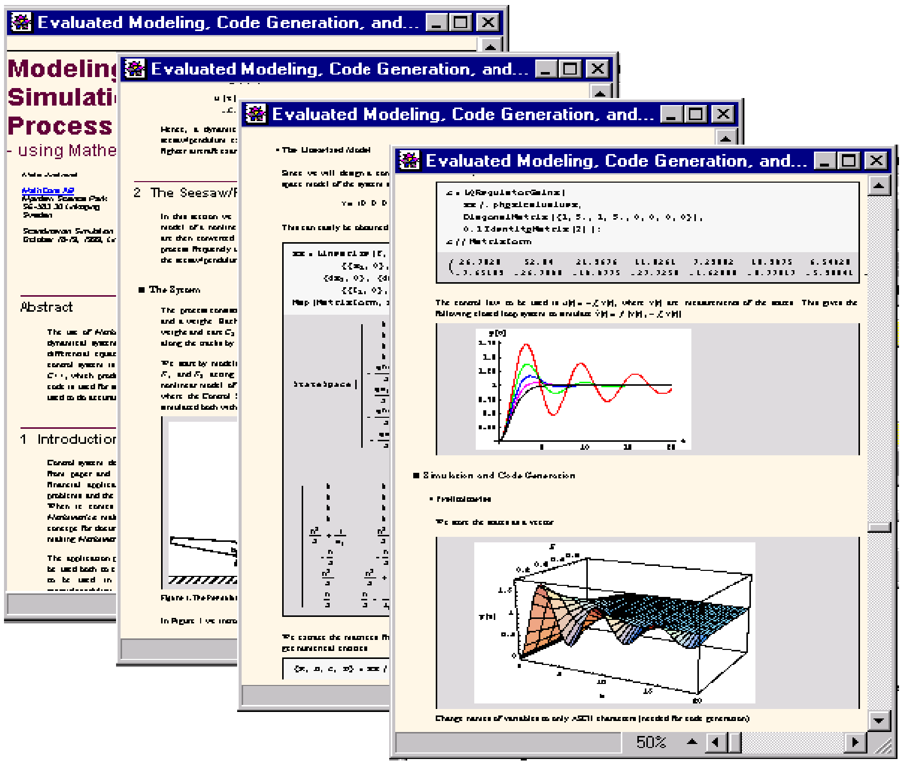
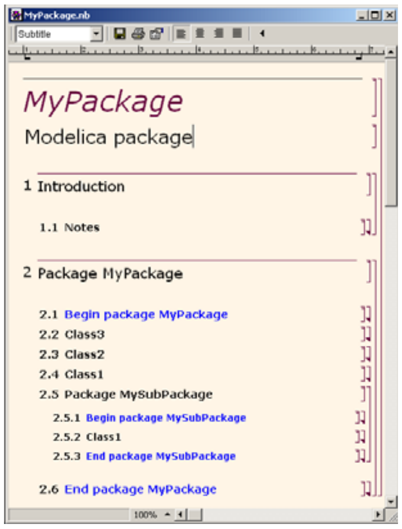


Figure 3-1. Examples of notebooks in the MathModelica modeling and simulation environment.

3.1.1 Tree Structured Hierarchical Document Representation

Traditional documents, e.g. books and reports, essentially always have a hierarchical structure. They are divided into sections, subsections, paragraphs, etc. Both the document itself and its sections usually have headings as labels for easier navigation. This kind of structure is also reflected in electronic notebooks. Every notebook corresponds to one document (one file) and contains a tree structure of cells. A cell can have different kinds of contents, and can even contain other cells. The notebook hierarchy of cells thus reflects the hierarchy of sections and subsections in a traditional document such as a book.


MyPackage.nb

Subtitle

MyPackage

Modelica package

1 Introduction

1.1 Notes

2 Package MyPackage

2.1 Begin package MyPackage

2.2 Class3

2.3 Class2

2.4 Class1

2.5 Package MySubPackage

2.5.1 Begin package MySubPackage

2.5.2 Class1

2.5.3 End package MySubPackage

2.6 End package MyPackage

100%

(a)

```
package MyPackage
  model class3
  ...
  end class3;
  model class2 ...
  model class1 ...
  package MySubPackage
    model class1
     ...
    end class1;
  end MySubPackage;
end MyPackage;
```

(b)

Figure 3-2. The package `MyPackage` in a notebook (a) and as Modelica text (b).

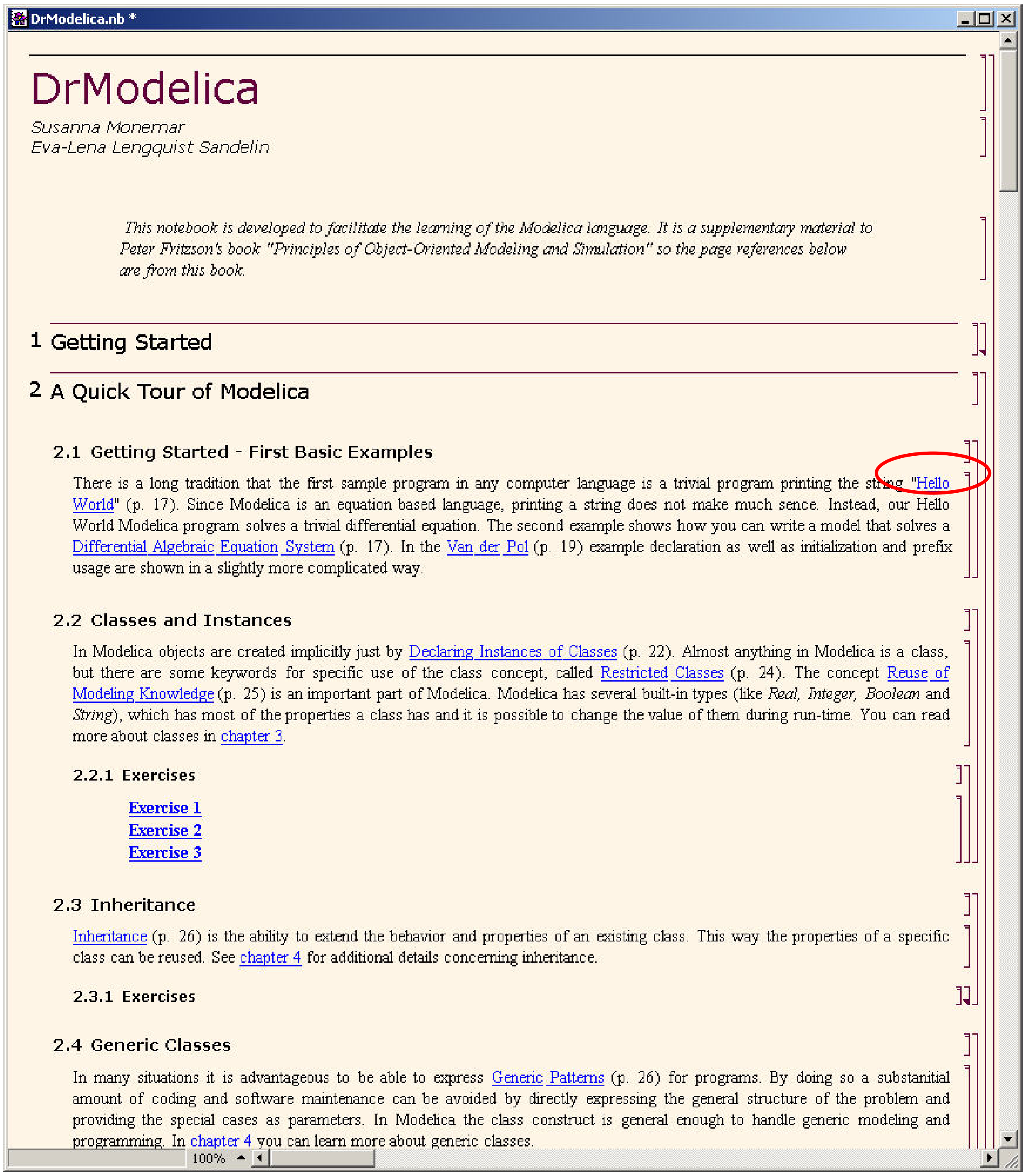
Modelica packages including documentation and test cases can also be stored as notebooks, e.g. as in Figure 3-1 or Figure 3-2. Those cells that contain Modelica model classes intended to be used from other models, e.g. library components or certain application models, can be marked as Modelica code cells. This means that it is possible to export the Modelica cells in the notebook `MyPackage.nb` of Figure 3-2a, into a file `MyPackage.mo` with the contents shown in Figure 3-2b.

3.2 The DrModelica Tutoring System

Understanding programs is hard, especially code written by someone else. For educational purposes it is essential to be able to show the source code and to give an explanation of it at the same time.

Moreover, it is important to show the result of the source code's execution. In modeling and simulation it is also important to have the source code, the documentation about the source code, the execution results of the simulation model, and the documentation of the simulation results in the same document. The reason is that the problem solving process in computational simulation is an iterative process that often requires a modification of the original mathematical model and its software implementation after the interpretation and validation of the computed results corresponding to an initial model.

Most of the environments associated with equation-based modeling languages focus more on providing efficient numerical algorithms rather than giving attention to the aspects that should facilitate the learning and teaching of the language. There is a need for an environment facilitating the learning and understanding of Modelica. These are the reasons for developing the DrModelica teaching material for Modelica and for teaching modeling and simulation.



DrModelica.nb *

DrModelica

Susanna Monemar
Eva-Lena Lengquist Sandelin

This notebook is developed to facilitate the learning of the Modelica language. It is a supplementary material to Peter Fritzson's book "Principles of Object-Oriented Modeling and Simulation" so the page references below are from this book.

1 Getting Started

2 A Quick Tour of Modelica

2.1 Getting Started - First Basic Examples

There is a long tradition that the first sample program in any computer language is a trivial program printing the string "Hello World" (p. 17). Since Modelica is an equation based language, printing a string does not make much sence. Instead, our Hello World Modelica program solves a trivial differential equation. The second example shows how you can write a model that solves a Differential Algebraic Equation System (p. 17). In the Van der Pol (p. 19) example declaration as well as initialization and prefix usage are shown in a slightly more complicated way.

2.2 Classes and Instances

In Modelica objects are created implicitly just by Declaring Instances of Classes (p. 22). Almost anything in Modelica is a class, but there are some keywords for specific use of the class concept, called Restricted Classes (p. 24). The concept Reuse of Modeling Knowledge (p. 25) is an important part of Modelica. Modelica has several built-in types (like *Real, Integer, Boolean* and *String*), which has most of the properties a class has and it is possible to change the value of them during run-time. You can read more about classes in chapter 3.

2.2.1 Exercises

Exercise 1
Exercise 2
Exercise 3

2.3 Inheritance

Inheritance (p. 26) is the ability to extend the behavior and properties of an existing class. This way the properties of a specific class can be reused. See chapter 4 for additional details concerning inheritance.

