

OpenModelica Users Guide

Preliminary Draft, 2006-05-02
for OpenModelica 1.4.0

Version 0.7, May 2006

Peter Fritzson

Peter Aronsson, Adrian Pop, Håkan Lundvall,
Bernhard Bachmann, David Broman, Anders Fernström,
Daniel Hedberg, Elmir Jagudin, Kaj Nyström,
Andreas Remar, Levon Saldamli, Anders Sandholm

Copyright by:

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

Copyright © 1998-2006, PELAB, Department of Computer and Information Science, Linköpings universitet.

All rights reserved.

This document is part of OpenModelica: www.ida.liu.se/projects/OpenModelica

Contact: OpenModelica@ida.liu.se

(Here using the new BSD license, see also <http://www.opensource.org/licenses/bsd-license.php>)

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of Linköpings universitet nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.

IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Modelica[®] is a registered trademark of Modelica Association.

MathModelica[®] is a registered trademark of MathCore Engineering AB.

Mathematica[®] is a registered trademark of Wolfram Research Inc.

Table of Contents

Table of Contents	5
Preface	7
Chapter 1 Introduction	9
1.1 System Overview	10
1.1.1 Implementation Status	11
1.2 Interactive Session with Examples.....	12
1.2.1 Starting the Interactive Session	12
1.2.2 Trying the Bubblesort Function.....	12
1.2.3 Trying the system and cd Commands.....	13
1.2.4 Modelica Library and DCMotor Model	14
1.2.5 BouncingBall and Switch Models	16
1.2.6 Clear All Models	18
1.2.7 VanDerPol Model and Parametric Plot	18
1.2.8 Variables, Functions, and Types of Variables	19
1.2.9 Using External Functions	20
1.2.10 Quit OpenModelica	21
1.3 Commands for the Interactive Session Handler	21
Chapter 2 Using the Graphical Model Editor	23
2.1 Building a Simple DCMotor Model.....	24
2.1.1 Creating a New Model.....	24
2.1.2 The Graphic Editor Text View	27
2.1.3 Plotting	28
Chapter 3 OMNotebook with DrModelica	29
3.1 Interactive Notebooks with Literate Programming	29
3.1.1 Mathematica Notebooks	29
3.1.2 OMNotebook	29
3.2 The DrModelica Tutoring System – an Application of OMNotebook.....	30
3.3 OpenModelica Notebook Commands	35
3.3.1 Cells.....	36
3.3.2 Cursors.....	36
3.4 Selection of Text or Cells.....	36
3.4.1 File Menu	37
3.4.2 Edit Menu	37
3.4.3 Cell Menu	38
3.4.4 Format Menu	38
3.4.5 Insert Menu.....	39
3.4.6 Window Menu	39
3.4.7 Help Menu	39
3.4.8 Additional Features	39
3.5 References.....	40

Chapter 4	Emacs Textual Model Editor/Browser	42
Chapter 5	MDT – The OpenModelica Development Tooling Eclipse Plugin.....	44
5.1	Introduction.....	44
5.2	Installation.....	44
5.3	Getting started.....	45
5.3.1	Configuring the OpenModelica Compiler	45
5.3.2	Using the Modelica Perspective	45
5.3.3	Creating a Project	45
5.3.4	Creating a Package	46
5.3.5	Creating a Class.....	47
5.3.6	Syntax Checking.....	48
5.3.7	Indentation Support	48
5.3.8	Code Completion.....	49
Chapter 6	Modelica Algorithmic Subset Debugger	51
6.1	The Debugger Commands	51
6.2	Starting the Modelica Debugging Subprocess	51
6.3	Setting/Deleting Breakpoints	52
6.4	Stepping and Running	52
6.5	Examining Data.....	53
6.6	Additional commands	55
6.7	Hints for Debugging Large Programs	56
6.8	Summary of Debugger Commands	56
Appendix A	Contributors to OpenModelica	59
A.1	OpenModelica Contributors 2006.....	59
A.2	OpenModelica Contributors 2005.....	59
A.3	OpenModelica Contributors 2004.....	59
A.4	OpenModelica Contributors 2003.....	60
A.5	OpenModelica Contributors 2002.....	60
A.6	OpenModelica Contributors 2001	60
A.7	OpenModelica Contributors 2000.....	60
A.8	OpenModelica Contributors 1999.....	60
A.9	OpenModelica Contributors 1998.....	60
Index	61

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 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.

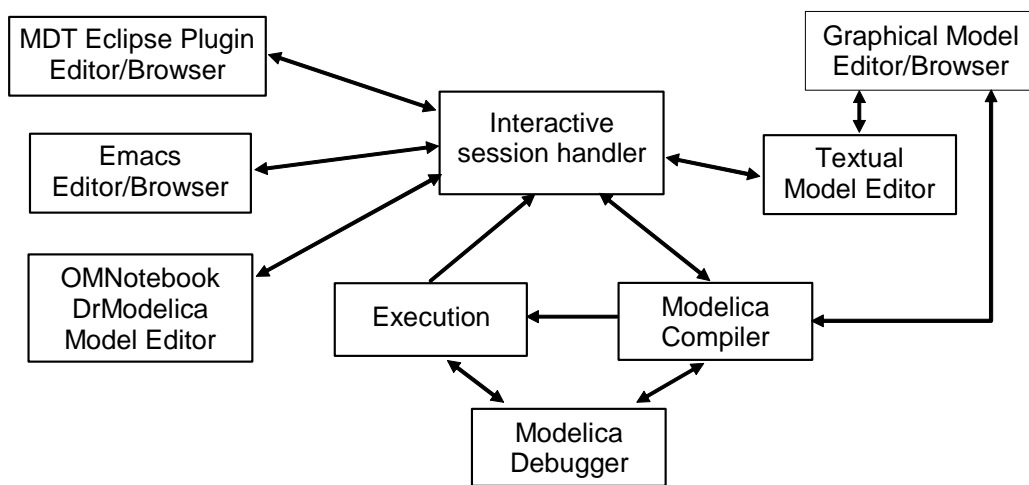


Figure 1-1. The architecture of the OpenModelica environment. Arrows denote data and control flow. The interactive session handler receives commands and shows results from evaluating commands and expressions that are translated and executed. Several subsystems provide different forms of browsing and 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 called MDT (Modelica Development Tooling) 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

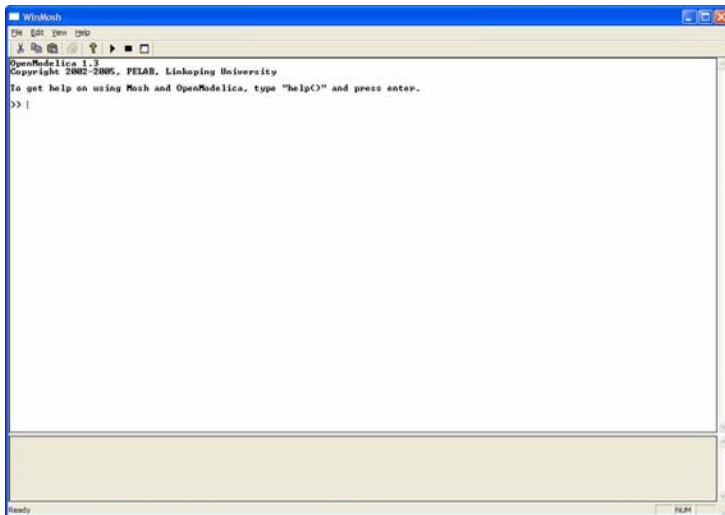
In the current OpenModelica implementation version 1.3.2 (April 2006), not all subsystems are yet integrated as well as is indicated in Figure 1-1. Currently there are two versions of the Modelica compiler, one which supports most of standard Modelica including simulation, and is connected to the interactive session handler, the notebook editor, and the graphic model editor, and another meta-programming Modelica compiler version (called MetaModelica compiler) which is integrated with the debugger, Eclipse, 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, called OMShell – the OpenModelica Shell). Most of these examples are also available in the OpenModelica notebook `UsersGuideExamples.onb` in the `testmodels` directory, see also Chapter 3.

1.2.1 Starting the Interactive Session

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



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}
```

1.2.2 Trying the Bubblesort Function

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

```
>> loadFile("C:/OpenModelica1.4.0/testmodels/bubblesort.mo")
true
```

The function `bubblesort` is called below to sort the vector `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:/OpenModelica1.4.0/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;
```

1.2.3 Trying the system and cd Commands

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:\OpenModelica1.4.0\testmodels"

>> cd("../")
"C:\OpenModelica1.4.0"

>> cd("C:\\OpenModelica1.4.0\\testmodels")
"C:\OpenModelica1.4.0\testmodels"
```

1.2.4 Modelica Library and DCMotor Model

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:/OpenModelica1.4.0/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 il;
  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,il.p);
  connect(il.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 il.v "Voltage drop between the two pins (= p.v - n.v)";
Real il.i "Current flowing from pin p to pin n";
Real il.p.v "Potential at the pin";
Real il.p.i "Current flowing into the pin";
Real il.n.v "Potential at the pin";
Real il.n.i "Current flowing into the pin";
parameter Real il.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;
"

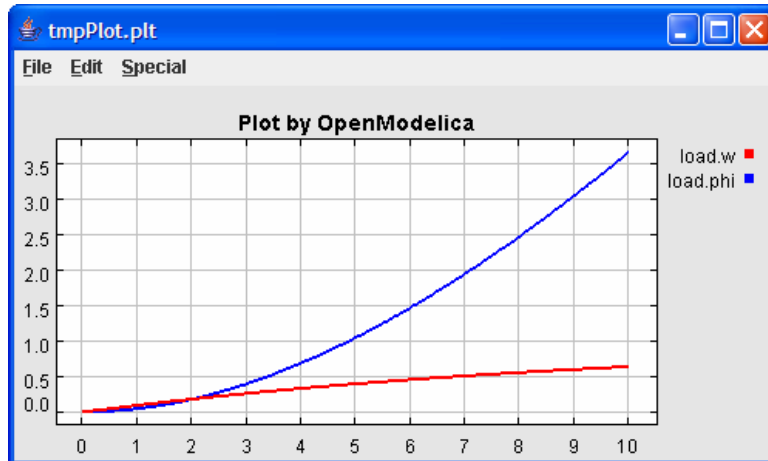
```

We plot part of the simulated result:

```

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

```



1.2.5 BouncingBall and Switch Models

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:/OpenModelica1.4.0/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;
"

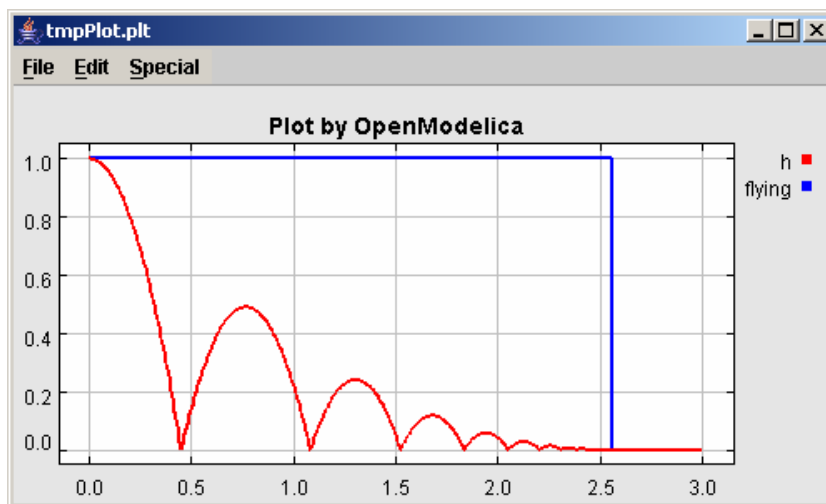
```


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"
```



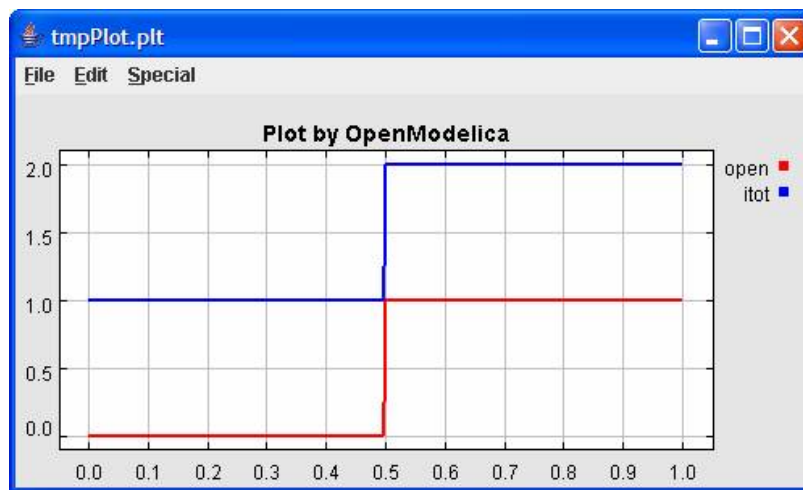
We enter a switch model, to test if-equations (e.g. copy and paste from another file and push enter):

```
>> model Switch
  Real v;
  Real i;
  Real il;
  Real itot;
  Boolean open;
equation
  itot = i + il;

  if open then
    v = 0;
  else
    i = 0;
  end if;
  1 - il = 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.

1.2.6 Clear All Models

Now, first clear all loaded libraries and models:

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

List the loaded models – nothing left:

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

1.2.7 VanDerPol Model and Parametric Plot

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

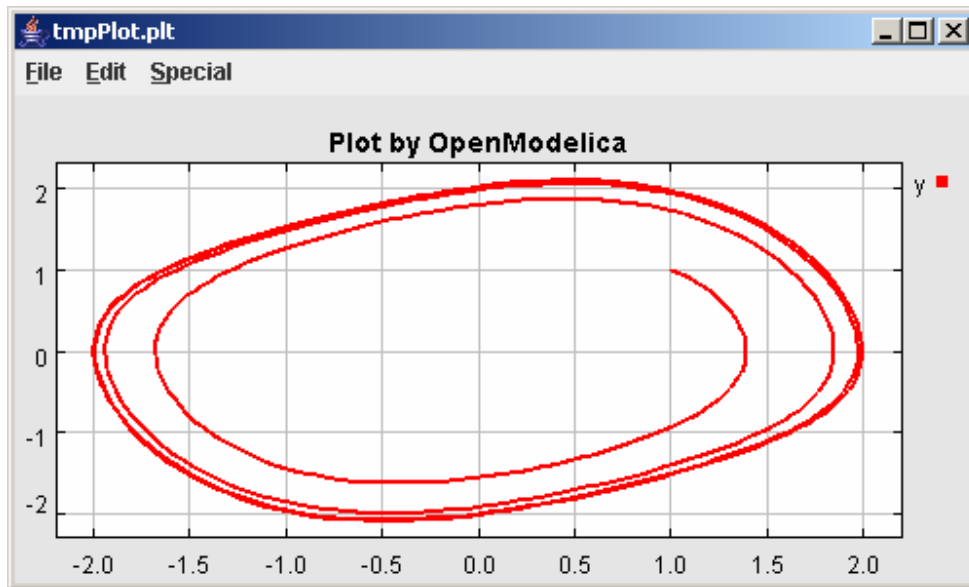
```
>> loadFile("C:/OpenModelica1.4.0/testmodels/VanDerPol.mo")
true
```

It is simulated:

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

It is plotted:

```
plotParametric(x,y);
```



Perform code instantiation to flat form 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;
"
```