2.3.1 Exercises

2.4 Generic Classes

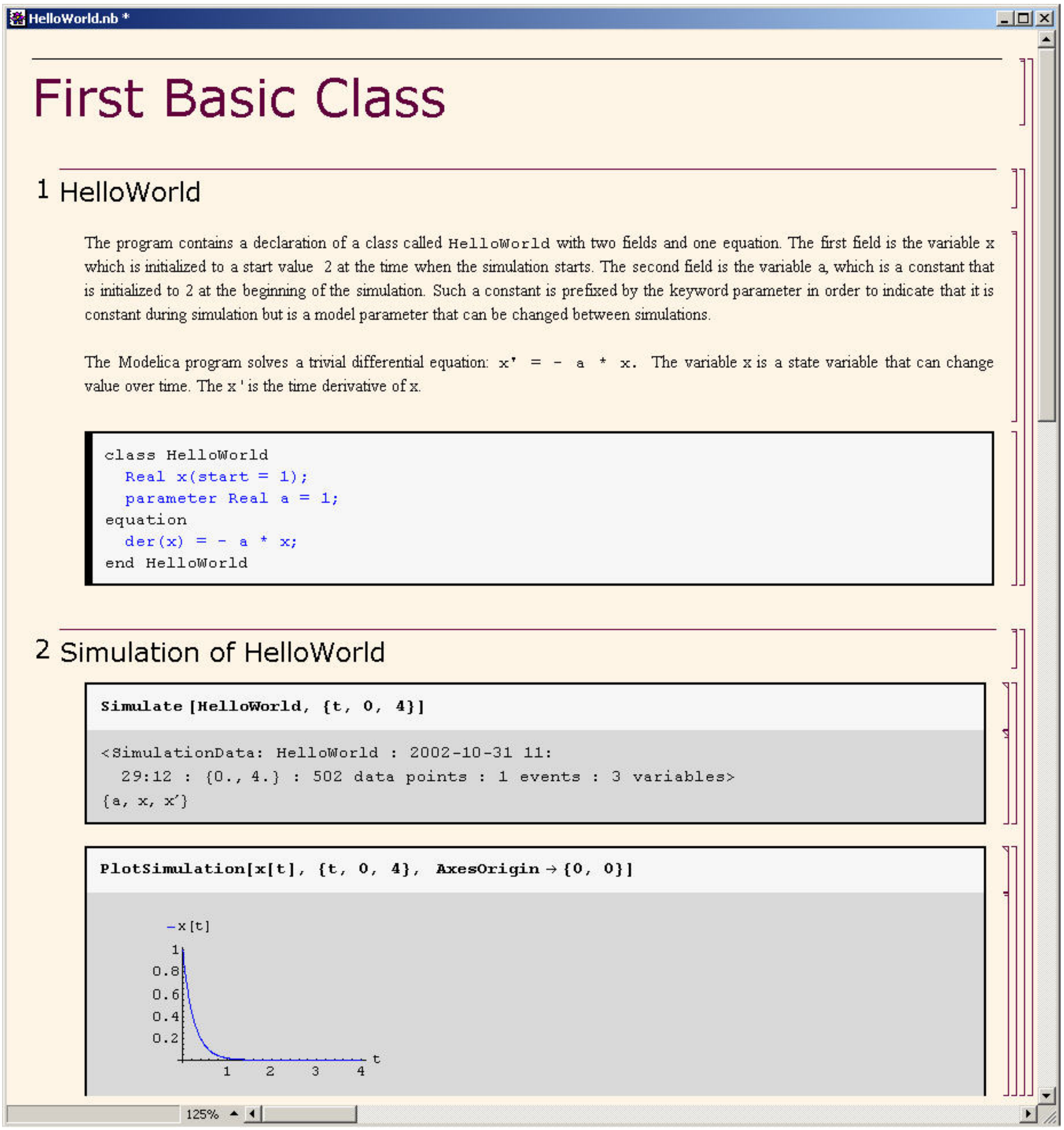
In many situations it is advantageous to be able to express Generic Patterns (p. 26) for programs. By doing so a substanitial amount of coding and software maintenance can be avoided by directly expressing the general structure of the problem and providing the special cases as parameters. In Modelica the class construct is general enough to handle generic modeling and programming. In chapter 4 you can learn more about generic classes.

100%

Figure 3-3. The front-page notebook of the DrModelica tutoring system.

DrModelica has a hierarchical structure represented as notebooks. The front-page notebook is similar to a table of contents that holds all other notebooks together by providing links to them. This particular notebook is the first page the user will see (Figure 3-3).

In each chapter of DrModelica the user is presented a short summary of the corresponding chapter of the book "Principles of Object-Oriented Modeling and Simulation with Modelica 2.1" by Peter Fritzson. The summary introduces some *keywords*, being hyperlinks that will lead the user to other notebooks describing the keywords in detail.



First Basic Class

1 HelloWorld

The program contains a declaration of a class called HelloWorld with two fields and one equation. The first field is the variable x which is initialized to a start value 2 at the time when the simulation starts. The second field is the variable a, which is a constant that is initialized to 2 at the beginning of the simulation. Such a constant is prefixed by the keyword parameter in order to indicate that it is constant during simulation but is a model parameter that can be changed between simulations.

The Modelica program solves a trivial differential equation: x' = - a * x. The variable x is a state variable that can change value over time. The x ' is the time derivative of x.

```
class HelloWorld
  Real x(start = 1);
  parameter Real a = 1;
equation
  der(x) = - a * x;
end HelloWorld
```

2 Simulation of HelloWorld

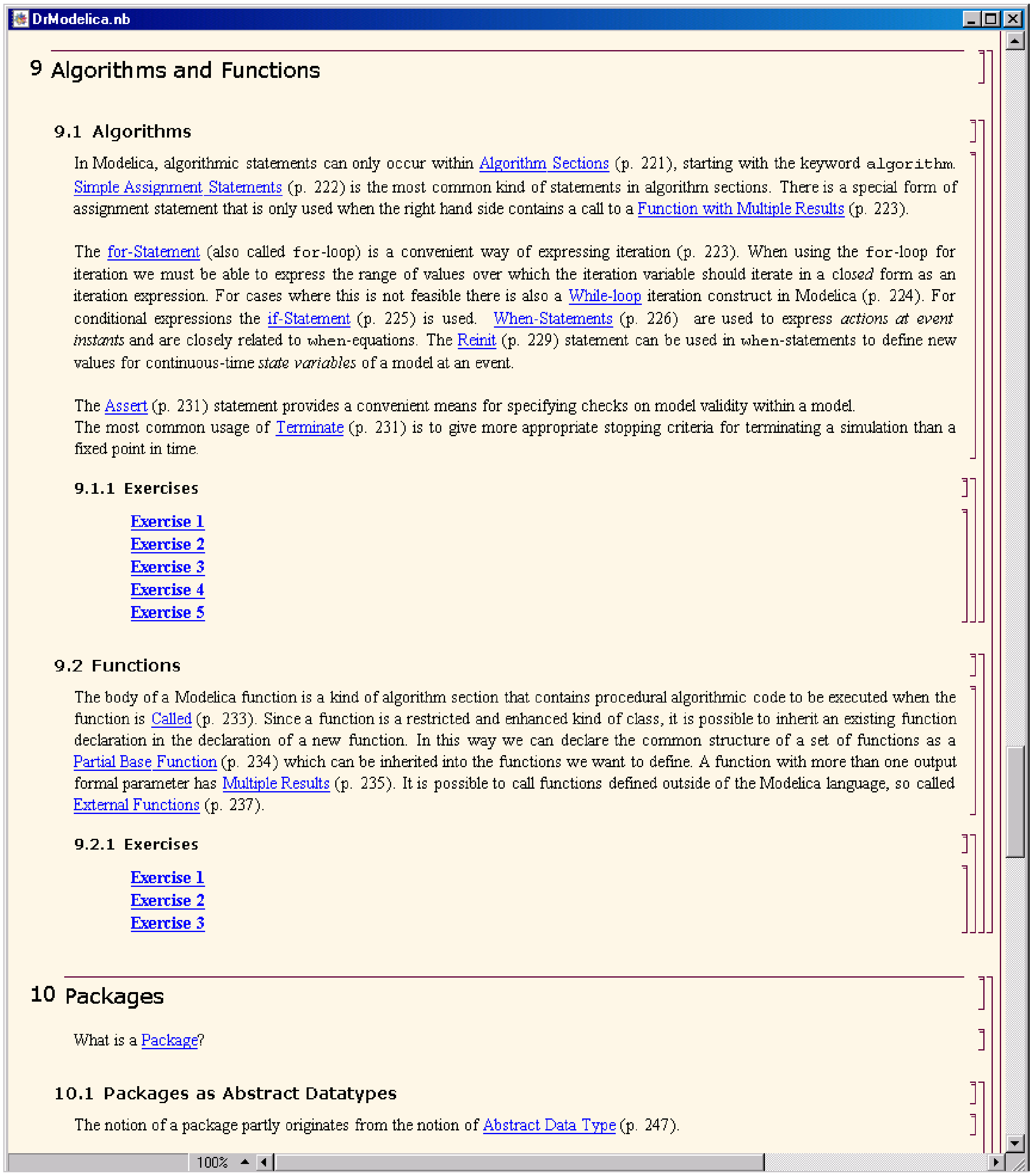
```
Simulate[HelloWorld, {t, 0, 4}]

<SimulationData: HelloWorld : 2002-10-31 11:
  29:12 : {0., 4.} : 502 data points : 1 events : 3 variables>
{a, x, x′}
```

```
PlotSimulation[x[t], {t, 0, 4}, AxesOrigin → {0, 0}]
```

Figure 3-4. The `HelloWorld` class simulated and plotted using the MathModelica version of DrModelica.

Now, let us consider that the link "*HelloWorld*" in DrModelica Section 2.1 in **Error! Reference source not found.** is clicked by the user. The new notebook, to which the user is being linked (see Figure 3-4), is not only a textual description but also contains one or more examples explaining the specific keyword. In this class, `HelloWorld`, a differential equation is specified.

No information in a notebook is fixed, which implies that the user can add, change, or remove anything in a notebook. Alternatively, the user can create an entirely new notebook in order to write his/her own programs or copy examples from other notebooks. This new notebook can be linked from existing notebooks.



9 Algorithms and Functions

9.1 Algorithms

In Modelica, algorithmic statements can only occur within Algorithm Sections (p. 221), starting with the keyword `algorithm`. Simple Assignment Statements (p. 222) is the most common kind of statements in algorithm sections. There is a special form of assignment statement that is only used when the right hand side contains a call to a Function with Multiple Results (p. 223).

The for-Statement (also called `for`-loop) is a convenient way of expressing iteration (p. 223). When using the `for`-loop for iteration we must be able to express the range of values over which the iteration variable should iterate in a *closed* form as an iteration expression. For cases where this is not feasible there is also a While-loop iteration construct in Modelica (p. 224). For conditional expressions the if-Statement (p. 225) is used. When-Statements (p. 226) are used to express *actions at event instants* and are closely related to `when`-equations. The Reinit (p. 229) statement can be used in `when`-statements to define new values for continuous-time *state variables* of a model at an event.

The Assert (p. 231) statement provides a convenient means for specifying checks on model validity within a model.
The most common usage of Terminate (p. 231) is to give more appropriate stopping criteria for terminating a simulation than a fixed point in time.

9.1.1 Exercises

Exercise 1
Exercise 2
Exercise 3
Exercise 4
Exercise 5

9.2 Functions

The body of a Modelica function is a kind of algorithm section that contains procedural algorithmic code to be executed when the function is Called (p. 233). Since a function is a restricted and enhanced kind of class, it is possible to inherit an existing function declaration in the declaration of a new function. In this way we can declare the common structure of a set of functions as a Partial Base Function (p. 234) which can be inherited into the functions we want to define. A function with more than one output formal parameter has Multiple Results (p. 235). It is possible to call functions defined outside of the Modelica language, so called External Functions (p. 237).

9.2.1 Exercises

Exercise 1
Exercise 2
Exercise 3

10 Packages

What is a Package?