1.2.8 Variables, Functions, and Types of Variables

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}
```

Clear again:

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

1.2.9 Using External Functions

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:

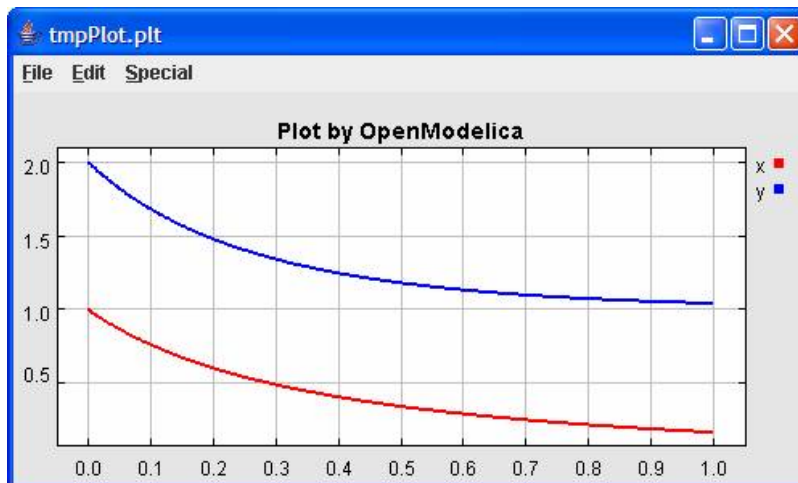
```
loadFile("ExternalLibraries.mo");
system("gcc -c -o libExternalFunc1_ext.o ExternalFunc1.c");
system("gcc -c -o libExternalFunc2.a ExternalFunc2.c");
simulate(ExternalLibraries);
```

We run the script:

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

and plot the results:

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



1.2.10 Quit OpenModelica

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 handler.

`simulate(modelname)` Translate a model named *modelname* and simulate it.

<code>simulate(modelname [, startTime=<Real>] [, stopTime=<Real>] [, numberOfIntervals=<Integer>])</code>	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.
<code>plot(vars)</code>	Plot the variables given as a vector or a scalar, e.g. <code>plot({x1,x2})</code> or <code>plot(x1)</code> .
<code>plotParametric(var1,var2)</code>	Plot var2 relative to var1 from the most recently simulated model, e.g. <code>plotParametric(x,y)</code> .
<code>cd()</code>	Return the current directory.
<code>cd(dir)</code>	Change directory to the directory given as string.
<code>clear()</code>	Clear all loaded definitions.
<code>clearVariables()</code>	Clear all defined variables.
<code>instantiateModel(modelname)</code>	Performs code instantiation of a model/class and return a string containing the flat class definition.
<code>list()</code>	Return a string containing all loaded class definitions.
<code>list(modelname)</code>	Return a string containing the class definition of the named class.
<code>listVariables()</code>	Return a vector of the names of the currently defined variables.
<code>loadModel(classname)</code>	Load model or package of name <i>classname</i> from MODELICAPATH.
<code>loadFile(str)</code>	Load Modelica file (.mo) with name given as string argument <i>str</i> .
<code>readFile(str)</code>	Load file given as string <i>str</i> and return a string containing the file content.
<code>runScript(str)</code>	Execute script file with file name given as string argument <i>str</i> .
<code>system(str)</code>	Execute <i>str</i> 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.
<code>timing(expr)</code>	Evaluate expression <i>expr</i> and return the number of seconds (elapsed time) the evaluation took.
<code>typeof(variable)</code>	Return the type of the <i>variable</i> as a string.
<code>saveModel(str,modelname)</code>	Save the model/class with name <i>modelname</i> in the file given by the string argument <i>str</i> .
<code>help()</code>	Print this helptext (returned as a string).
<code>quit()</code>	Leave and quit the OpenModelica environment

Chapter 2

Using the Graphical Model Editor

This chapter just presents a very simple example of using graphical modeling of Modelica models. A model is built using the graphical model editor by using drag-and-drop of already developed and freely available model components from the Modelica Standard Library.

NOTE: *This chapter is just a short sample of using the graphical model editor. See www.mathcore.com for the current manual and the complete MathModelica Graphic Model Editor Users Guide. As mentioned previously, the graphic editor is not part of OpenModelica, but an OpenModelica Edition of the GraphicEditor that works together with OpenModelica can be downloaded. (The OpenModelica edition of the editor is free for non-commercial usage, and will soon be available; now i beta-test).*

The Modelica Standard Library can be 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.

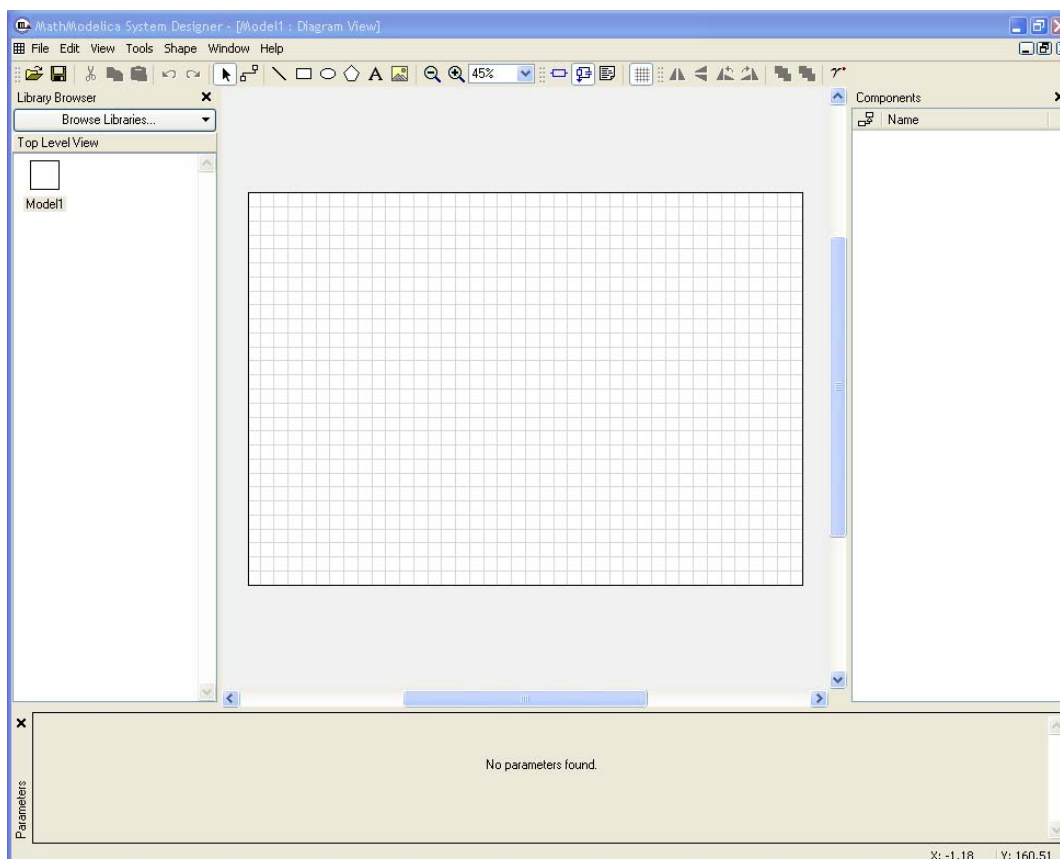


Figure 2-1. The Graphical Model Editor with the class browser to the left, the graphic editing area in the middle and the instance component browser to the right.

To open the library, click on the `Browse libraries` button in the class browser to the left. As shown by Figure 2-2, the Modelica Standard Library is hierarchically structured into sublibraries.

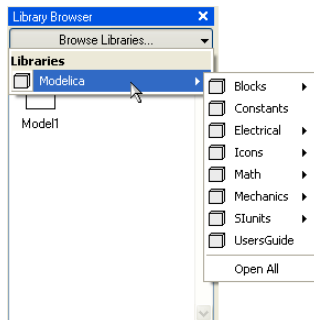


Figure 2-2. The Graphical Model Editor with the class browser showing the Modelica Standard library opened up into sublibraries.

The following list briefly mentions some of the most important sublibraries in the Modelica standard library, as well as the Users Guide:

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

2.1 Building a Simple DCMotor 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.

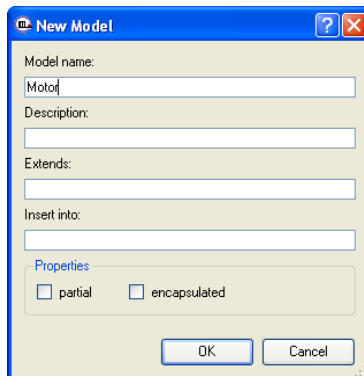


Figure 2-3. Dialog box for creating a new model.

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.

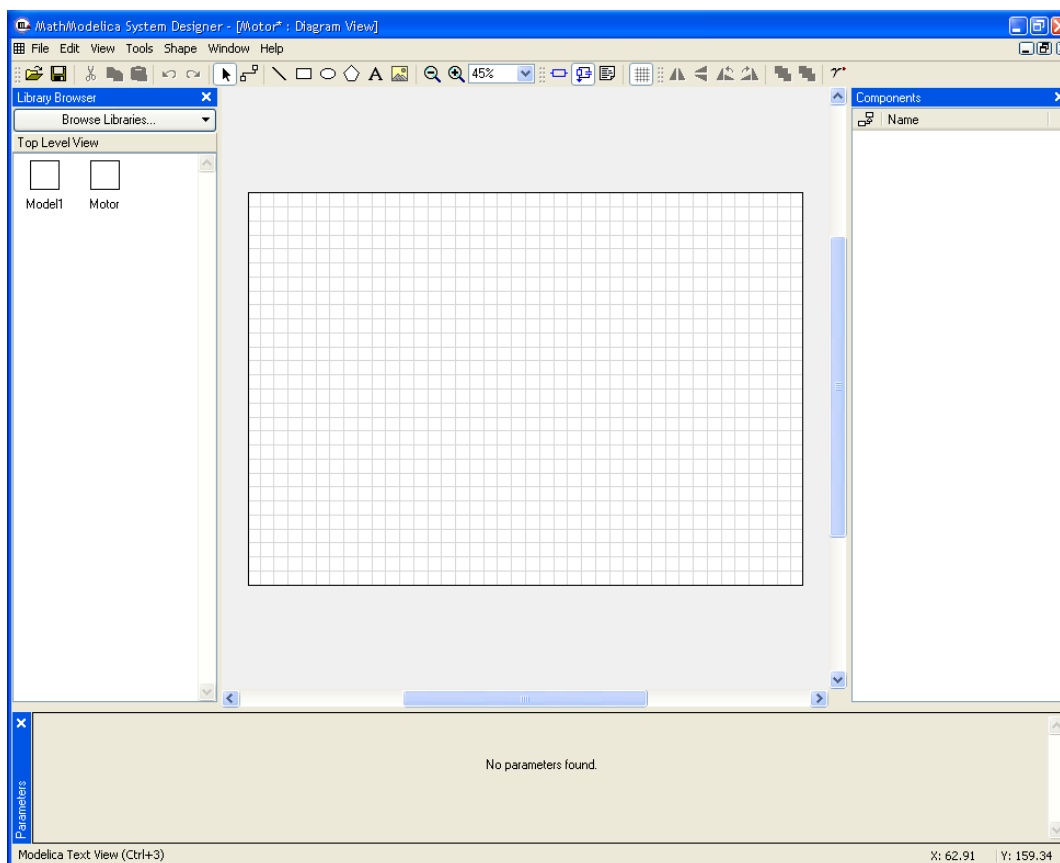


Figure 2-4. The Graphical Model Editor with the new `Motor` model appearing as a question mark icon in the class browser window to the left.

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.

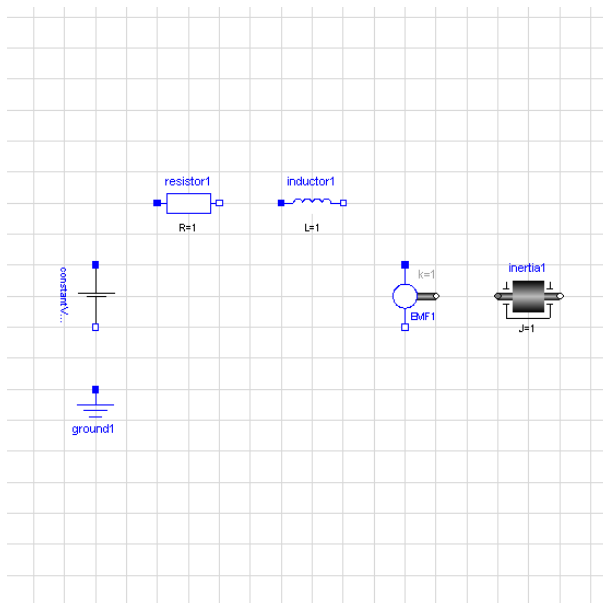


Figure 2-5. Several components dragged into the diagram view of the Graphic Model Editor.

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 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.

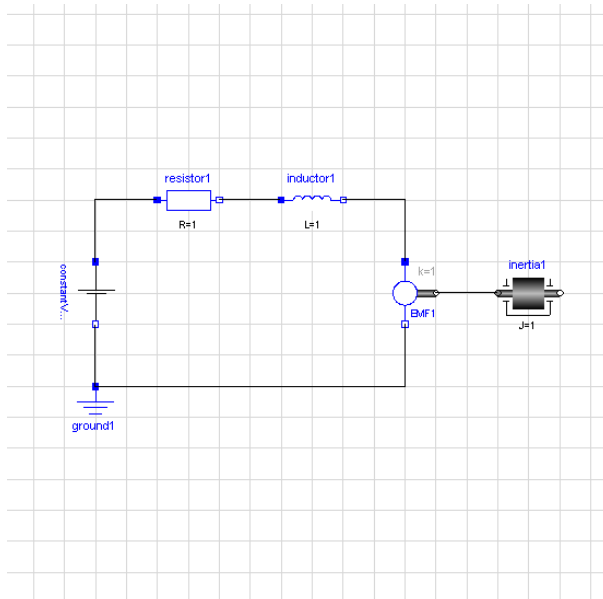


Figure 2-6. The components connected into a simple DC motor model.

2.1.2 The Graphic Editor Text View

The Modelica code of a Model can also be viewed and edited using the Graphic Editor text view (Figure 2-7):

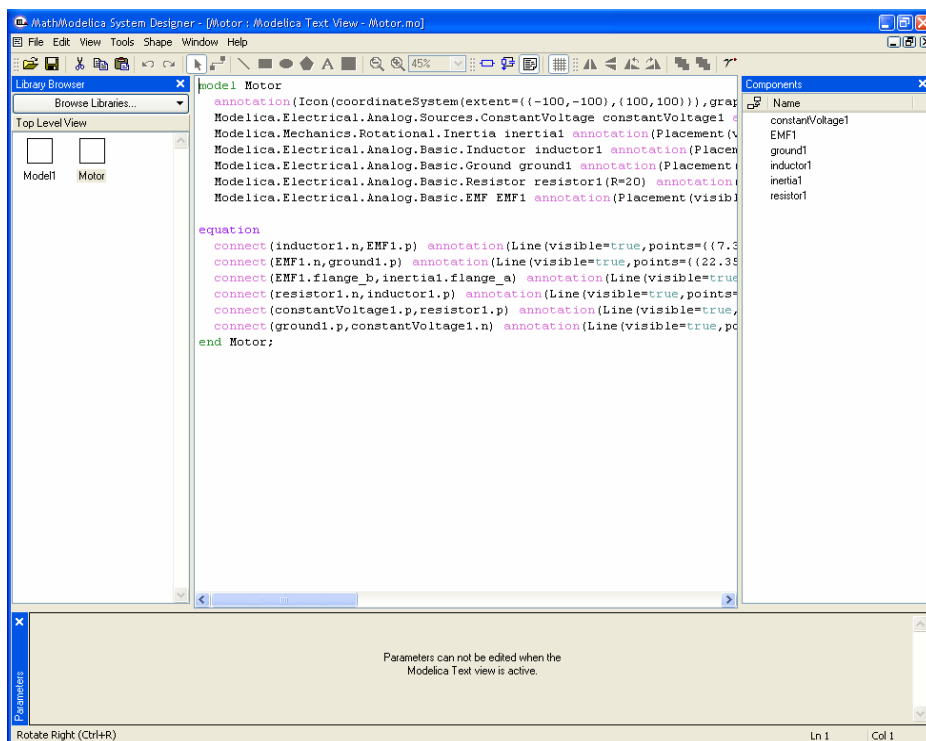


Figure 2-7. Graphic Model Editor text view.

2.1.3 Plotting

After the model has been translated and simulated, any of its variables can be plotted (Figure 2-8). Plotting from the Graphic Model Editor is not available in the free OpenModelica Edition. Instead, plotting can be made from the command line through the plot command, as in the examples shown in Section 1.2.

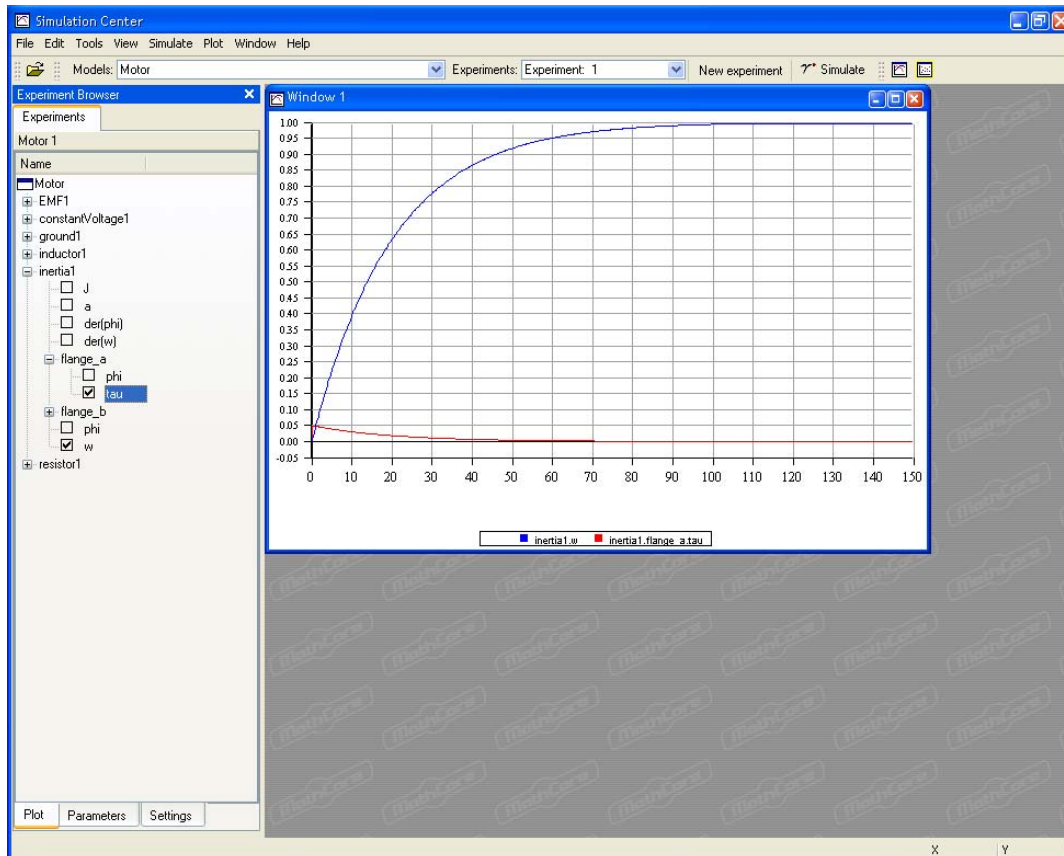


Figure 2-8. MathModelica plot window created after a simulation of the `Motor` model, using the full version of the MathModelica Graphic Model Editor (plotting not available in the free OpenModelica edition).

Chapter 3

OMNotebook with DrModelica

This chapter covers the OpenModelica electronic notebook subsystem, called OMNotebook, together with the DrModelica tutoring system for teaching Modelica, which is using such notebooks.

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.

3.1.1 Mathematica Notebooks

Literate Programming (Knuth 1984) is a form of programming where programs are integrated with documentation in the same document. Mathematica notebooks (Wolfram 1997) is one of the first WYSIWYG (What-You-See-Is-What-You-Get) systems that support Literate Programming. Such notebooks are used, e.g., in the MathModelica modeling and simulation environment, e.g. see Figure 3-1 below and Chapter 19 in (Fritzson 2004)

3.1.2 OMNotebook

The OMNotebook software (Axelsson 2005, Fernström 2006) is a new open source free software that gives 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.

The OMNotebook 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. OMNotebook is a simple open-source software tool for an electronic notebook supporting Modelica.

A more advanced electronic notebook tool, also supporting mathematical typesetting and many other facilities, is provided by Mathematica notebooks in the MathModelica environment, see Figure 3-1.

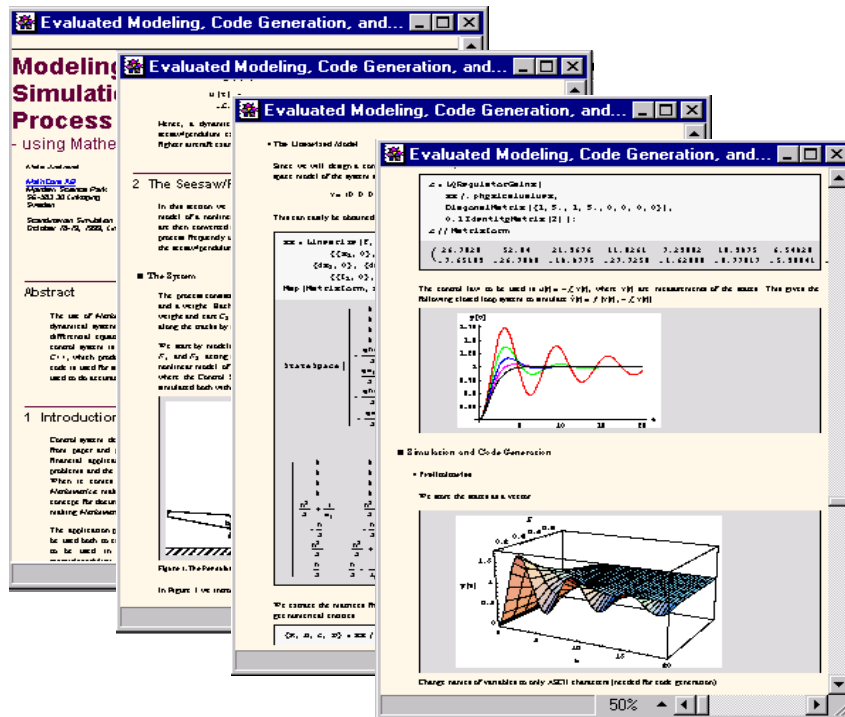


Figure 3-1. Examples of Mathematica notebooks in the MathModelica modeling and simulation environment.

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..

3.2 The DrModelica Tutoring System – an Application of OMNotebook

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.

An earlier version of DrModelica was developed using the MathModelica environment. The rest of this chapter is concerned with the OMNotebook version of DrModelica and on the OMNotebook tool itself.

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 3-2).

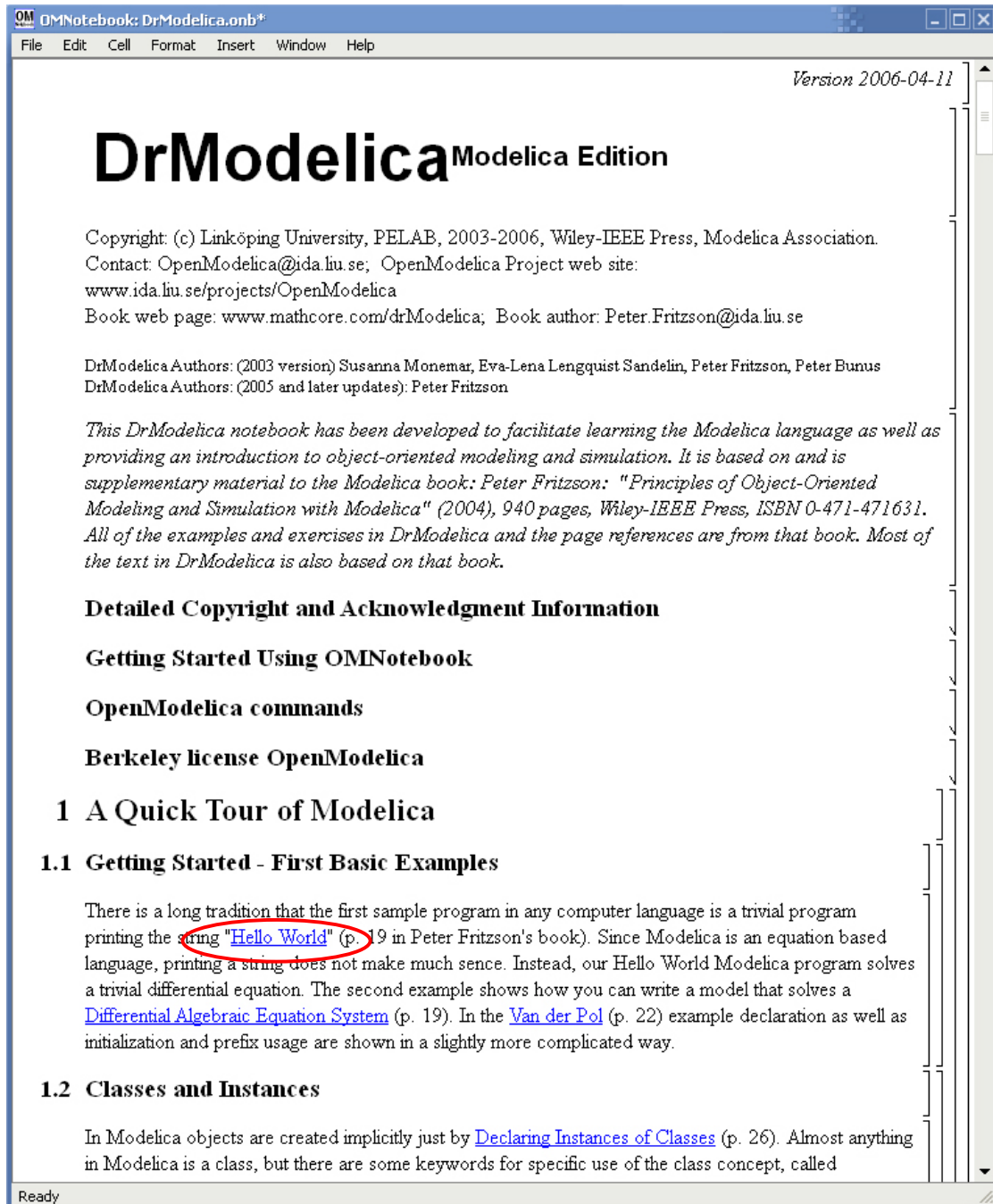


Figure 3-2. The front-page notebook of the OMNotebook version of the DrModelica tutoring system.

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.

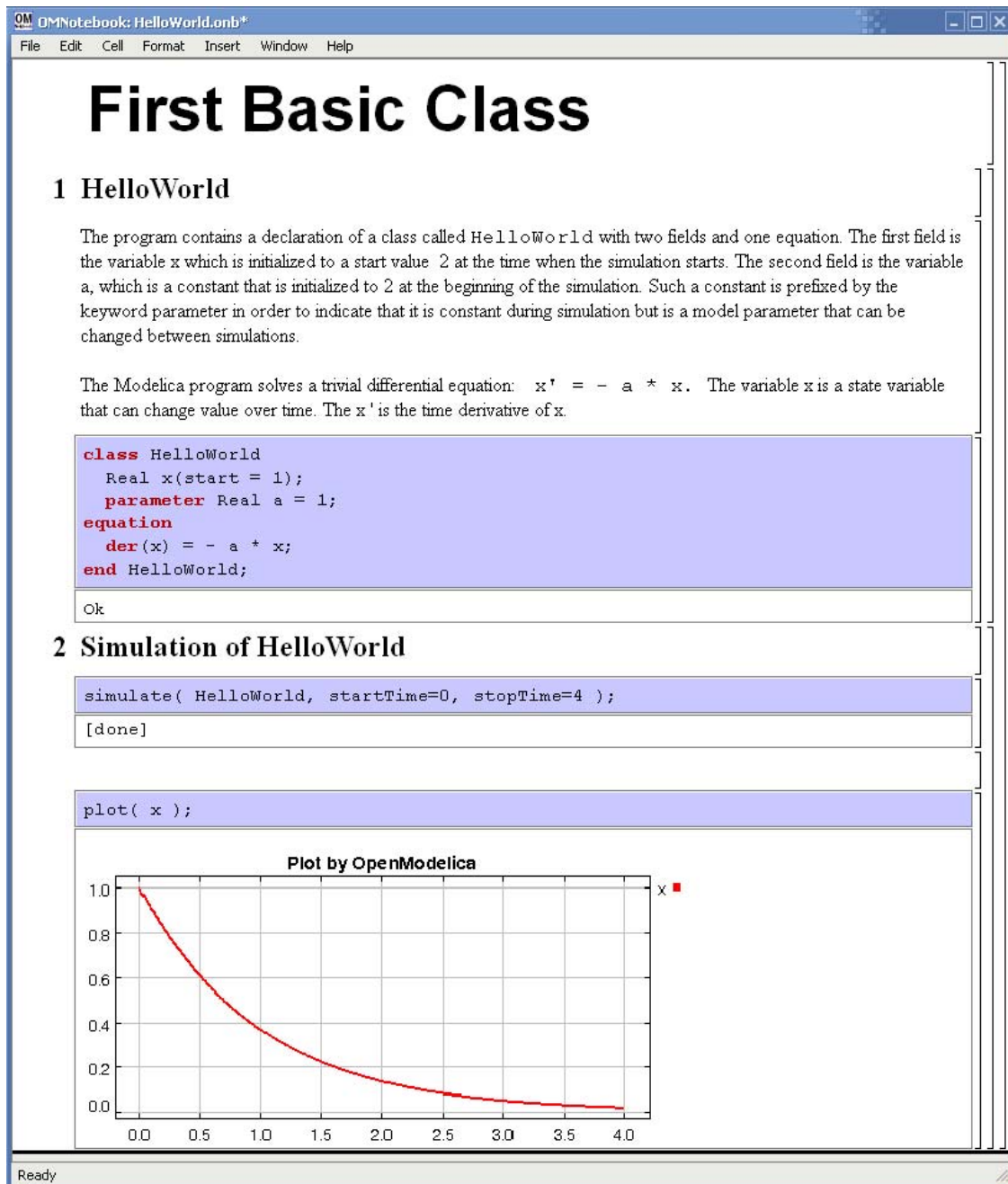


Figure 3-3. The HelloWorld class simulated and plotted using the OMNotebook version of DrModelica.

Now, let us consider that the link “HelloWorld” in DrModelica Section is clicked by the user. The new HelloWorld notebook (see Figure 3-3), to which the user is being linked, 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.

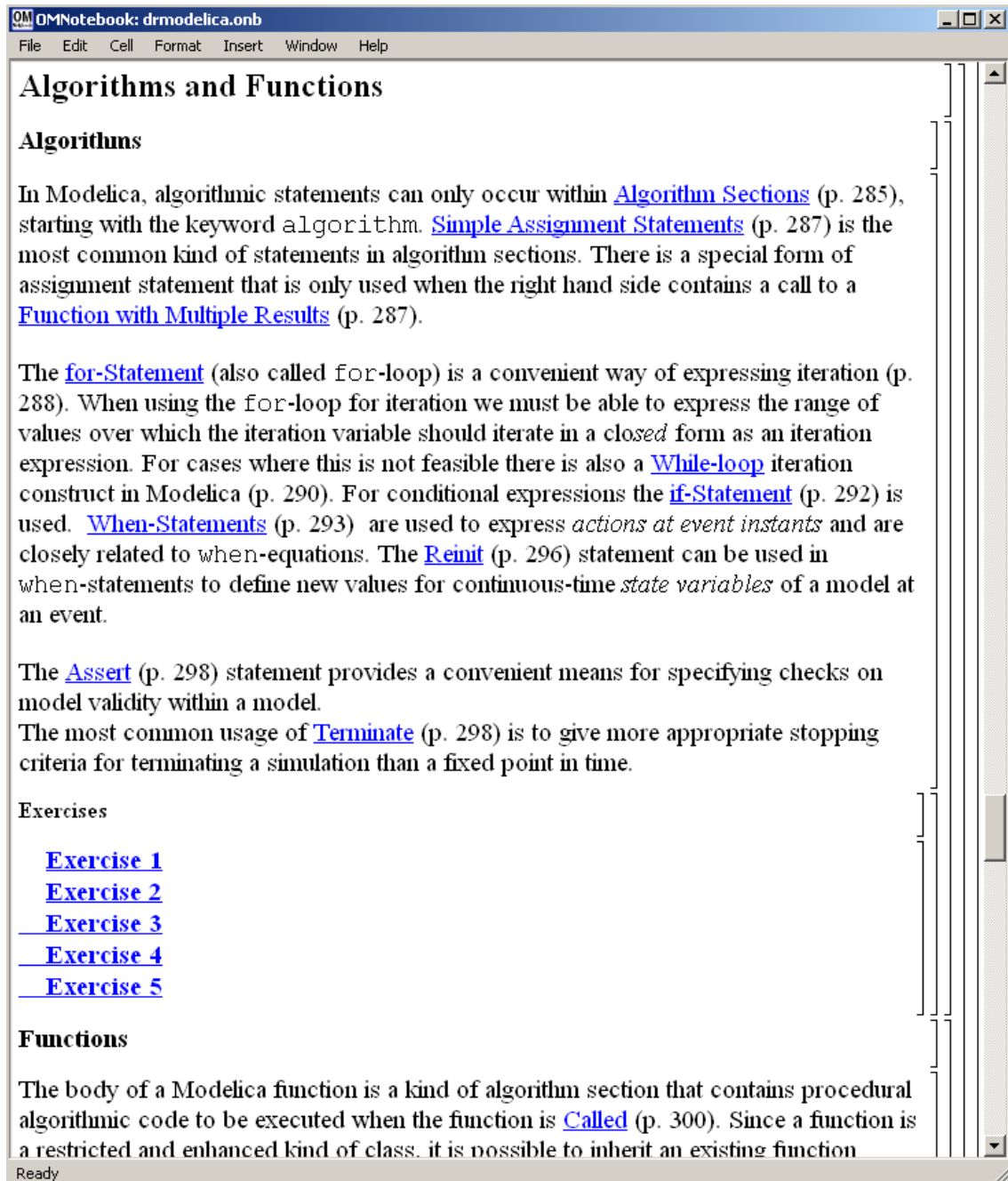


Figure 3-4. DrModelica Chapter on Algorithms and Functions in the main page of the OMNotebook version of DrModelica.

When a class has been successfully evaluated the user can simulate and plot the result, as previously depicted in Figure 3-3 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 3-4 shows part of 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.

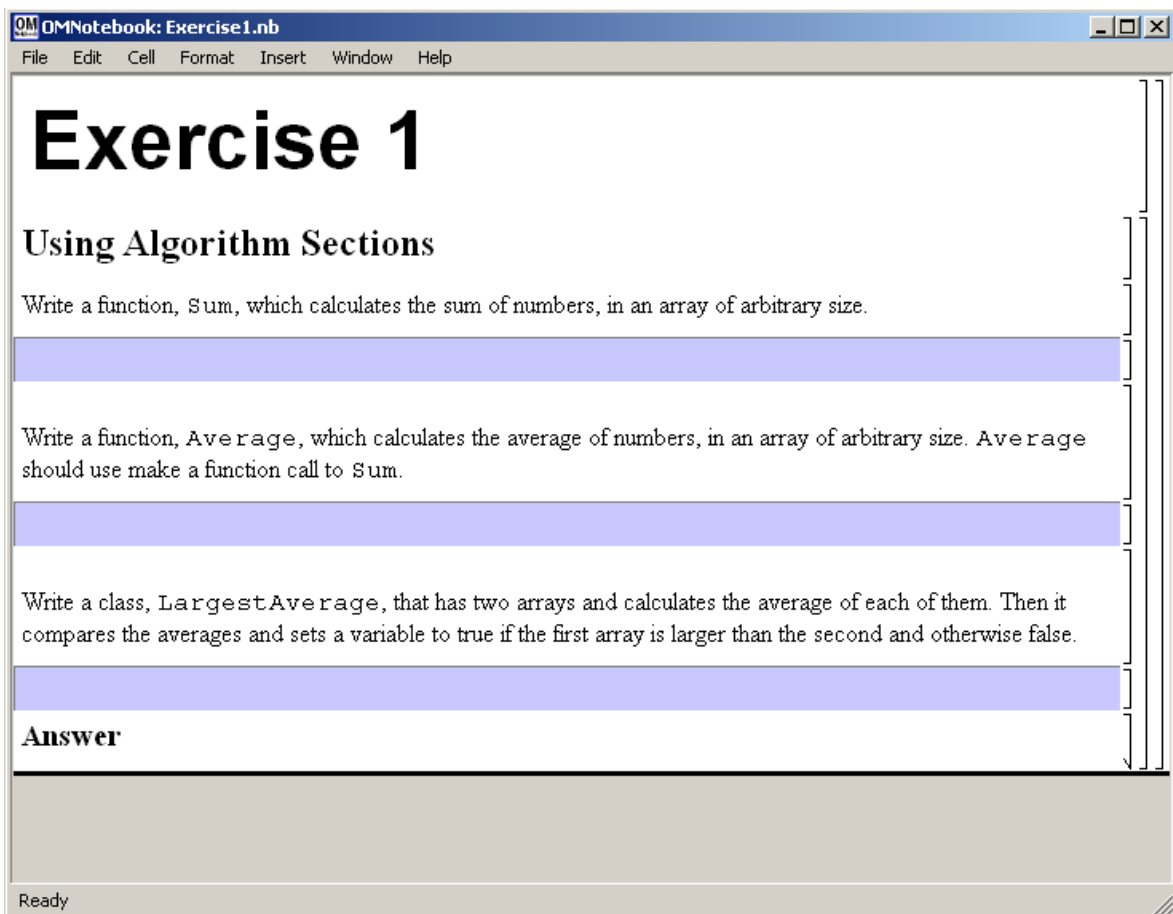


Figure 3-5. Exercise 1 in Chapter 9 of DrModelica.

Exercise 1 from Chapter 9 is shown in Figure 3-5. 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 3-6.

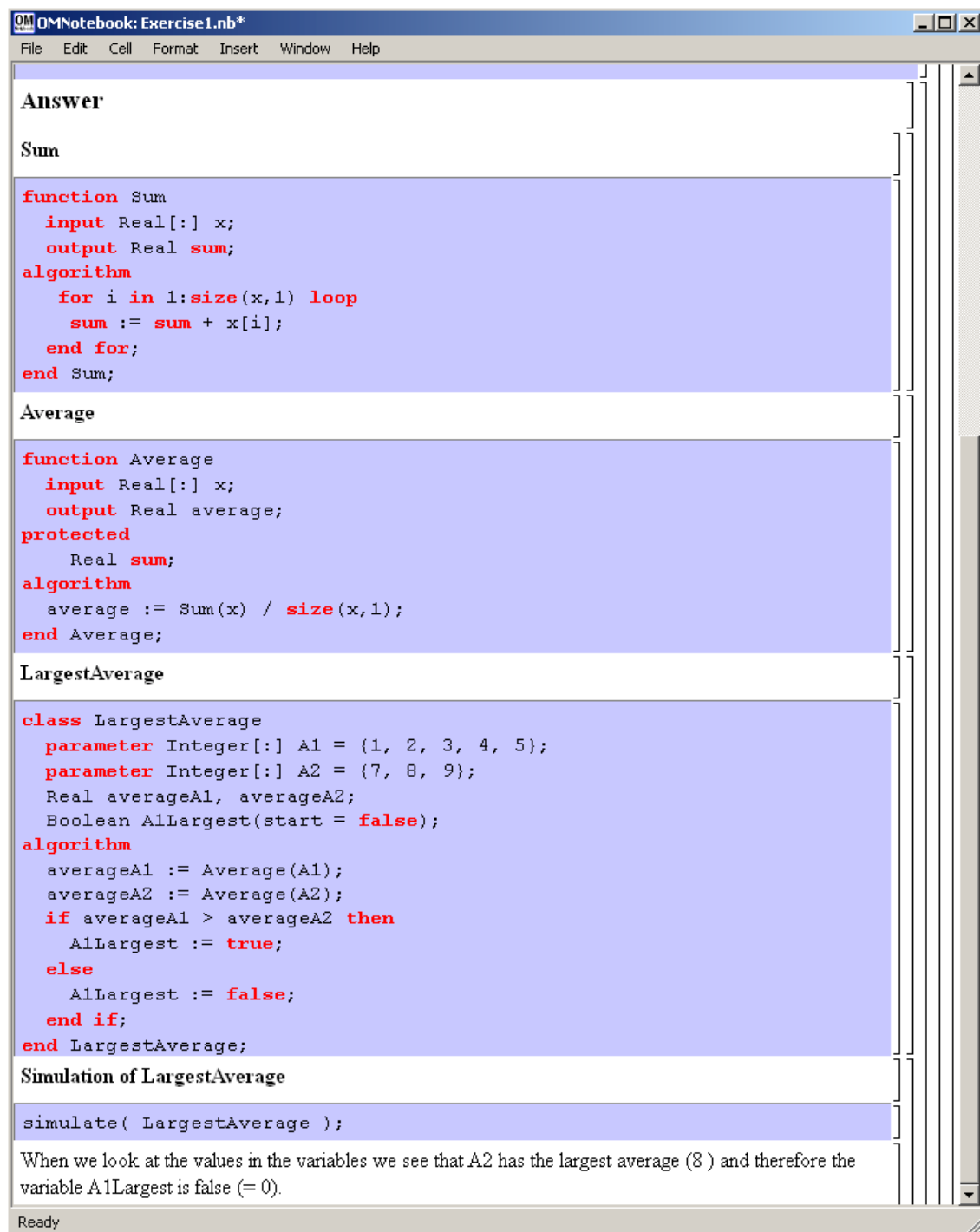


Figure 3-6. The answer section to Exercise 1 in Chapter 9 of DrModelica.

3.3 OpenModelica Notebook Commands

OMNotebook currently supports the commands and concepts that are summarized in this section.

3.3.1 Cells

Everything inside an OMNotebook document is made out of cells. A cell basically contains a chunk of data. That data can be text, images, or other cells. OMNotebook has four types of cells: `headercell`, `textcell`, `inputcell`, and `groupcell`. Cells are ordered in a tree structure, where one cell can be a parent to one or more additional cells. A tree view is available close to the right border in the notebook window to display the relation between the cells.

- *Textcell* – This cell type are used to display ordinary text and images. Each `textcell` has a style that specifies how text is displayed. The cell's style can be changed in the menu `Format->Styles`, example of different styles are: `Text`, `Title`, and `Subtitle`. The `Textcell` type also has support for following links to other notebook documents.
- *Inputcell* – This cell type has support for syntax highlighting and evaluation. It is intended to be used for writing program code, e.g. Modelica code. Evaluation is done by pressing the key combination `Shift+Return` or `Shift+Enter`. All the text in the cell is sent to OMC (OpenModelica Compiler/interpreter), where the text is evaluated and the result is displayed below the `inputcell`. By double-clicking on the cell marker in the tree view, the `inputcell` can be collapsed causing the result to be hidden.