10.1 Packages as Abstract Datatypes

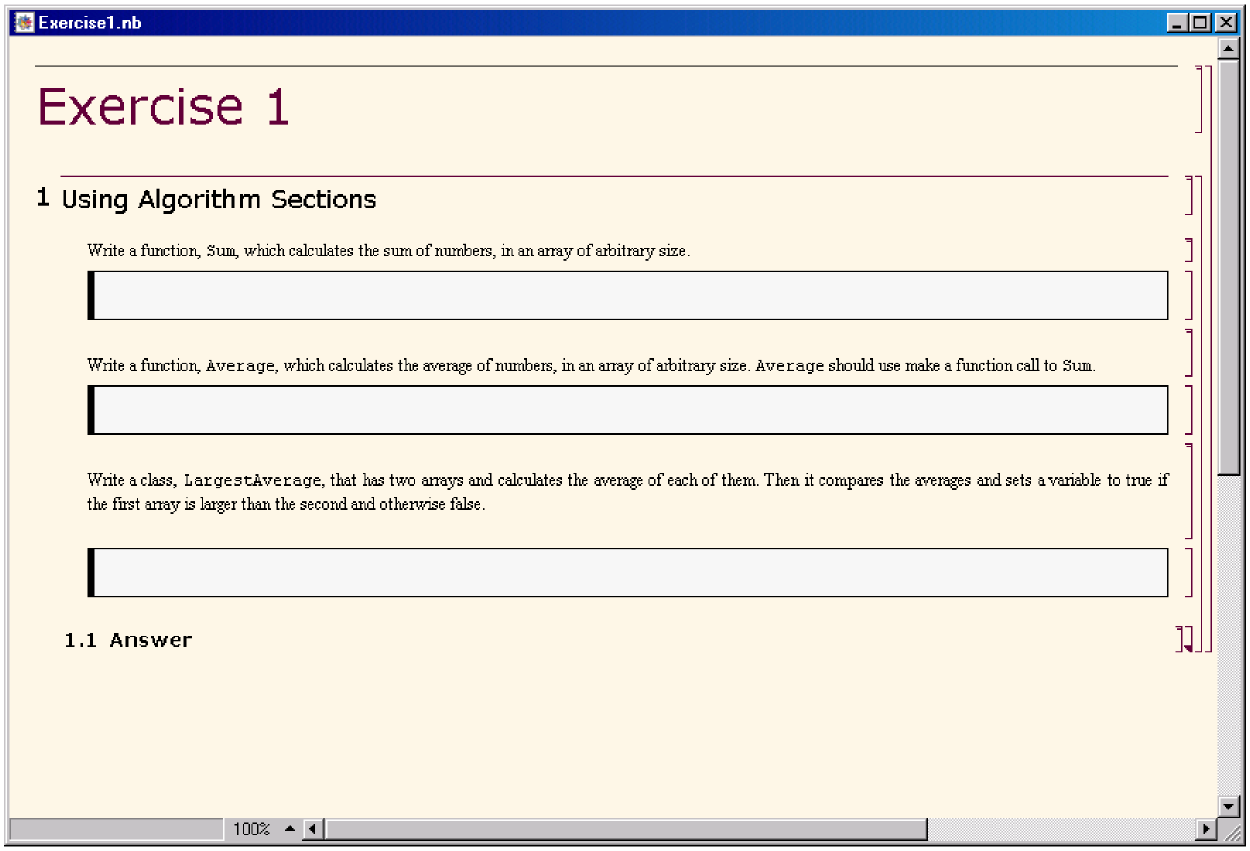
The notion of a package partly originates from the notion of Abstract Data Type (p. 247).

Figure 3-5. DrModelica Chapter 9 in the main page of the MathModelica version of DrModelica.

When a class has been successfully evaluated the user can simulate and plot the result, as depicted in Figure 3-4 for the simple `HelloWorld` example model..

After reading a chapter in DrModelica the user can immediately practice the newly acquired information by doing the exercises that concern the specific chapter. Exercises have been written in order to elucidate language constructs step by step based on the pedagogical assumption that a student learns better *"using the strategy of learning by doing"*. The exercises consist of either theoretical questions or practical programming assignments. All exercises provide answers in order to give the user immediate feedback.

Figure 3-5 shows Chapter 9 of the DrModelica teaching material. Here the user can read about language constructs, like `algorithm` sections, when-statements, and `reinit` equations, and then practice these constructs by solving the exercises corresponding to the recently studied section.



Exercise 1

1 Using Algorithm Sections

Write a function, Sum, which calculates the sum of numbers, in an array of arbitrary size.

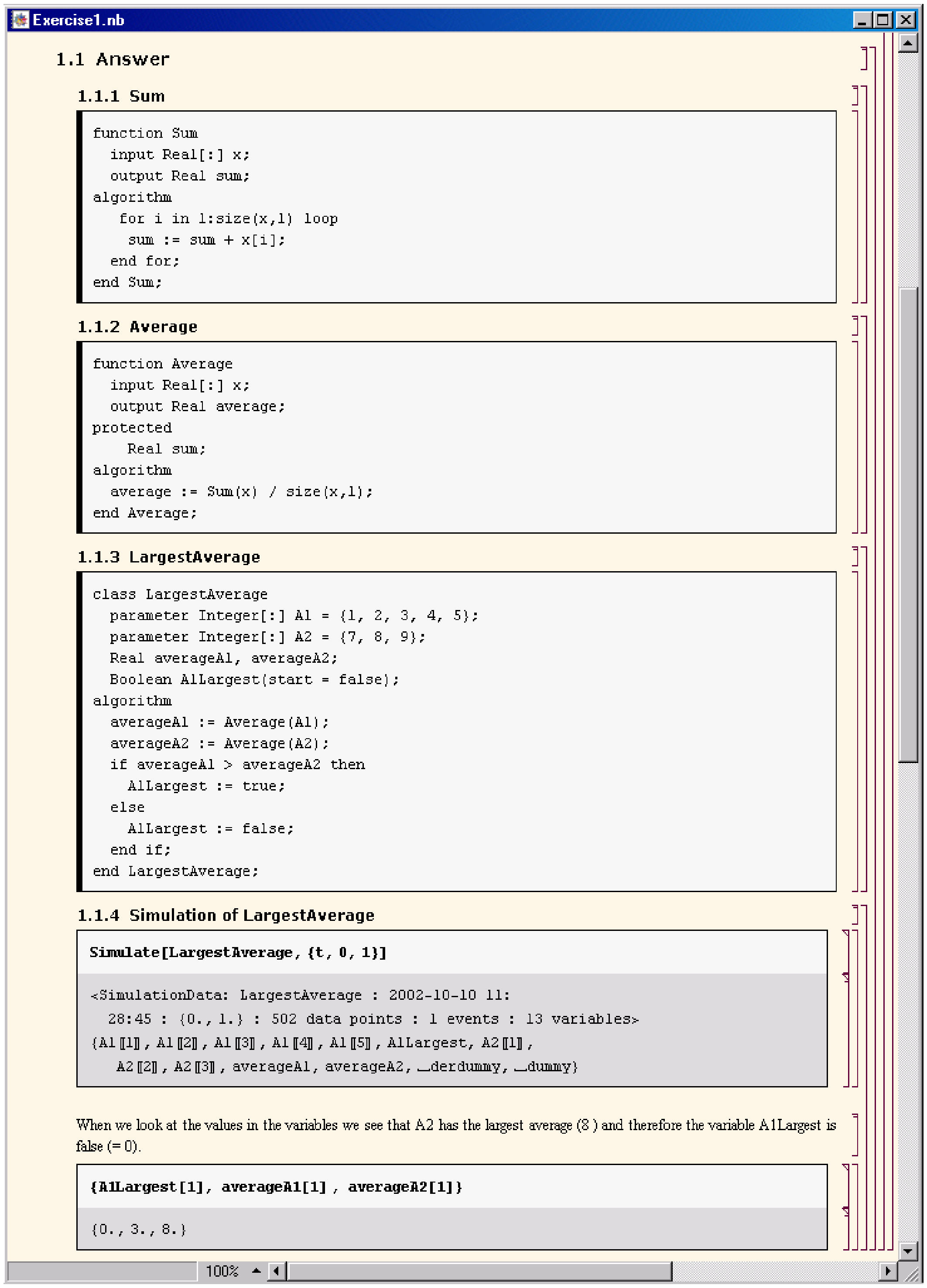
Write a function, Average, which calculates the average of numbers, in an array of arbitrary size. Average should use make a function call to Sum.

Write a class, LargestAverage, that has two arrays and calculates the average of each of them. Then it compares the averages and sets a variable to true if the first array is larger than the second and otherwise false.

1.1 Answer

Figure 3-6. Exercise 1 in Chapter 9 of DrModelica.

Exercise 1 in Section 9.1.1 is shown in Figure 3-6. In this exercise the user has the opportunity to practice different language constructs and then compare the solution to the answer for the exercise. Notice that the answer is not visible until the *Answer* section is expanded. The answer is shown in Figure 3-7.



1.1 Answer

1.1.1 Sum

```
function Sum
  input Real[:] x;
  output Real sum;
algorithm
   for i in 1:size(x,1) loop
    sum := sum + x[i];
  end for;
end Sum;
```

1.1.2 Average

```
function Average
  input Real[:] x;
  output Real average;
protected
    Real sum;
algorithm
  average := Sum(x) / size(x,1);
end Average;
```

1.1.3 LargestAverage

```
class LargestAverage
  parameter Integer[:] A1 = {1, 2, 3, 4, 5};
  parameter Integer[:] A2 = {7, 8, 9};
  Real averageA1, averageA2;
  Boolean A1Largest(start = false);
algorithm
  averageA1 := Average(A1);
  averageA2 := Average(A2);
  if averageA1 > averageA2 then
    A1Largest := true;
  else
    A1Largest := false;
  end if;
end LargestAverage;
```

1.1.4 Simulation of LargestAverage

```
Simulate[LargestAverage, {t, 0, 1}]
```

```
<SimulationData: LargestAverage : 2002-10-10 11:
  28:45 : {0., 1.} : 502 data points : 1 events : 13 variables>
{A1[[1]], A1[[2]], A1[[3]], A1[[4]], A1[[5]], A1Largest, A2[[1]],
   A2[[2]], A2[[3]], averageA1, averageA2, ␣derdummy, ␣dummy}
```

When we look at the values in the variables we see that A2 has the largest average (8) and therefore the variable A1Largest is false (= 0).

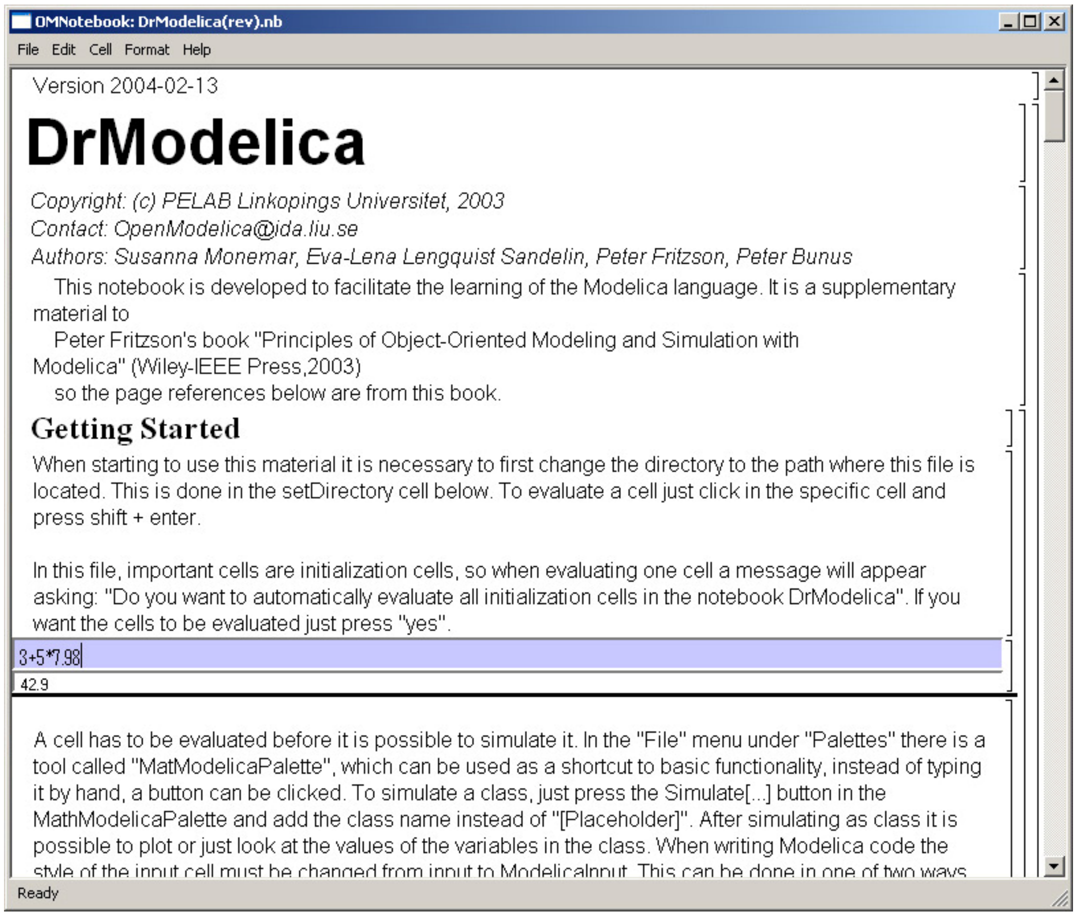
```
{A1Largest[1], averageA1[1] , averageA2[1]}
```

```
{0., 3., 8.}
```

Figure 3-7. The answer section to Exercise 1 in Chapter 9 of DrModelica.