- *Groupcell* – This cell type is used to group together other cell. A `groupcell` can be opened or closed. When a `groupcell` is opened all the cells inside the `groupcell` are visible, but when the `groupcell` is closed only the first cell inside the `groupcell` is visible. The state of the `groupcell` is changed by the user double-clicking on the cell marker in the tree view. When the `groupcell` is closed the marker is changed and the marker has an arrow at the bottom.

3.3.2 Cursors

An OMNotebook document contains cells which in turn contain text. Thus, two kinds of cursors are needed for positioning, text cursor and cell cursor:

- *Textcursor* – A cursor between characters in a cell, appearing as a small vertical line. Position the cursor by clicking on the text or using the arrow buttons.
- *Cellcursor* – This cursor shows which cell currently has the input focus. The cursor is basically just a thin black horizontal line. The `cellcursor` is positioned by clicking on a cell, clicking between cells, or using the menu item `Cell->Next Cell` or `Cell->Previous Cell`. The cursor can also be moved with the key combination `Ctrl+Up` or `Ctrl+Down`.

3.4 Selection of Text or Cells

To perform operations on text or cells we often need to select a range of characters or cells.

- *Select characters* – There are several ways of selecting characters, e.g. double-clicking on a word, clicking and dragging the mouse, or click followed by a shift-click at an adjacent position selects the text between the previous click and the position of the most recent shift-click.
- *Select cells* – Cells can be selected by clicking on them. Holding down `Ctrl` and clicking on the cell markers in the tree view allows several cells to be selected, one at a time. Several cells can be selected at once in the tree view by holding down the `Shift` key. Holding down `Shift` selects all cells between last selected cell and the cell clicked on. This only works if both cells belong to the same `groupcell`.

3.4.1 File Menu

The following file related operations are available in the file menu:

- *Create a new notebook* – A new notebook can be created using the menu `File->New` or the key combination `Ctrl+N`. A new document window will then open, with a new document inside.
- *Open a notebook* – To open a notebook use `File->Open` in the menu or the key combination `Ctrl+O`. Only files of the type `.onb` or `.nb` can be opened. If a file does not follow the OMNotebook format or the FullForm Mathematica Notebook format, a message box is displayed telling the user what is wrong. Mathematica Notebooks must be converted to fullform before they can be opened in OMNotebook.
- *Save a notebook* – To save a notebook use the menu item `File->Save` or `File->Save As`. If the notebook has not been saved before the save as dialog is shown and a filename can be selected. OMNotebook can only save in xml format and the saved file is not compatible with Mathematica. Key combination for save is `Ctrl+S` and for save as `Ctrl+Shift+S`. The saved file by default obtains the file extension `.onb`.
- *Print* – Printing a document to a printer is done by pressing the key combination `Ctrl+P` or using the menu item `File->Print`. A normal print dialog is displayed where the usually properties can be changed.
- *Import old document* – Old documents, saved with the old version of OMNotebook where a different file format was used, can be opened using the menu item `File->Import->Old OMNotebook file`. Old documents have the extension `.xml`.
- *Export text* – The text inside a document can be exported to a text document. The text is exported to this document without almost any structure saved. The only structure that is saved is the cell structure. Each paragraph in the text document will contain text from one cell. To use the export function, use menu item `File->Export->Pure Text`.
- *Close a notebook window* – A notebook window can be closed using the menu item `File->Close` or the key combination `Ctrl+F4`. Any unsaved changes in the document are lost when the notebook window is closed.
- *Quitting OMNotebook* – To quit OMNotebook, use menu item `File->Quit` or the key combination `Ctrl+Q`. This closes all notebook windows; users will have the option of closing OMC also. OMC will not automatically shutdown because other programs may still use it. Evaluating the command `quit()` has the same result as exiting OMNotebook.

3.4.2 Edit Menu

- *Editing cell text* – Cells have a set of of basic editing functions. The key combination for these are: Undo (`Ctrl+Z`), Redo (`Ctrl+Y`), Cut (`Ctrl+X`), Copy (`Ctrl+C`) and Paste (`Ctrl+V`). These functions can also be accessed from the edit menu; Undo (`Edit->Undo`), Redo (`Edit->Redo`), Cut (`Edit->Cut`), Copy (`Edit->Copy`) and Paste (`Edit->Paste`). Selection of text is done in the usual way by double-clicking, triple-clicking (select a paragraph), dragging the mouse, or using (`Ctrl+A`) to select all text within the cell.
- *Cut cell* – Cells can be cut from a document with the menu item `Edit->Cut` or the key combination `Ctrl+X`. The cut function will always cut cells if cells have been selected in the tree view, otherwise the cut function cuts text.
- *Copy cell* – Cells can be copied from a document with the menu item `Edit->Copy` or the key combination `Ctrl+C`. The copy function will always copy cells if cells have been selected in the tree view, otherwise the copy function copy text.