3.3 Using OpenModelica Notebooks on DrModelica

As mentioned in the introduction to this chapter, the OpenModelica notebook facility (OMNotebook) is a simplified implementation of the basic electronic notebook facilities, but advanced enough to represent hierarchical documents, simple type setting, text editing, etc., which is enough to be able to read in the whole DrModelica teaching material.



Version 2004-02-13

DrModelica

Copyright: (c) PELAB Linkopings Universitet, 2003
Contact: OpenModelica@ida.liu.se
Authors: Susanna Monemar, Eva-Lena Lengquist Sandelin, Peter Fritzson, Peter Bunus

This notebook is developed to facilitate the learning of the Modelica language. It is a supplementary material to
Peter Fritzson's book "Principles of Object-Oriented Modeling and Simulation with Modelica" (Wiley-IEEE Press,2003)
so the page references below are from this book.

Getting Started

When starting to use this material it is necessary to first change the directory to the path where this file is located. This is done in the setDirectory cell below. To evaluate a cell just click in the specific cell and press shift + enter.

In this file, important cells are initialization cells, so when evaluating one cell a message will appear asking: "Do you want to automatically evaluate all initialization cells in the notebook DrModelica". If you want the cells to be evaluated just press "yes".

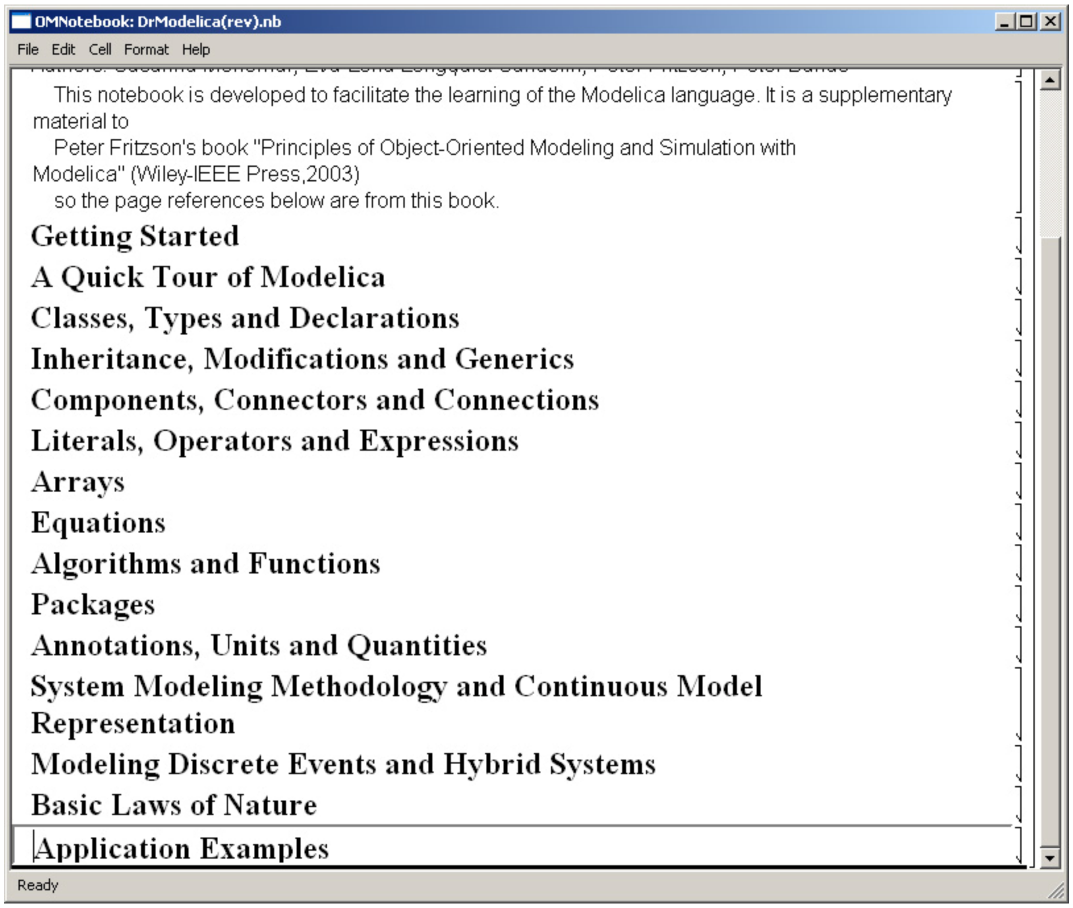
```
3+5*7.98
42.9
```

A cell has to be evaluated before it is possible to simulate it. In the "File" menu under "Palettes" there is a tool called "MatModelicaPalette", which can be used as a shortcut to basic functionality, instead of typing it by hand, a button can be clicked. To simulate a class, just press the Simulate[...] button in the MathModelicaPalette and add the class name instead of "[Placeholder]". After simulating as class it is possible to plot or just look at the values of the variables in the class. When writing Modelica code the style of the input cell must be changed from input to ModelicaInput. This can be done in one of two ways

Figure 3-8. The start page (main page) of DrModelica in the OpenModelica notebook system.

This is exemplified by Figure 3-8, showing the DrModelica main page (start page) in the teaching material.

As can be seen from Figure 3-9 and Figure 3-10, the OpenModelica notebook implementation can already represent the hierarchical structure of documents, cells, etc., but lacks some polishing in terms of formatting and available commands. Also, no graphic information can currently be represented.

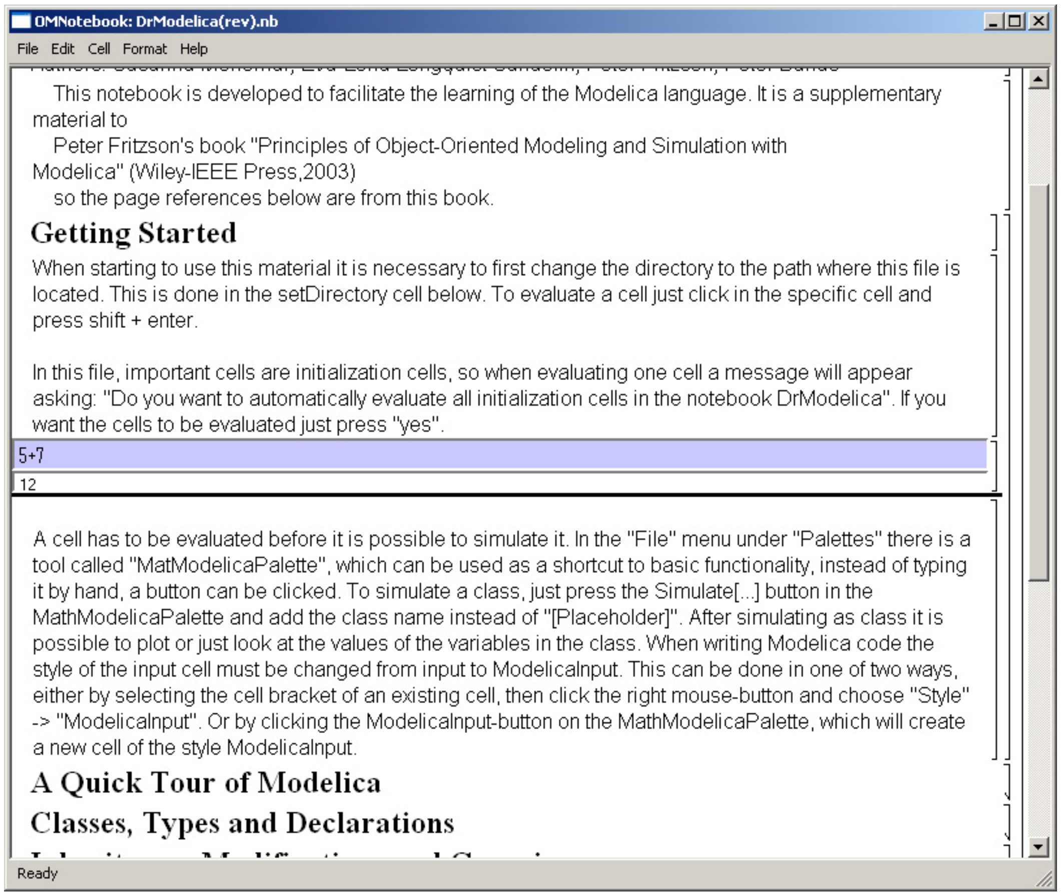


This notebook is developed to facilitate the learning of the Modelica language. It is a supplementary material to
Peter Fritzson's book "Principles of Object-Oriented Modeling and Simulation with Modelica" (Wiley-IEEE Press,2003)
so the page references below are from this book.

Getting Started
A Quick Tour of Modelica
Classes, Types and Declarations
Inheritance, Modifications and Generics
Components, Connectors and Connections
Literals, Operators and Expressions
Arrays
Equations
Algorithms and Functions
Packages
Annotations, Units and Quantities
System Modeling Methodology and Continuous Model Representation
Modeling Discrete Events and Hybrid Systems
Basic Laws of Nature
Application Examples

Figure 3-9. The DrModelica main notebook in OpenModelica with most cells closed, only showing the title of each cell, thus creating a form of table-of-contents for the notebook.

Figure 3-9 shows the main notebook of DrModelica will all cells closed, only showing the heading of each cell/section. This looks like a kind of table-of-contents, which is convenient for navigation in the notebook. To read a closed section, just click on it and it will open.



This notebook is developed to facilitate the learning of the Modelica language. It is a supplementary material to
Peter Fritzson's book "Principles of Object-Oriented Modeling and Simulation with Modelica" (Wiley-IEEE Press,2003)
so the page references below are from this book.

Getting Started

When starting to use this material it is necessary to first change the directory to the path where this file is located. This is done in the setDirectory cell below. To evaluate a cell just click in the specific cell and press shift + enter.

In this file, important cells are initialization cells, so when evaluating one cell a message will appear asking: "Do you want to automatically evaluate all initialization cells in the notebook DrModelica". If you want the cells to be evaluated just press "yes".

```
5+7
12
```

A cell has to be evaluated before it is possible to simulate it. In the "File" menu under "Palettes" there is a tool called "MatModelicaPalette", which can be used as a shortcut to basic functionality, instead of typing it by hand, a button can be clicked. To simulate a class, just press the Simulate[...] button in the MathModelicaPalette and add the class name instead of "[Placeholder]". After simulating as class it is possible to plot or just look at the values of the variables in the class. When writing Modelica code the style of the input cell must be changed from input to ModelicaInput. This can be done in one of two ways, either by selecting the cell bracket of an existing cell, then click the right mouse-button and choose "Style" -> "ModelicaInput". Or by clicking the ModelicaInput-button on the MathModelicaPalette, which will create a new cell of the style ModelicaInput.

A Quick Tour of Modelica

Classes, Types and Declarations

Figure 3-10. The OpenModelica DrModelica notebook, showing the evaluation of a simple expression (5+7) in a cell in the middle, followed by the creation of an output cell to contain the result: 12..