- *Paste cell* – To paste copied or cut cells the cell cursor must be selected in the location where the cells should be pasted. This is done by clicking on the cell cursor. Pasting cells is done from the

menu Edit->Paste or the key combination Ctrl+V. If the cell cursor is selected the paste function will always paste cells. OMNotebook share the same application-wide clipboard. Therefore cells that have been copied from one document can be pasted into another document. Only pointers to the copied or cut cells are added to the clipboard, thus the cell that should be pasted must still exist. Consequently a cell can not be pasted from a document that has been closed.

- *View expression* – Text in a cell is stored internally as a subset of HTML code and the menu item Edit->View Expression let the user switch between viewing the text or the internal HTML representation. Changes made to the HTML code will affect how the text is displayed.

3.4.3 Cell Menu

- *Add textcell* – A new textcell is added with the menu item Cell->Add Cell (previous cell style) or the key combination Alt+Enter. The new textcell gets the same style as the previous selected cell had.
- *Add inputcell* – A new inputcell is added with the menu item Cell->Add Inputcell or the key combination Ctrl+Shift+I.
- *Add groupcell* – A new groupcell is inserted with the menu item Cell->Groupcell or the key combination Ctrl+Shift+G. The selected cell will then become the first cell inside the groupcell.
- *Ungroup groupcell* – A groupcell can be ungrouped by selecting it in the tree view and using the menu item Cell->Ungroup Groupcell or by using the key combination Ctrl+Shift+U. Only one groupcell at a time can be ungrouped.
- *Split cell* – Splitting a cell is done with the menu item Cell->Split cell or the key combination Ctrl+Shift+P. The cell is splitted at the position of the text cursor.
- *Delete cell* – The menu item Cell->Delete Cell will delete all cells that have been selected in the tree view. If no cell is selected this action will delete the cell that have been selected by the cellcursor. This action can also be called with the key combination Ctrl+Shift+D or the key Del (only works when cells have been selected in the tree view).
- *Cellcursor* – This cell type is a special type that shows which cell that currently has the focus. The cell is basically just a thin black line. The cellcursor is moved by clicking on a cell or using the menu item Cell->Next Cell or Cell->Previous Cell. The cursor can also be moved with the key combination Ctrl+Up or Ctrl+Down.

3.4.4 Format Menu

- *Textcell* – This cell type is used to display ordinary text and images. Each textcell has a style that specifies how text is displayed. The cells style can be changed in the menu Format->Styles, examples of different styles are: Text, Title, and Subtitle. The Textcell type also have support for following links to other notebook documents.
- *Text manipulation* – There are a number of different text manipulations that can be done to change the appearance of the text. These manipulations include operations like: changing font, changing color and make text bold, but also operations like: changing the alignment of the text and the margin inside the cell. All text manipulations inside a cell can be done on single letters, words or the entire text. Text settings are found in the Format menu. The following text manipulations are available in OMNotebook:
 - > Font family
 - > Font face (Plain, Bold, Italic, Underline)
 - > Font size
 - > Font stretch
 - > Font color

- > Text horizontal alignment
- > Text vertical alignment
- > Border thickness
- > Margin (outside the border)
- > Padding (inside the border)

3.4.5 Insert Menu

- *Insert image* – Images are added to a document with the menu item `Insert->Image` or the key combination `Ctrl+Shift+M`. After an image has been selected a dialog appears, where the size of the image can be chosen. The images actual size is the default value of the image. OMNotebook stretches the image accordantly to the selected size. All images are saved in the same file as the rest of the document.
- *Insert link* – A document can contain links to other OMNotebook file or Mathematica notebook and to add a new link a piece of text must first be selected. The selected text make up the part of the link that the user can click on. Inserting a link is done from the menu `Insert->Link` or with the key combination `Ctrl+Shift+L`. A dialog window, much like the one used to open documents, allows the user to choose the file that the link refers to. All links are saved in the document with a relative file path so documents that belong together easily can be moved from one place to another without the links failing.

3.4.6 Window Menu

- *Change window* – Each opened document has its own document window. To switch between those use the Window menu. The window menu lists all titles of the open documents, in the same order as they were opened. To switch to another document, simple click on the title of that document.

3.4.7 Help Menu

- *About OMNotebook* – Accessing the about message box for OMNotebook is done from the menu `Help->About OMNotebook`.
- *About Qt* – To access the message box for Qt, use the menu `Help->About Qt`.
- *Help Text* – Opening the help text (document `OMNotebookHelp.onb`) for OMNotebook can be done in the same way as any OMNotebook document is opened or with the menu `Help->Help Text`. The menu item can also be triggered with the key `F1`.