The OpenModelica Notebook facility is coupled to the OpenModelica compiler and simulator, thus allowing evaluation of expressions, simulation of models, and interactive session commands as specified in Section 1.3.

3.4 OpenModelica Notebook Commands

The current prototype (2005-05-11) of OpenModelica notebooks support the following operations:

- Opening and closing groups of cells by double clicking the hierarchical tree view (to the right).
- Evaluation of Modelica code, commands, and expressions in input cells by typing `SHIFT+RETURN`. The evaluation results are shown in a created output cell.
- Opening and loading notebook files stored in XML-format (Command: `CTRL+O`, Menu: File > Open).
- Opening and loading notebook files stored in FullForm Mathematica notebook format (Command: `CTRL+O`, Menu: File > Open).
- Saving notebook files in XML format (Command: `CTRL+S`, Menu: File > Open or File > Save As).
- Close current document (Command: `CTRL+W` or `ALT+F4`, Menu: File > Close).
- Terminating the notebook subsystem (Command: `ALT+Q`, Menu: File > Quit).
- Select a cell, by a single click on the cell in the tree view to the right.

- Select more cells by a single click on the cell in the tree view to the right and holding `CTRL` pressed down.
- Possibility to edit the style template to change the appearance of different cell types. This is done by editing the file stylesheet.xml.
- Move cursor, by `CTRL + UP ARROW` or `CTRL + DOWN ARROW`, or Menu: Cell > Next Cell or Cell > Privous Cell.
- Select and copy text inside a cell.
- Color marking and indentation of Modelica code.
- Show how text is stored inside a cell, by menu Edit -> View Expression.
- Add a new cell (Command: `CTRL+A`, Menu: Cell > Add).
- Create a input cell for openmodelica code (Command: `CTRL+SHIFT+I`, Menu: Format > Input cell).
- Undo on text changes inside a cell (Command: `CTRL+Z`).
- Create a group cell around the cell selected by the cursor (Command: `CTRL+SHIFT+G`, Menu: Format > Group cell).
- Remove selected cell/cells (Command: `CTRL+SHIFT+D`, Menu: Cell > Delete).
- Cut selected cell/cells (Command: `CTRL+SHIFT+X`, Menu: Cell > Cut Cell).
- Copy selected cell/cells (Command: `CTRL+SHIFT+C`, Menu: Cell > Copy Cell).
- Paste cut or copyed cell/cells (Command: `CTRL+SHIFT+V`, Menu: Cell > Paste Cell).
- Change text style for a cell, with menu Format -> Styles -> (select style).
- Clickable active links that opens the linked document when clicked on the link. Links can only be opend if the cell containing the link is not selected.

The following functionality is ongoing work and is currently being implemented:

- Add own links to the text.
- Moving a cell by dragging it in the tree view.
- Pasting graphic images into special graphic cells.
- Display plotted images in special graphic cells.
- Change text settings on individual words and letters.
- Export document text to pure text form with no structure saved.
- Undo/Redo for cell changes.
- Search and replace functions.
- Print function.
- Change between different stylesheets at runtime.
- Change syntax highlighting colors at runtime.
- Magnification on the document.
- Convert the cell type from one type to another type.

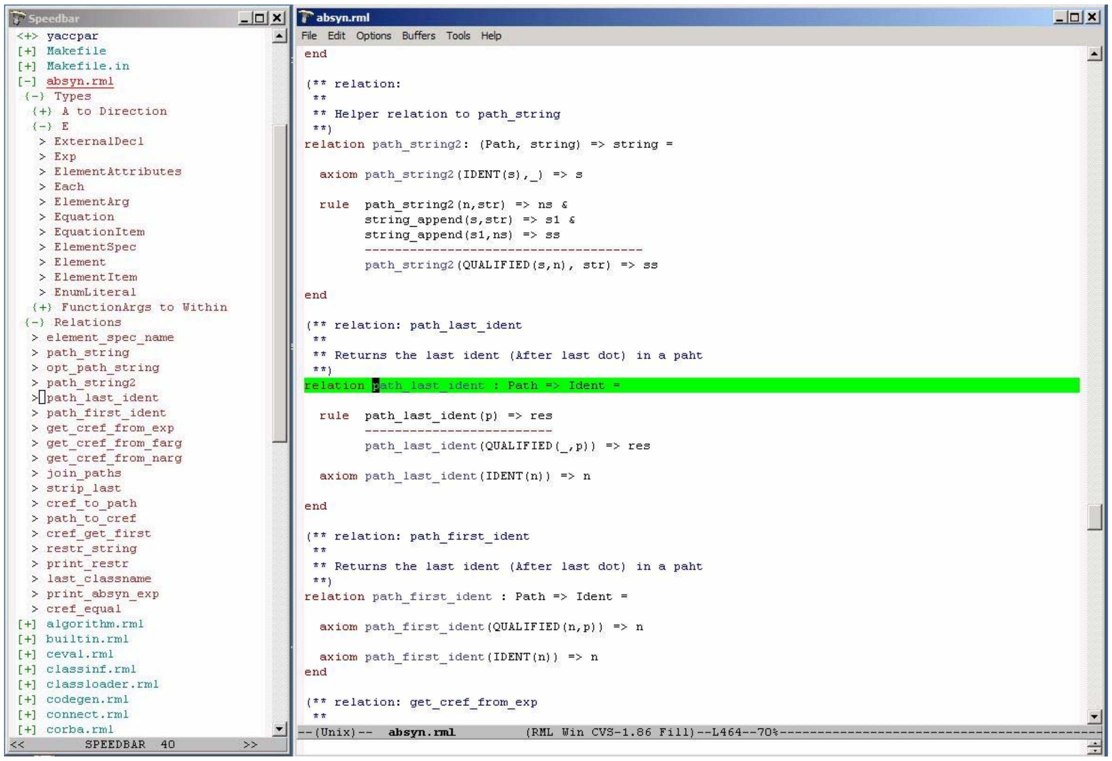
Chapter 4

Emacs Textual Model Editor/Browser

The Emacs Modelica mode provides facilities for keyword highlighting, suppressing annotations, etc.

(?? Need to describe those facilities, including how the Modelica mode is started).

Another quite useful facility is the Speedbar menu, depicted in Figure 4-1. (?? This Screendump shows the same facility used for RML code, not Modelica code. Needs to be updated)



```
<+> yaccpar
[+] Makefile
[+] Makefile.in
[-] absyn.rml
 {-} Types
  {+} A to Direction
  {-} E
   > ExternalDecl
   > Exp
   > ElementAttributes
   > Each
   > ElementArg
   > Equation
   > EquationItem
   > ElementSpec
   > Element
   > ElementItem
   > EnumLiteral
  {+} FunctionArgs to Within
 {-} Relations
  > element_spec_name
  > path_string
  > opt_path_string
  > path_string2
  >path_last_ident
  > path_first_ident
  > get_cref_from_exp
  > get_cref_from_farg
  > get_cref_from_narg
  > join_paths
  > strip_last
  > cref_to_path
  > path_to_cref
  > cref_get_first
  > restr_string
  > print_restr
  > last_classname
  > print_absyn_exp
  > cref_equal
[+] algorithm.rml
[+] builtin.rml
[+] ceval.rml
[+] classinf.rml
[+] classloader.rml
[+] codegen.rml
[+] connect.rml
[+] corba.rml
<<        SPEEDBAR  40        >>
```

```
end

(** relation:
 **
 ** Helper relation to path_string
 **)
relation path_string2: (Path, string) => string =

  axiom path_string2(IDENT(s),_) => s

  rule  path_string2(n,str) => ns &
        string_append(s,str) => s1 &
        string_append(s1,ns) => ss
        -------------------------------------
        path_string2(QUALIFIED(s,n), str) => ss

end

(** relation: path_last_ident
 **
 ** Returns the last ident (After last dot) in a paht
 **)
relation path_last_ident : Path => Ident =

  rule  path_last_ident(p) => res
        -------------------------
        path_last_ident(QUALIFIED(_,p)) => res

  axiom path_last_ident(IDENT(n)) => n

end

(** relation: path_first_ident
 **
 ** Returns the last ident (After last dot) in a paht
 **)
relation path_first_ident : Path => Ident =

  axiom path_first_ident(QUALIFIED(n,p)) => n

  axiom path_first_ident(IDENT(n)) => n
end

(** relation: get_cref_from_exp
 **
--(Unix)--  absyn.rml        (RML Win CVS-1.86 Fill)--L464--70%------------------------------------------
```

Figure 4-1. Emacs with a speedbar menu to the left, which allows clicking on file names (for expansion or closing the file contents menu). An expanded file shows all function, class, and type declarations. By clicking on one of those, you can position the editor at the appropriate definition.

Give the command `M-x speedbar` to start the Speedbar menu. See Section 6.1 for an explanation to the notation M-x, etc.

When you open files the speedbar menu will automatically update itself. You can double-click with the left mouse button or single-click with the middle button to expand trees, and jump between files and program definitions.

At the top you see the search path to the current directory, where you can click on the directory names at different levels to jump back and forth in the hierarchy. Subdirectories are visible in the tree as expandable nodes.

It is also possible to right-click in the speedbar window to have a menu appear.

Chapter 5

MDT – The OpenModelica Development Tooling Eclipse Plugin

5.1 Introduction

The Modelica Development Tooling (MDT) Eclipse Plug-In integrates the OpenModelica compiler with Eclipse. MDT, together with the OpenModelica compiler, provides an environment for working with Modelica development projects.

The following features are available:

- Browsing support for Modelica projects, packages, and classes
- Wizards for creating Modelica projects, packages, and classes
- Syntax color highlighting
- Syntax checking
- Browsing of the Modelica Standard Library

5.2 Installation

The installation of MDT is accomplished by following the below installation instructions. These instructions assume that you have successfully downloaded and installed Eclipse (http://www.eclipse.org).

1. Start Eclipse
2. Select **Help -> Software Updates -> Find and Install...** from the menu
3. Select 'Search for new features to install' and click 'Next'
4. Select 'New Remote Site...'
5. Enter 'MDT' as name and 'http://www.ida.liu.se/labs/pelab/modelica/OpenModelica/MDT' as URL and click 'OK'
6. Make sure 'MDT' is selected and click 'Finish'
7. In the updates dialog select the 'MDT' feature and click 'Next'
8. Read through the license agreement, select 'I accept...' and click 'Next'
9. Click 'Finish' to install MDT

5.3 Getting started

5.3.1 Configuring the OpenModelica Compiler

MDT needs to be able to locate the binary of the compiler. It uses the environment variable OPENMODELICAHOME to do so.

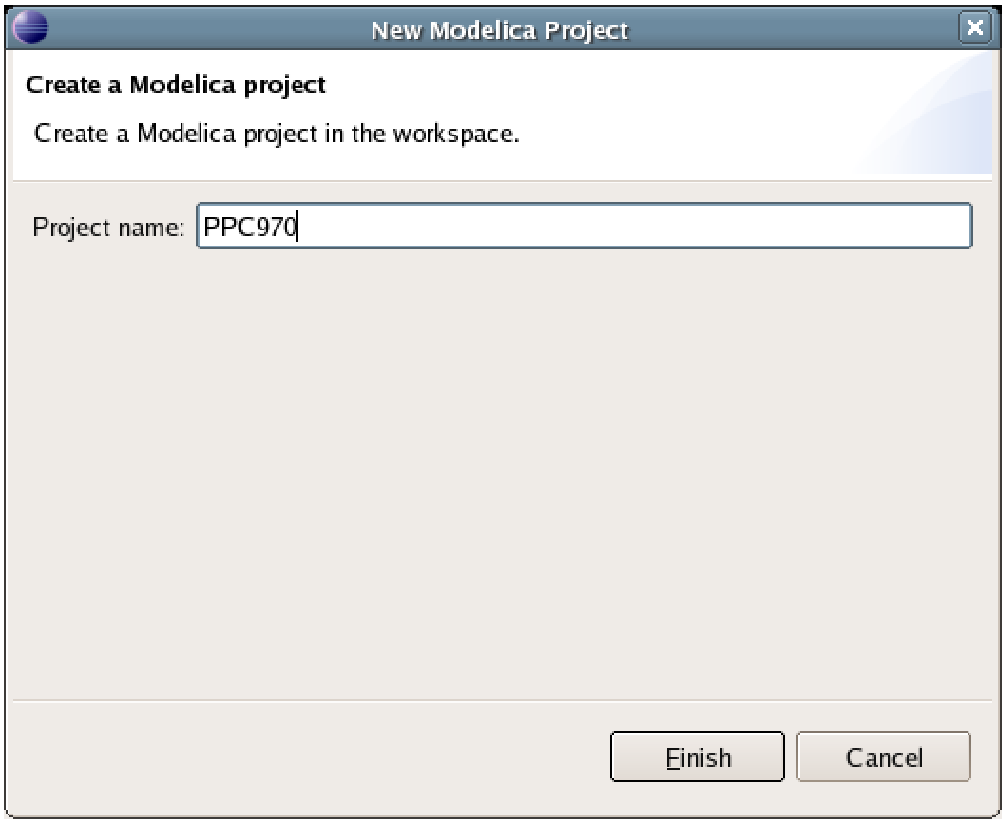
If you have problems using MDT, make sure that OPENMODELICAHOME is pointing to the folder where the Open Modelica Compiler is installed. In other words, OPENMODELICAHOME must point to the folder that contains the Open Modelica Compiler binary. On the Windows platform it's called omc.exe and on Unix platforms it's called omc.

5.3.2 Using the Modelica Perspective

The most convenient way to work with Modelica projects is to use to the Modelica perspective. To switch to the Modelica perspective, choose the **Window** menu item, pick **Open Perspective** followed by **Other...** Select the **Modelica** option from the dialog presented and click **OK**.

5.3.3 Creating a Project

To start a new project, use the **New Modelica Project** Wizard. It is accessible through **File -> New -> Modelica Project** or by right-clicking in the Modelica Projects view and selecting **New -> Modelica Project**.



New Modelica Project

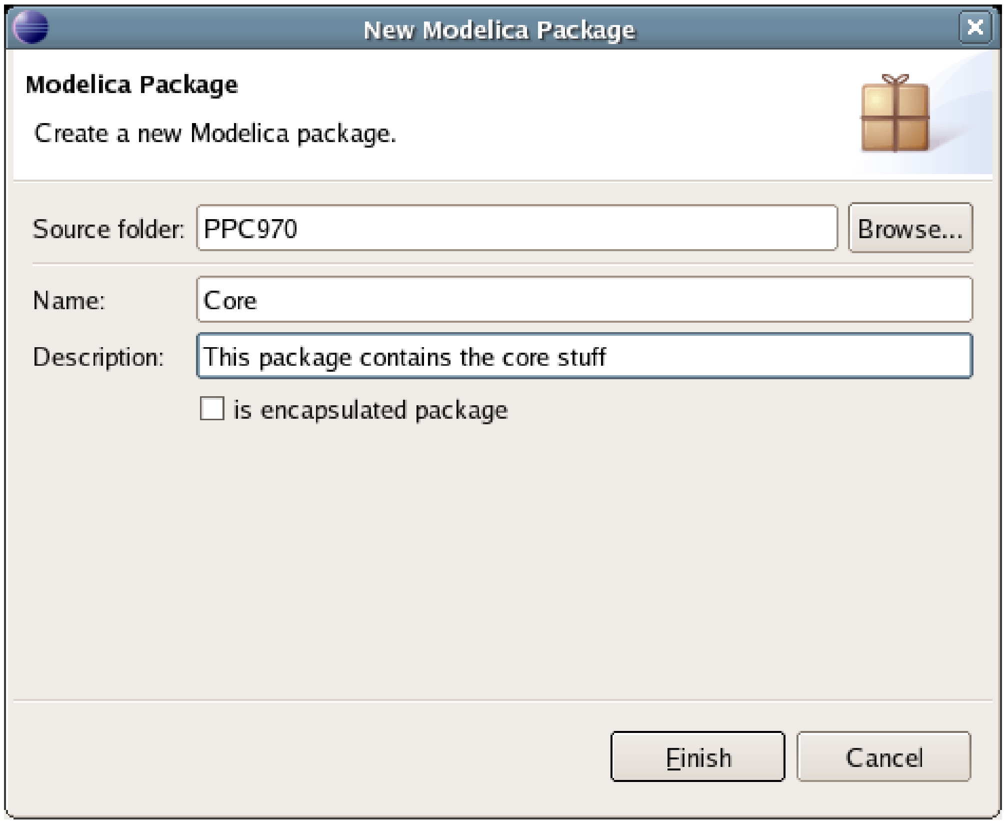
Create a Modelica project

Create a Modelica project in the workspace.

Project name: PPC970

5.3.4 Creating a Package

To create a new package inside a Modelica project, select **File -> New -> Modelica Package.** Enter the desired name of the package and a description of what it contains.



Source folder: PPC970

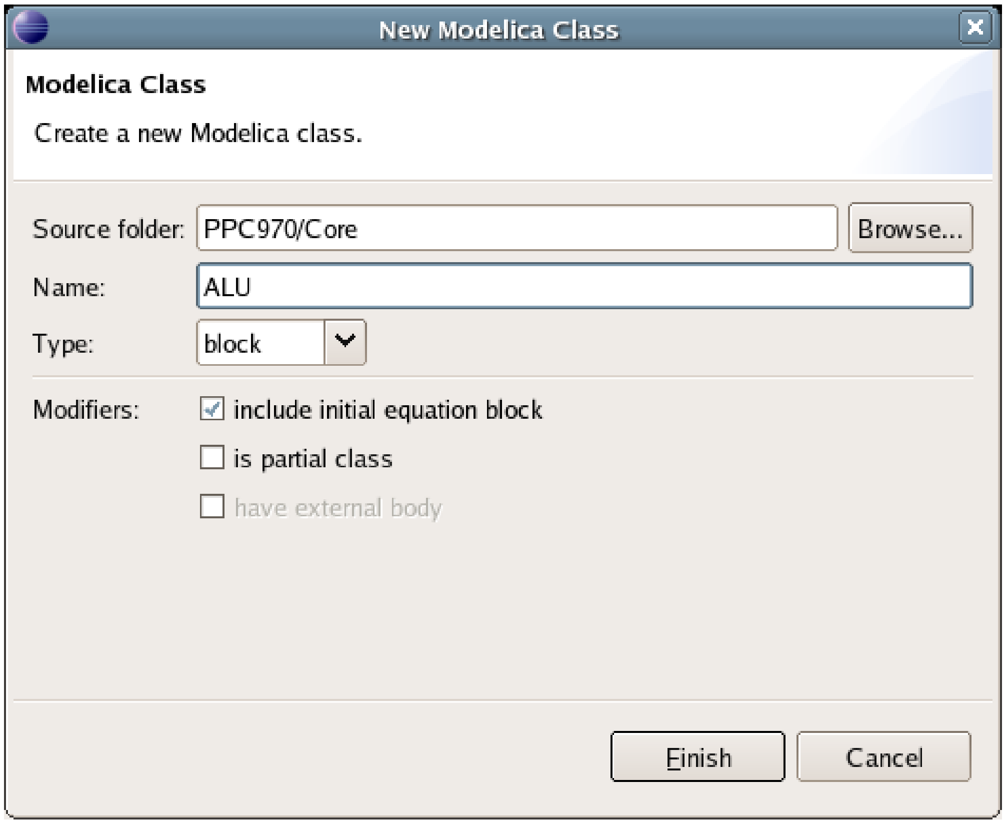
Name: Core

Description: This package contains the core stuff

☐ is encapsulated package

5.3.5 Creating a Class

To create a new Modelica class, select where in the hierarchy that you want to add your new class and select **File -> New -> Modelica Class**. When creating a Modelica class you can add different restrictions on what the class can contain. These can for example be `model`, `connector`, `block`, `record`, or `function`. When you have selected your desired class type, you can select modifiers that add code blocks to the generated code. ‘`Include initial code block`’ will for example add the line ‘`initial equation`’ to the class.



New Modelica Class

Modelica Class

Create a new Modelica class.

Source folder: PPC970/Core

Name: ALU

Type: block

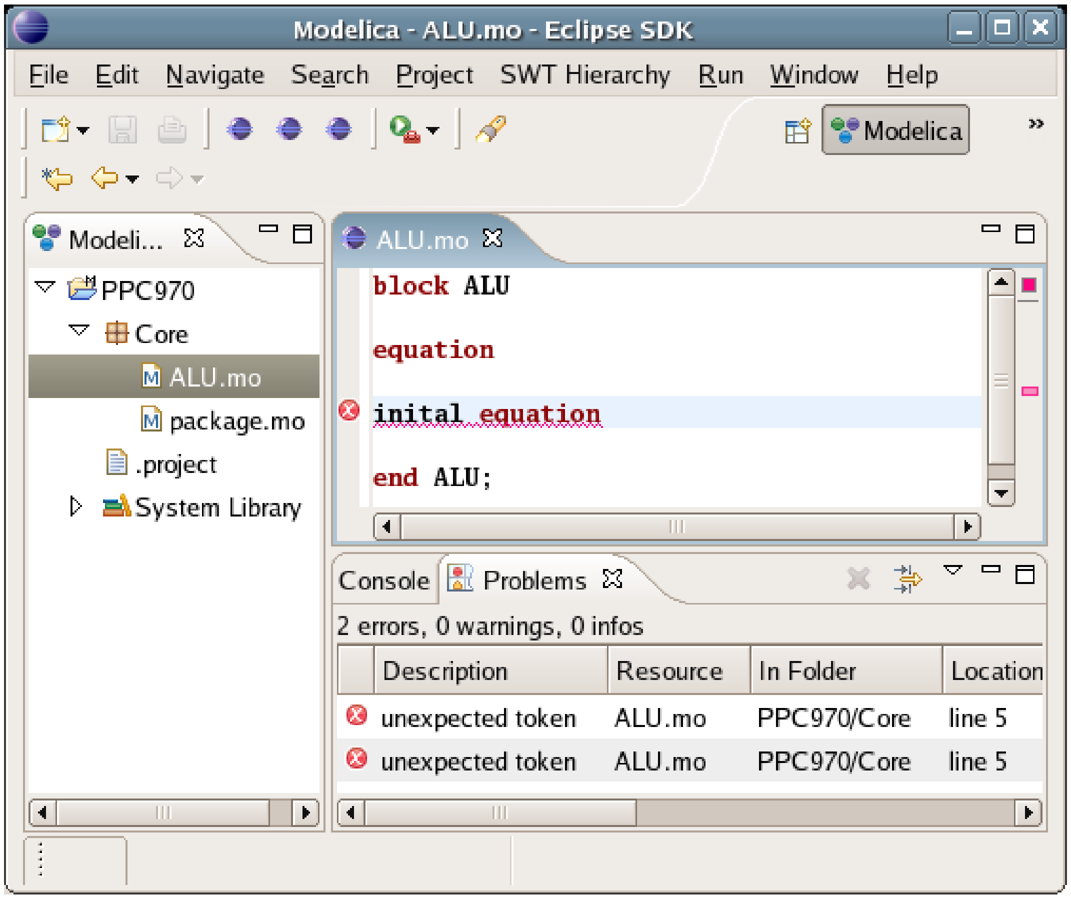
Modifiers: [✓] include initial equation block

[ ] is partial class

[ ] have external body

5.3.6 Syntax Checking

Whenever a Modelica (`.mo`) file is saved by the Modelica Editor, it is checked for syntactical errors. Any errors that are found are added to the Problems view and also marked in the source code editor. Errors are marked in the editor as a red circle with a white cross, a squiggly red line under the problematic construct, and as a red marker in the right-hand side of the editor. If you want to reach the problem, you can either click the item in the Problems view or select the red box in the right-hand side of the editor.



```
block ALU

equation

inital equation

end ALU;
```

2 errors, 0 warnings, 0 infos

Description	Resource	In Folder	Location
unexpected token	ALU.mo	PPC970/Core	line 5
unexpected token	ALU.mo	PPC970/Core	line 5

Chapter 6

Modelica Algorithmic Subset Debugger

This chapter presents a comprehensive Modelica debugger for an extended algorithmic subset of the Modelica language. This replaces debugging of algorithmic code using primitive means such as print statements or asserts which is complex, time-consuming and error- prone.