3.4.8 Additional Features

- *Links* – By clicking on a link, OMNotebook will open the document that is referred to in the link.
- *Update link* – All links are stored with relative file path. Therefore OMNotebook has functions that automatically updating links if a document is resaved in another folder. Every time a document is saved, OMNotebook checks if the document is saved in the same folder as last time. If the folder has changed, the links are updated.
- *Evaluate several cells* – Several inputcells can be evaluated at the same time by selecting them in the treeview and then pressing the key combination `Shift+Enter` or `Shift+Return`. The cells are evaluated in the same order as they have been selected. If a groupcell is selected all inputcells in that groupcell are evaluated, in the order they are located in the groupcell.

- *Command completion* – Inputcells have command completion support, which checks if the user is typing a command (or any keyword defined in the file `commands.xml`) and finish the command. If the user types the first two or three letters in a command, the command completion function fills in the rest. To use command completion, press the key combination `Ctrl+Space` or `Shift+Tab`. The first command that matches the letters written will then appear. Holding down `Shift` and pressing `Tab` (alternative holding down `Ctrl` and pressing `Space`) again will display the second command that matches. Repeated request to use command completion will loop through all commands that match the letters written. When a command is displayed by the command completion functionality any field inside the command that should be edited by the user is automatically selected. Some commands can have several of these fields and by pressing the key combination `Ctrl+Tab`, the next field will be selected inside the command.
 - > Active Command completion: `Ctrl+Space / Shift+Tab`
 - > Next command: `Ctrl+Space / Shift+Tab`
 - > Next field in command: `Ctrl+Tab`
- *Generated plot* – When plotting a simulation result, OMC uses the program `Ptplot` to create a plot. From `Ptplot` OMNotebook gets an image of the plot and automatically adds that image to the output part of an inputcell. Like all other images in a document, the plot is saved in the document file when the document is saved.
- *Stylesheet* – OMNotebook follows the style settings defined in `stylesheet.xml` and the correct style is applied to a cell when the cell is created.
- *Automatic Chapter Numbering* – OMNotebook automatically numbers different chapter, subchapter, section and other styles. The user can specify which styles should have chapter numbers and which level the style should have. This is done in the `stylesheet.xml` file. Every style can have a `<chapterLevel>` tag that specifies the chapter level. Level 0 or no tag at all, means that the style should not have any chapter numbering.
- *Scrollarea* – Scrolling through a document can be done by using the mouse wheel. A document can also be scrolled by moving the cell cursor up or down.
- *Syntax highlighter* – The syntax highlighter runs in a separated thread which speeds up the loading of large document that contains many Modelica code cells. The syntax highlighter only highlights when letters are added, not when they are removed. The color settings for the different types of keywords are stored in the file `modelicacolors.xml`. Besides defining the text color and background color of keywords, whether or not the keywords should be bold or/and italic can be defined.
- *Change indicator* – A star (*) will appear behind the filename in the title of notebook window if the document has been changed and needs saving. When the user closes a document that has some unsaved change, OMNotebook asks the user if he/she wants to save the document before closing. If the document never has been saved before, the save-as dialog appears so that a filename can be choosen for the new document.
- *Update menus* – All menus are constantly updated so that only menu items that are linked to actions that can be performed on the currently selected cell is enabled. All other menu items will be disabled. When a textcell is selected the Format menu is updated so that it indicates the text settings for the text, in the current cursor position.

3.5 References

Eric Allen, Robert Cartwright, Brian Stoler. DrJava: A lightweight pedagogic environment for Java. In Proceedings of the 33rd ACM Technical Symposium on Computer Science Education (SIGCSE 2002) (Northern Kentucky – The Southern Side of Cincinnati, USA, February 27 – March 3, 2002).

Ingemar Axelsson. OpenModelica Notebook for Interactive Structured Modelica Documents. Final thesis, LITH-IDA-EX-05/080-SE, Linköping University, Linköping, Sweden, October 21, 2005.

Anders Fernström, Ingemar Axelsson, Peter Fritzson, Anders Sandholm, Adrian Pop. OMNotebook – Interactive WYSIWYG Book Software for Teaching Programming. In Proc. of the Workshop on Developing Computer Science Education – How Can It Be Done?. Linköping University, Dept. Computer & Inf. Science, Linköping, Sweden, March 10, 2006.

Anders Fernström. Extending OMNotebook – An Interactive Notebook for Structured Modelica Documents. Final thesis to be presented spring 2006, Dept. Computer and Information Science, Linköping University, Sweden.

Peter Fritzson. Principles of Object Oriented Modeling and Simulation with Modelica 2.1, 940 pages, ISBN 0-471-471631, Wiley-IEEE Press. Feb. 2004.