The debugger is portable since it is based on transparent source code instrumentation techniques that are independent of the implementation platform.

The usual debugging functionality found in debuggers for procedural or traditional object-oriented languages is supported, such as setting and removing breakpoints, single-stepping, inspecting variables, back-trace of stack contents, tracing, etc.

We presents the debugger functionality by a debugging session on a short Modelica example. The functionality of the debugger is shown using pictures from the Emacs debugging mode for Modelica (`modelicadebug-mode`).

Note 1: The current (June 2005) implementation of the debugger only works together with the Modelica compiler version that supports an extended algorithmic subset of Modelica, without equations and simulation, but including meta-programming support. Both compiler versions will be merged into a single version in the near future. It is not yet released for general usage.

Note 2: when applying the debugger to debug the OpenModelica compiler itself, give the `make debug` command to compile the code with debugging turned on, or just the command: `make`, to compile it without debugging support.

6.1 The Debugger Commands

The Emacs Modelica debug mode is implemented as a specialization of the Grand Unified Debugger (GUD) interface (`gud-mode`) from Emacs. Because the Modelica debug mode is based on the GUD interface, some of the commands have the same familiar key bindings.

The actual commands sent to the debugger are also presented together with GUD commands preceded by the Modelica debugger prompt: `mdb@>`.

If the debugger commands have several alternatives these are presented using the notation: `alternative1|alternative2|....`

The optional command components are presented using notation: `[optional]`.

In the Emacs interface: `M-x` stands for holding down the `Meta` key (mapped to `Alt` in general) and pressing the key after the dash, here `x`, `C-x` stands for holding down the `Control (Ctrl)` key and pressing `x`, `<RET>` is equivalent to pressing the `Enter` key, and `<SPC>` to pressing the `Space` key.

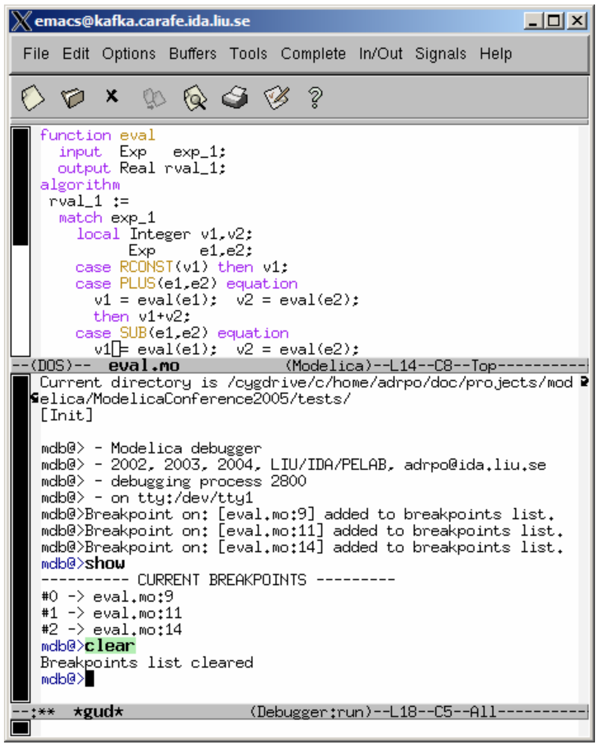
6.2 Starting the Modelica Debugging Subprocess

The command for starting the Modelica debugger under Emacs is the following:

```
M-x modelicadebug <RET> executable <RET>
```

6.3 Setting/Deleting Breakpoints

A part of a session using this type of commands is shown in Figure 6-1 below. The presentation of the commands follows.


emacs@kafka.carafe.ida.liu.se
File Edit Options Buffers Tools Complete In/Out Signals Help

```
function eval
  input  Exp   exp_1;
  output Real rval_1;
algorithm
 rval_1 :=
  match exp_1
    local Integer v1,v2;
          Exp     e1,e2;
    case RCONST(v1) then v1;
    case PLUS(e1,e2) equation
      v1 = eval(e1);  v2 = eval(e2);
      then v1+v2;
    case SUB(e1,e2) equation
      v1 = eval(e1);  v2 = eval(e2);
--(DOS)--  eval.mo            (Modelica)--L14--C8--Top----------
Current directory is /cygdrive/c/home/adrpo/doc/projects/mod
elica/ModelicaConference2005/tests/
[Init]

mdb@> - Modelica debugger
mdb@> - 2002, 2003, 2004, LIU/IDA/PELAB, adrpo@ida.liu.se
mdb@> - debugging process 2800
mdb@> - on tty:/dev/tty1
mdb@>Breakpoint on: [eval.mo:9] added to breakpoints list.
mdb@>Breakpoint on: [eval.mo:11] added to breakpoints list.
mdb@>Breakpoint on: [eval.mo:14] added to breakpoints list.
mdb@>show
---------- CURRENT BREAKPOINTS ---------
#0 -> eval.mo:9
#1 -> eval.mo:11
#2 -> eval.mo:14
mdb@>clear
Breakpoints list cleared
mdb@>
--:**  *gud*           (Debugger:run)--L18--C5--All----------
```

Figure 6-1. Using breakpoints.

To set a breakpoint on the line the cursor (point) is at:

```
C-x <SPC>
mdb@> break on file:lineno|string <RET>
```

To delete a breakpoint placed on the current source code line (`gud-remove`):

```
C-c C-d
C-x C-a C-d
mdb@> break off file:lineno|string <RET>
```

Instead of writing `break` one can use alternatives `br|break|breakpoint`.

Alternatively one can delete all breakpoints using:

```
mdb@> cl|clear <RET>
```

Showing all breakpoints:

```
mdb@> sh|show <RET>
```

6.4 Stepping and Running

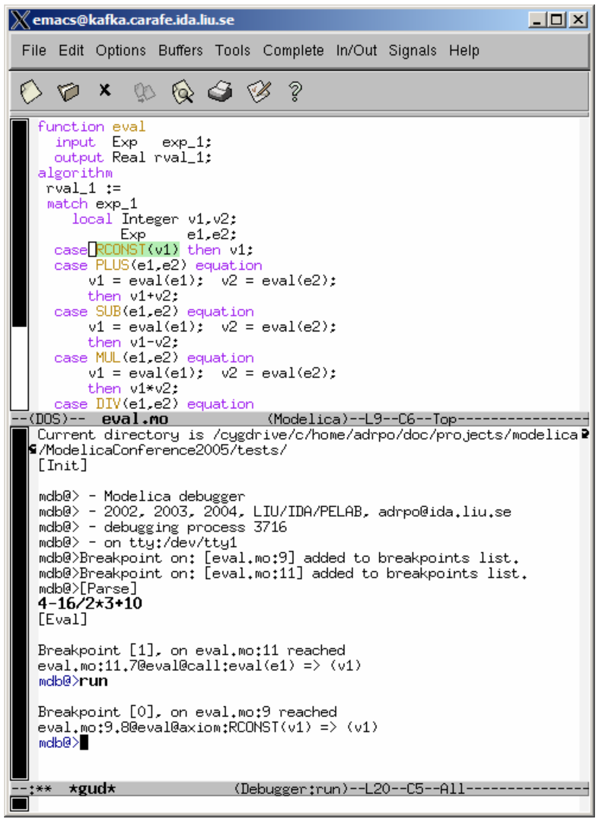
To perform one step (`gud-step`) in the Modelica code:

```
C-c C-s
C-x C-a C-s
mdb@> st|step <RET>
```

To continue after a step or a breakpoint (`gud-cont`) in the Modelica code:

```
C-c C-r
C-x C-a C-r
mdb@> ru|run <RET>
```

Examples of using these commands are presented in Figure 6-2.


emacs@kafka.carafe.ida.liu.se
File Edit Options Buffers Tools Complete In/Out Signals Help

```
function eval
  input  Exp   exp_1;
  output Real rval_1;
algorithm
 rval_1 :=
 match exp_1
    local Integer v1,v2;
          Exp     e1,e2;
  case RCONST(v1) then v1;
  case PLUS(e1,e2) equation
      v1 = eval(e1);  v2 = eval(e2);
      then v1+v2;
  case SUB(e1,e2) equation
      v1 = eval(e1);  v2 = eval(e2);
      then v1-v2;
  case MUL(e1,e2) equation
      v1 = eval(e1);  v2 = eval(e2);
      then v1*v2;
  case DIV(e1,e2) equation
--(DOS)--  eval.mo          (Modelica)--L9--C6--Top---------------
Current directory is /cygdrive/c/home/adrpo/doc/projects/modelica
/ModelicaConference2005/tests/
[Init]

mdb@> - Modelica debugger
mdb@> - 2002, 2003, 2004, LIU/IDA/PELAB, adrpo@ida.liu.se
mdb@> - debugging process 3716
mdb@> - on tty:/dev/tty1
mdb@>Breakpoint on: [eval.mo:9] added to breakpoints list.
mdb@>Breakpoint on: [eval.mo:11] added to breakpoints list.
mdb@>[Parse]
4-16/2*3+10
[Eval]

Breakpoint [1], on eval.mo:11 reached
eval.mo:11.7@eval@call:eval(e1) => (v1)
mdb@>run

Breakpoint [0], on eval.mo:9 reached
eval.mo:9.8@eval@axiom:RCONST(v1) => (v1)
mdb@>
--:**  *gud*          (Debugger:run)--L20--C5--All---------------
```

Figure 6-2. Stepping and running.

6.5 Examining Data

There are no GUD keybindings for these commands but they are inspired from the GNU Project debugger (GDB).

To print the contents/size of a variable one can write:

```
mdb@> pr|print variable_name <RET>
mdb@> sz|sizeof variable_name <RET>
```

at the debugger prompt. The size is displayed in bytes.

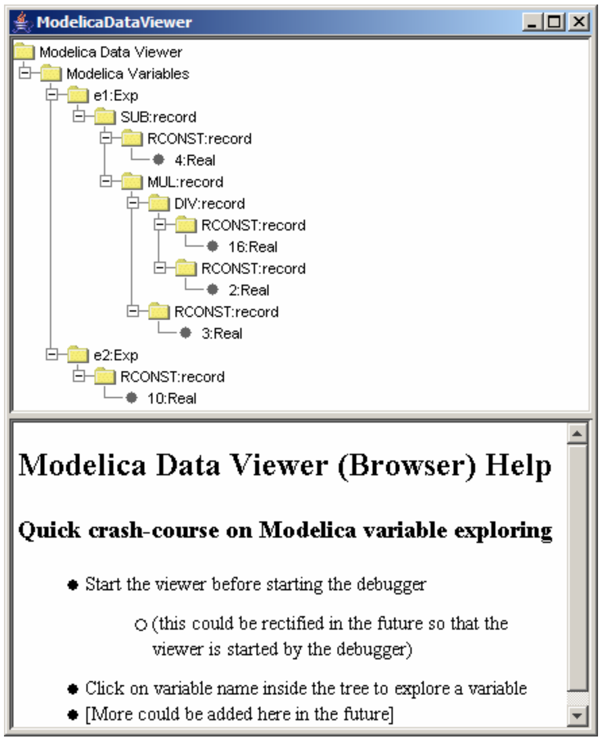
Variable values to be printed can be of a complex type and very large. One can restrict the depth of printing using:

```
mdb@> [set] de|depth integer <RET>
```

Moreover, we have implemented an external viewer written in Java called `ModelicaDataViewer` to browse the contents of such a large variable. To send the contents of a variable to the external viewer for inspection one can use the command:

```
mdb@> bw|browse|gr|graph var_name <RET>
```

at the debugger prompt. The debugger will try to connect to the `ModelicaDataViewer` and send the contents of the variable. The external data browser has to be started a priori. If the debugger cannot connect to the external viewer within a specified timeout a warning message will be displayed. A picture of the external `ModelicaDataViewer` tool is presented in Figure 6-3.



```
Modelica Data Viewer
  Modelica Variables
    e1:Exp
      SUB:record
        RCONST:record
          4:Real
        MUL:record
          DIV:record
            RCONST:record
              16:Real
            RCONST:record
              2:Real
          RCONST:record
            3:Real
    e2:Exp
      RCONST:record
        10:Real
```

Modelica Data Viewer (Browser) Help

Quick crash-course on Modelica variable exploring

- Start the viewer before starting the debugger
  - (this could be rectified in the future so that the viewer is started by the debugger)
- Click on variable name inside the tree to explore a variable
- [More could be added here in the future]

Figure 6-3. Modelica Data Viewer (Browser) for data structures, here a small abstract syntax tree.

If the variable which one tries to print does not exist in the current scope (not a live variable) a notifying warning message will be displayed.

Automatic printing of variables at every step or breakpoint can be specified by adding a variable to a display list:

```
mdb@> di|display variable_name <RET>
```

To print the entire display list:

```
mdb@> di|display <RET>
```

Removing a display variable from the display list:

```
mdb@> un|undisplay variable_name <RET>
```

Removing all variables from the display list:

```
mdb@> undisplay <RET>
```

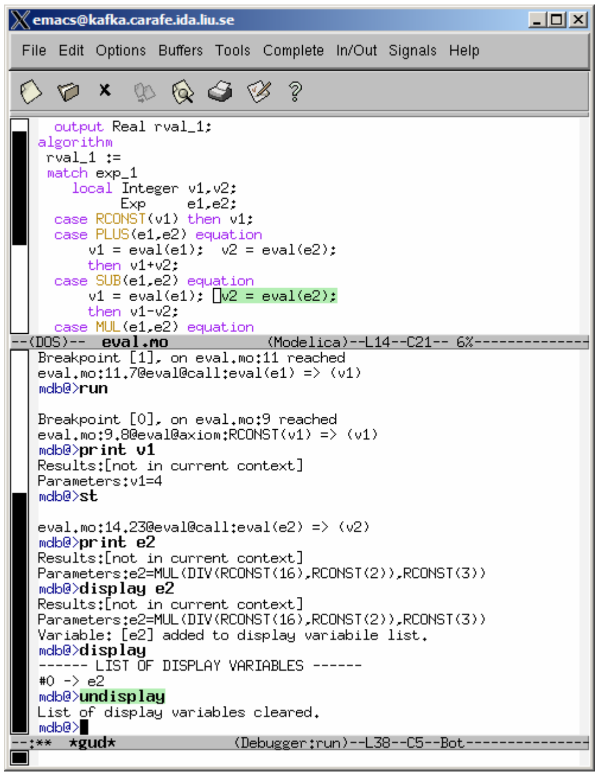
Printing the current live variables:

```
mdb@> li|live|livevars <RET>
```

Instructing the debugger to print or to disable the print of the live variable names at each step/breapoint:

```
mdb@> [set] li|live|livevars [on|off]<RET>
```

Figure 6-4 shows examples of some of these commands within a debugging session:


emacs@kafka.carafe.ida.liu.se
File Edit Options Buffers Tools Complete In/Out Signals Help

```
  output Real rval_1;
algorithm
 rval_1 :=
 match exp_1
    local Integer v1,v2;
          Exp     e1,e2;
  case RCONST(v1) then v1;
  case PLUS(e1,e2) equation
      v1 = eval(e1);  v2 = eval(e2);
      then v1+v2;
  case SUB(e1,e2) equation
      v1 = eval(e1); v2 = eval(e2);
      then v1-v2;
  case MUL(e1,e2) equation
--(DOS)--  eval.mo           (Modelica)--L14--C21-- 6%------------
Breakpoint [1], on eval.mo:11 reached
eval.mo:11.7@eval@call:eval(e1) => (v1)
mdb@>run

Breakpoint [0], on eval.mo:9 reached
eval.mo:9.8@eval@axiom:RCONST(v1) => (v1)
mdb@>print v1
Results:[not in current context]
Parameters:v1=4
mdb@>st

eval.mo:14.23@eval@call:eval(e2) => (v2)
mdb@>print e2
Results:[not in current context]
Parameters:e2=MUL(DIV(RCONST(16),RCONST(2)),RCONST(3))
mdb@>display e2
Results:[not in current context]
Parameters:e2=MUL(DIV(RCONST(16),RCONST(2)),RCONST(3))
Variable: [e2] added to display variabile list.
mdb@>display
------ LIST OF DISPLAY VARIABLES ------
#0 -> e2
mdb@>undisplay
List of display variables cleared.
mdb@>
--:**  *gud*           (Debugger:run)--L38--C5--Bot-------------
```

Figure 6-4. Examining variable values using print and display commands.

6.6 Additional commands

The stack contents (backtrace) can be displayed using:

```
mdb@> bt|backtrace <RET>
```

Because the contents of the stack can be quite large, one can print a filtered view of it:

```
mdb@> fbt|fbacktrace filter_string <RET>
```

Also, one can restrict the numbers of entries the debugger is storing using:

```
mdb@> maxbt|maxbacktrace integer <RET>
```

For displaying the status of the Modelica runtime:

```
mdb@> sts|stat|status <RET>
```

The status of the extended Modelica runtime comprises information regarding the garbage collector, allocated memory, stack usage, etc.

The current debugging settings can be displayed using:

```
mdb@> stg|settings <RET>
```

The settings printed are: the maximum remembered backtrace entries, the depth of variable printing, the current breakpoints, the live variables, the list of the display variables and the status of the runtime system.

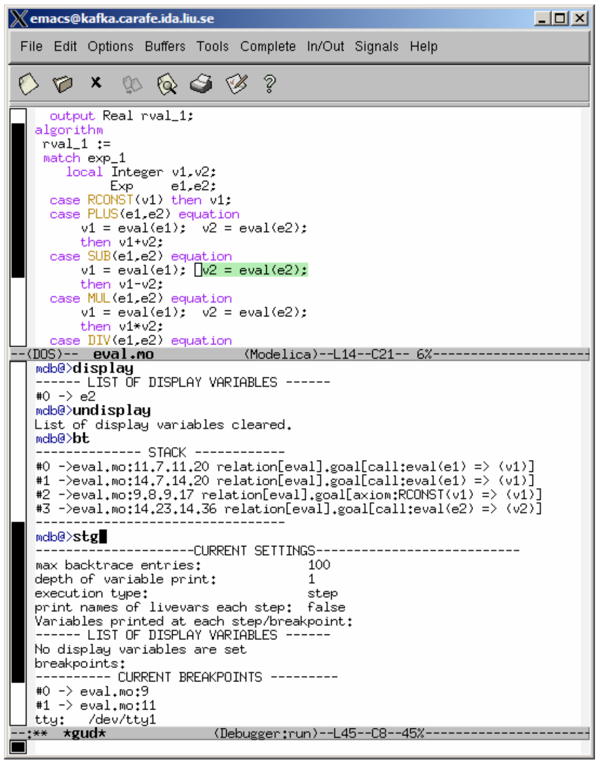
One can invoke the debugging help by issuing:

```
mdb@> he|help <RET>
```

For leaving the debugger one can use the command:

```
mdb@> qu|quit|ex|exit|by|bye <RET>
```

A session using these commands is presented in Figure 6-5 below:


emacs@kafka.carafe.ida.liu.se
File Edit Options Buffers Tools Complete In/Out Signals Help

```
  output Real rval_1;
algorithm
 rval_1 :=
 match exp_1
    local Integer v1,v2;
          Exp     e1,e2;
  case RCONST(v1) then v1;
  case PLUS(e1,e2) equation
      v1 = eval(e1);  v2 = eval(e2);
      then v1+v2;
  case SUB(e1,e2) equation
      v1 = eval(e1); v2 = eval(e2);
      then v1-v2;
  case MUL(e1,e2) equation
      v1 = eval(e1);  v2 = eval(e2);
      then v1*v2;
  case DIV(e1,e2) equation
--(DOS)--  eval.mo          (Modelica)--L14--C21-- 6%--------------------
mdb@>display
------ LIST OF DISPLAY VARIABLES ------
#0 -> e2
mdb@>undisplay
List of display variables cleared.
mdb@>bt
-------------- STACK ------------
#0 ->eval.mo:11.7.11.20 relation[eval].goal[call:eval(e1) => (v1)]
#1 ->eval.mo:14.7.14.20 relation[eval].goal[call:eval(e1) => (v1)]
#2 ->eval.mo:9.8.9.17 relation[eval].goal[axiom:RCONST(v1) => (v1)]
#3 ->eval.mo:14.23.14.36 relation[eval].goal[call:eval(e2) => (v2)]
---------------------------------
mdb@>stg
---------------------CURRENT SETTINGS---------------------------
max backtrace entries:             100
depth of variable print:           1
execution type:                    step
print names of livevars each step:  false
Variables printed at each step/breakpoint:
------ LIST OF DISPLAY VARIABLES ------
No display variables are set
breakpoints:
---------- CURRENT BREAKPOINTS ---------
#0 -> eval.mo:9
#1 -> eval.mo:11
tty:   /dev/tty1
--:**  *gud*         (Debugger:run)--L45--C8--45%--------------------
```

Figure 6-5. Additional debugger commands.

6.7 Hints for Debugging Large Programs

In order to faster get to an interesting place when debugging a large program such as the OpenModelica compiler itself, you can put a breakpoint at the place where you would like to start the investigation, but give the `fast` debug command when starting the execution from the beginning. In that case the debugger will avoid saving backtrace and variables up to this breakpoint. Then you can turn off backtrace and run the debugger as usual.

6.8 Summary of Debugger Commands

The following is a complete list of the current debugger commands

`br|break|breakpoint` *string* [on|off]	Setting/unsetting breakpoints
`cl|clear`	Clear all breakpoints
`sh|show`	Show all breakpoints
`bt|backtrace`	Print the backtrace (stack)
`fbt|fbacktrace` *filter*	Print filtered backtrace (stack)
`mb|maxbacktrace` *int* (0=full, default=0)	Set the maximum of backtrace entries (stack).
`ca|callchain`	Print the call chain
`fca|fcallchain` *filter*	Print filtered call chain

`mc|maxcallchain` *integer*	Set the maximum of callchain entries. (0=full, default=100)
`[set] de|depth` *integer*	Set the depth of variable printing. (0=full, default=10)
`[set] ms|maxstring` *integer*	Set how may chars we print from long strings. (0=full, default=60)
`set st|step [on|off]`	Set the execution mode.
`st|step|<ENTER>|<CR>`	Perform one step.
`ne|next`	Jump over next statement.
`ru|run`	Run the program.
`stg|settings`	Print the current settings.
`he|help`	Showing help.
`sts|stat|status`	Printing the status of Modelica runtime.
`li|live|livevars`	Print the names of live variables.
`[set] li|live|livevars [on|off]`	On/Off printing names of livevars each step.
`pr|print` *var_name*	Print the live variable.
`sz|size|sizeof` *var_name*	Print sizeof the live variable.
`di|display` *var_name*	Display the live variable each step.
`ud|undisplay` *var_name*	Un-display the live variable.
`di|display`	Show display variables.
`ud|undisplay`	Un-display ALL display variables.
`gr|graph` *var_name*	Send the live variable to external viewer.
`pty|printtype` *identifier*	Print type info on any Modelica id.
`fa|fast`	FAST debugging: no backtrace, callchain, livevars.
`qu|quit|ex|exit|by|bye`	Exiting the debugger/program.

Index