Knuth, Donald E. Literate Programming. The Computer Journal, NO27(2), pp. 97–111, May 1984.

Eva-Lena Lengquist-Sandelin, Susanna Monemar, Peter Fritzson, and Peter Bunus. DrModelica – A Web-Based Teaching Environment for Modelica. In Proceedings of the 44th Scandinavian Conference on Simulation and Modeling (SIMS'2003), available at www.scan-sims.org. Västerås, Sweden. September 18-19, 2003.

The Modelica Association. The Modelica Language Specification Version 2.2, March 2005. <http://www.modelica.org>.

Stephen Wolfram. The Mathematica Book. Wolfram Media Inc, 1997.

Chapter 4

Emacs Textual Model Editor/Browser

An Emacs Modelica mode provides facilities for keyword highlighting, suppressing annotations, etc. It can be downloaded from the OMDevelopers part of the OpenModelica web page www.ida.liu.se/projects/OpenModelica.

(?? 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. Currently not included in the Modelica mode.)

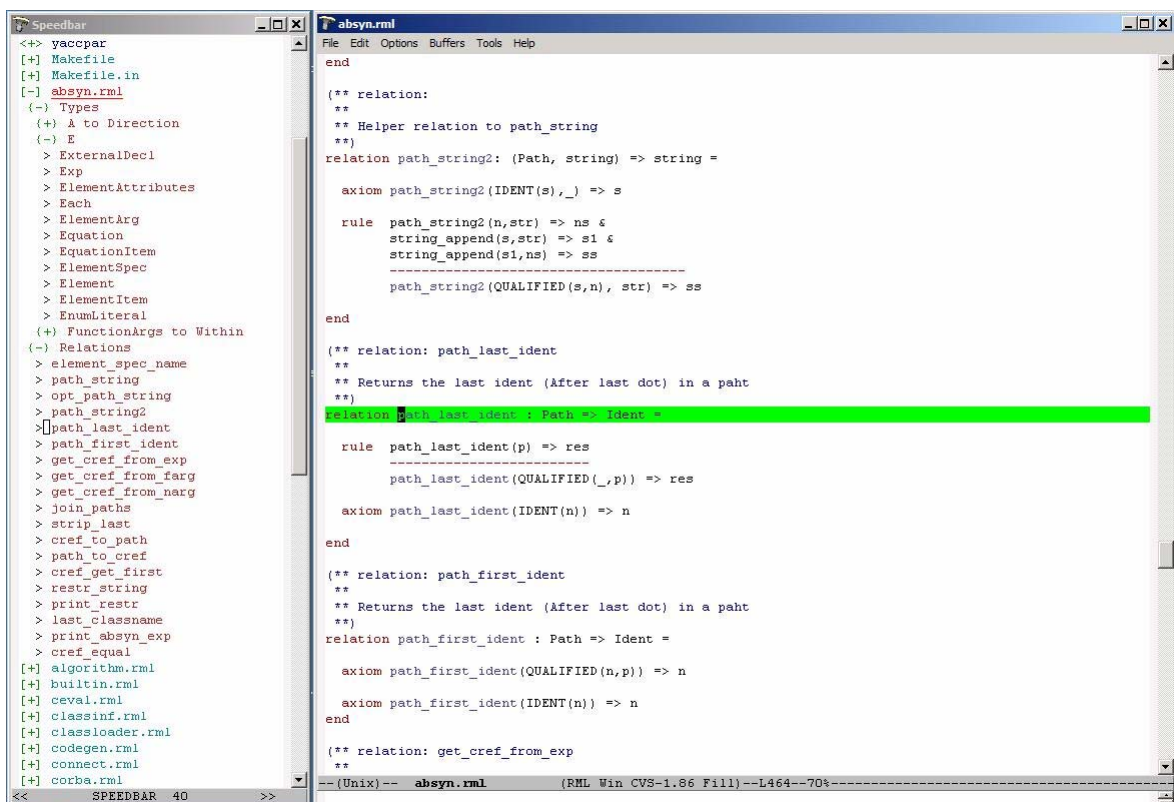


Figure 4-1. Emacs with a speedbar menu to the left, which allows clicking on file names (for expansion or closing the file contents menu). An expanded file shows all function, class, and type declarations. By clicking on one of those, you can position the editor at the appropriate definition.

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
- Code completion for class names and function argument lists.

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.

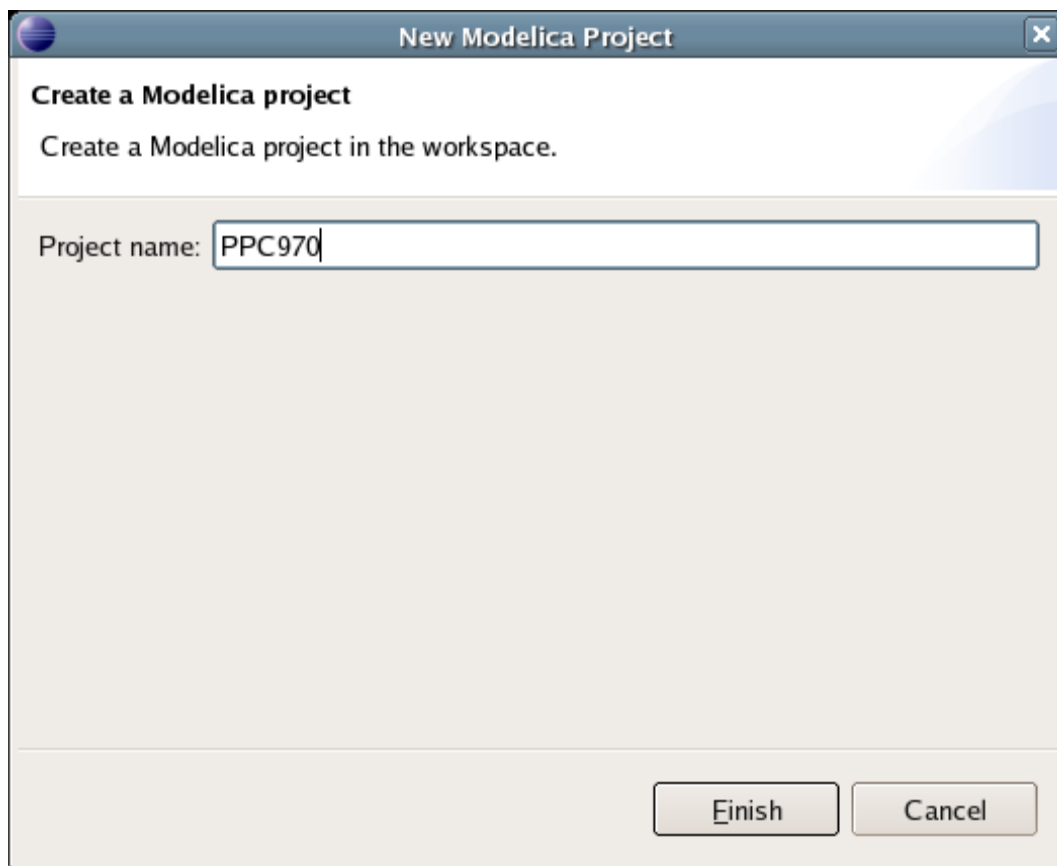
If you have problems using MDT, make sure that OPENMODELICAHOME is pointing to the folder where the Open Modelica Compiler is installed. In other words, OPENMODELICAHOME must point to the folder that contains the Open Modelica Compiler (OMC) binary. On the Windows platform it's called omc.exe and on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

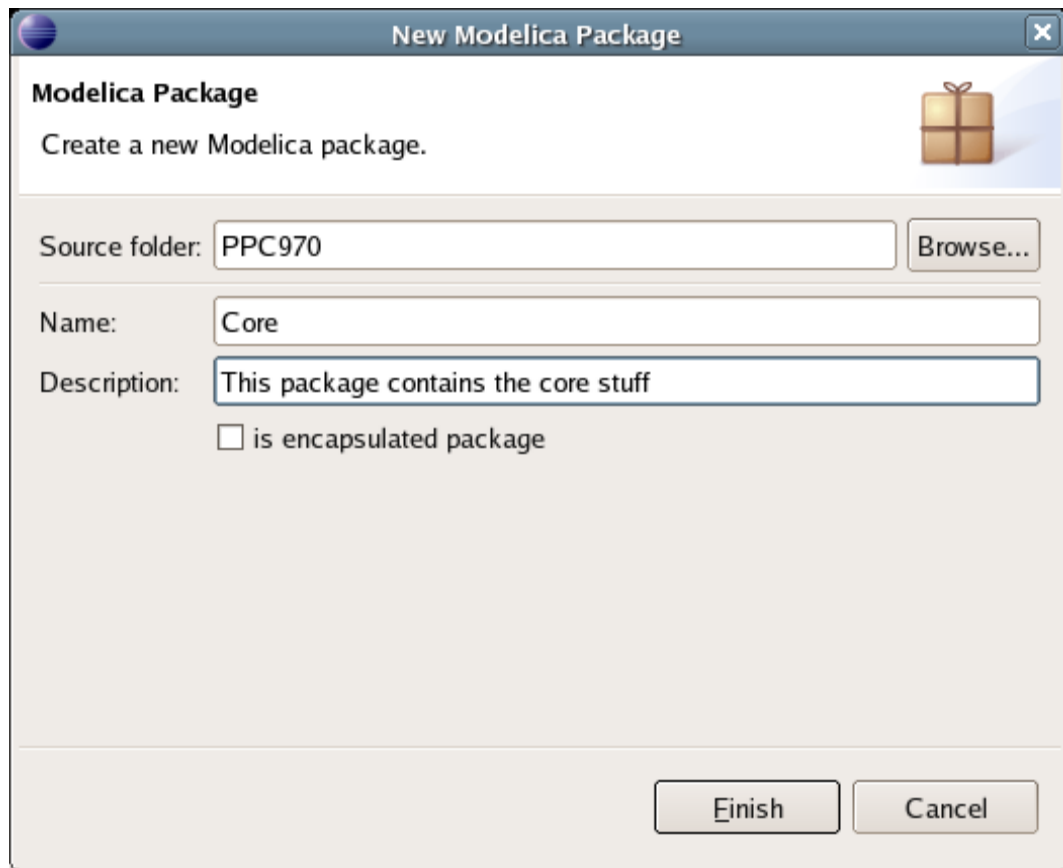
5.3.3 Creating a Project

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.



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.



The screenshot shows a dialog box titled "New Modelica Package" with a close button (X) in the top right corner. The dialog has a header section with the title "Modelica Package" and a subtitle "Create a new Modelica package." next to a gift icon. Below the header, there are three input fields: "Source folder:" with the text "PPC970" and a "Browse..." button; "Name:" with the text "Core"; and "Description:" with the text "This package contains the core stuff". Below the description field is a checkbox labeled "is encapsulated package" which is currently unchecked. At the bottom right of the dialog are two buttons: "Finish" and "Cancel".

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 Browse...

Name: ALU

Type: block ▼

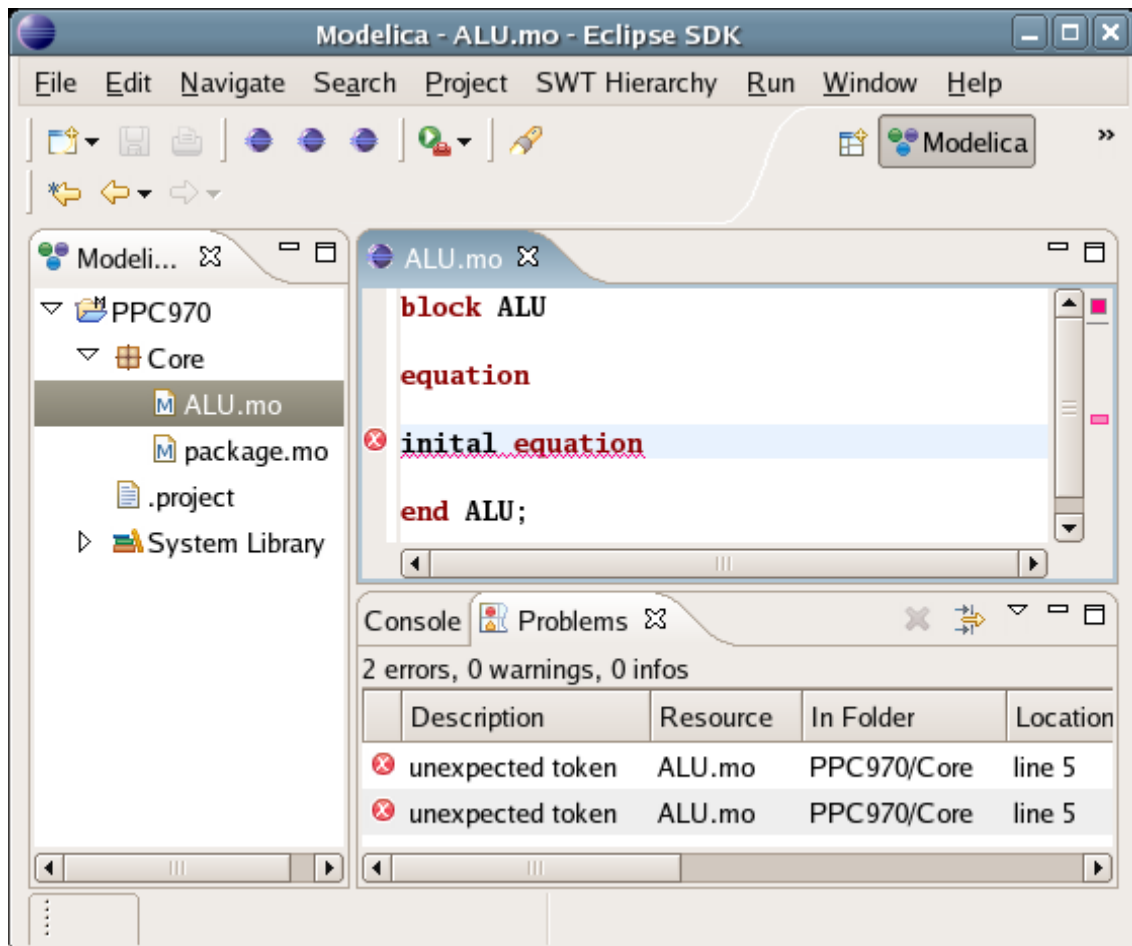
Modifiers:

- ☒ include initial equation block
- ☐ is partial class
- ☐ have external body

Finish Cancel

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.

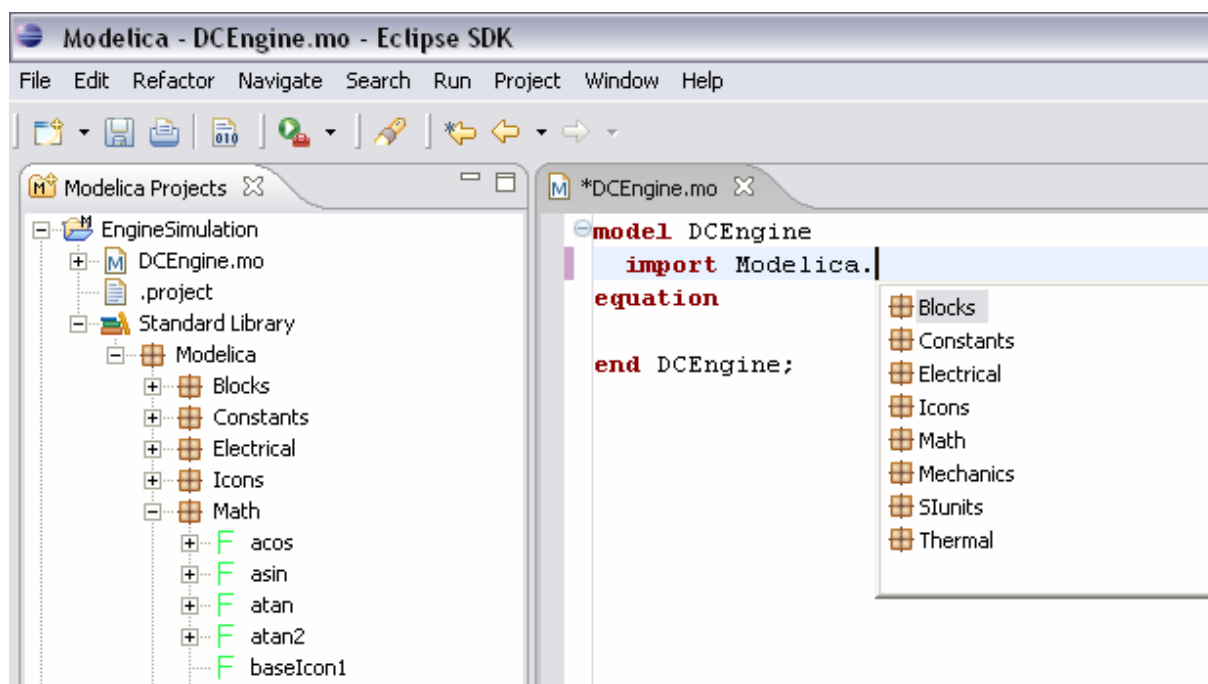


5.3.7 Indentation Support

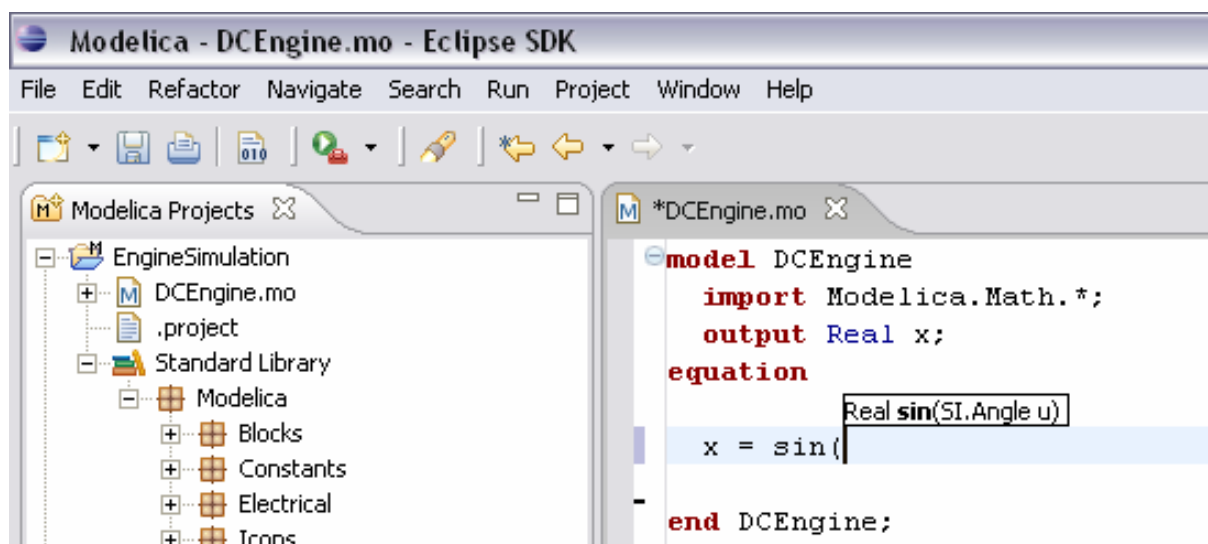
MDT currently has support for automatic indentation. When typing the Return (Enter) key, the next line is indented correctly. You can also correct indentation of the current line or a range selection using CTRL+I or "Correct Indentation" action on the toolbar or in the Edit menu.

5.3.8 Code Completion

MDT supports Code Completion in two variants. The first variant, code completion when typing a dot after a class (package) name, shows alternatives in a menu:



The second variant is useful when typing a call to a function. It shows the function signature (formal parameter names and types) in a popup when typing the parenthesis after the function name, here the signature `Real sin(SI.Angle u)` of the `sin` function:



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.

Note: This Debugger is not yet released for general usage. There is current ongoing work in integrating the Debugger into the MDT/Eclipse plugin for Modelica.

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 present 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 (March 2006) 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.

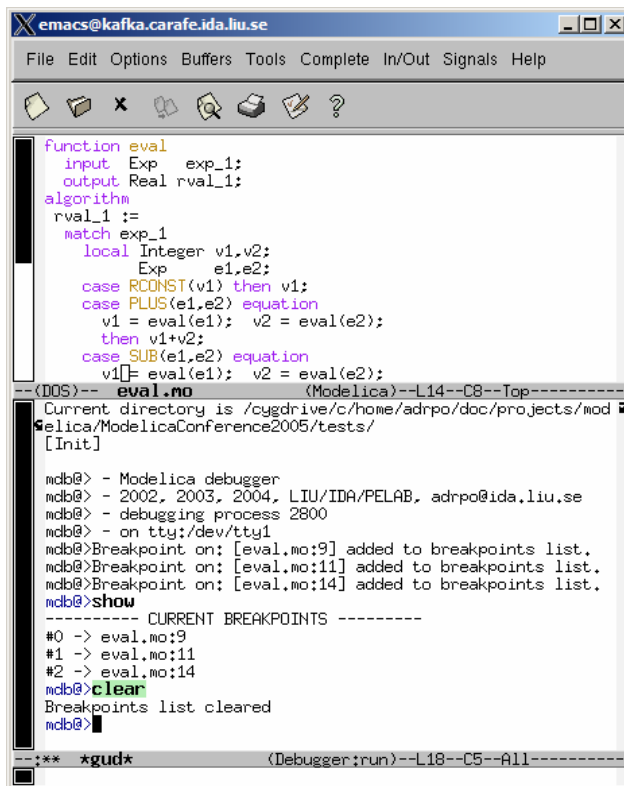


Figure 6-1. Using breakpoints.

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.

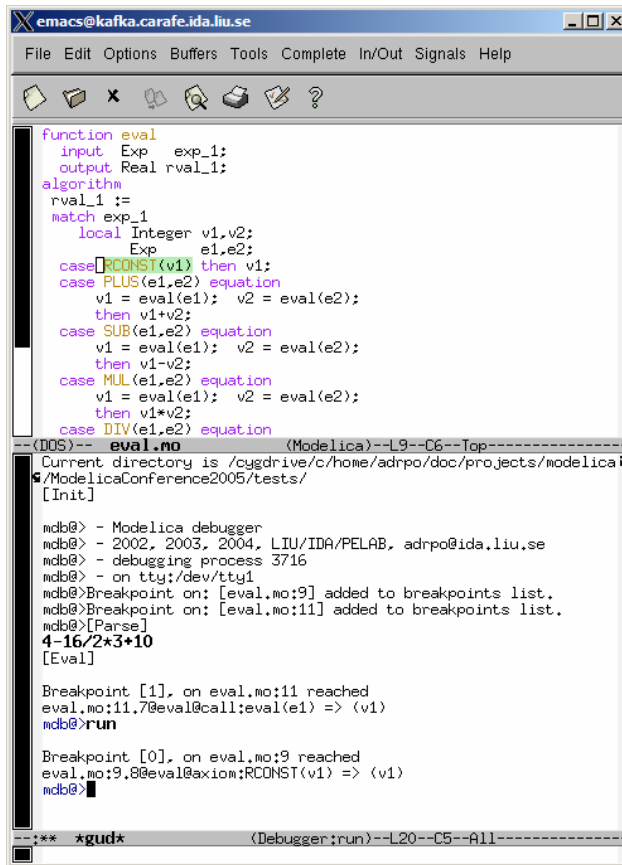


Figure 6-2. Stepping and running.

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.

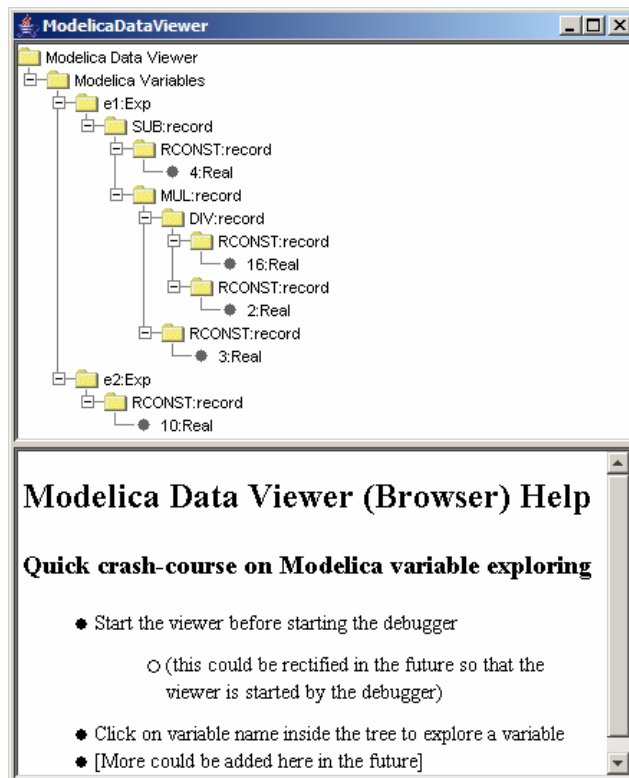


Figure 6-3. Modelica Data Viewer (Browser) for data structures, here a small abstract syntax tree.

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/breakpoint:

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

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

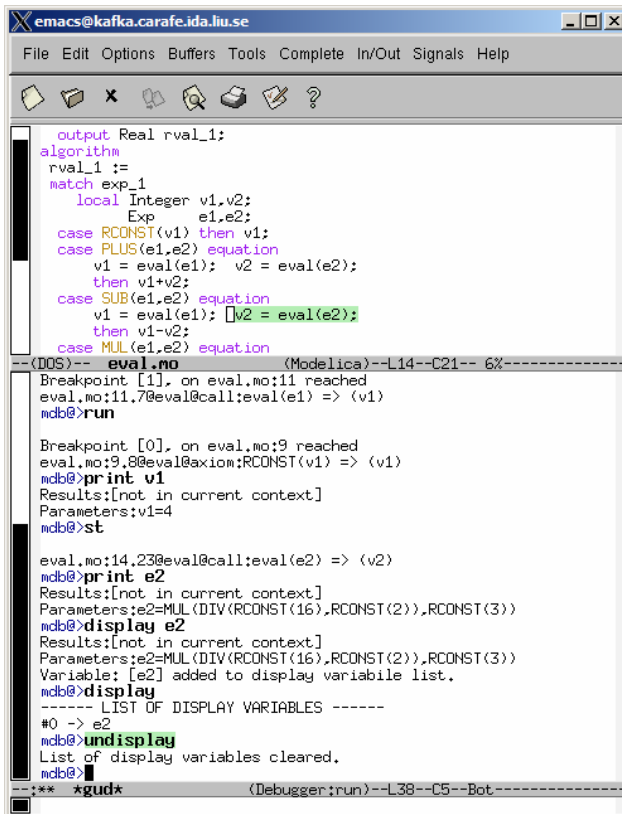


Figure 6-4. Examining variable values using print and display commands.

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.

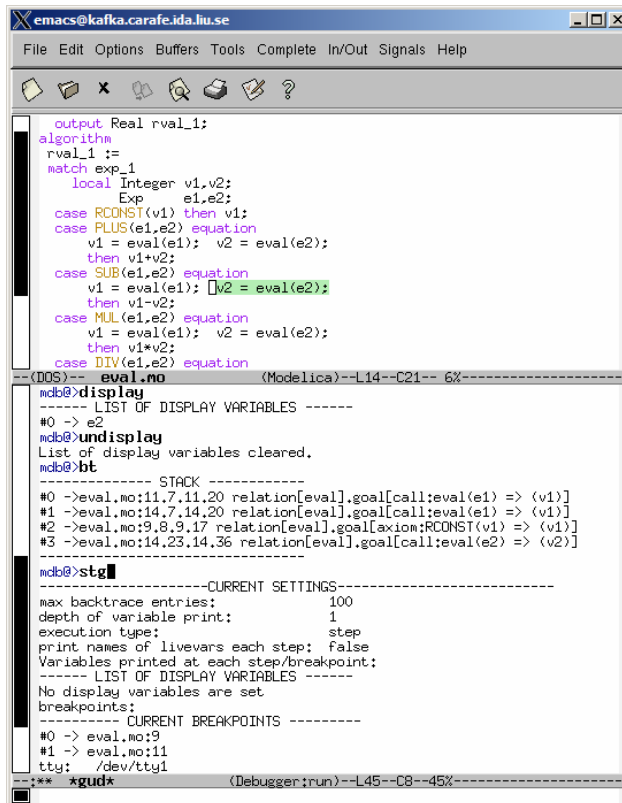
One can invoke the debugging help by issuing:

```
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
v1 = eval(e1); v2 = eval(e2);
then v1/v2;
end match;
end eval;

--(DDB)-- 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
*** xgudx (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 <i>string</i> [on off]	Setting/unsetting breakpoints
cl clear	Clear all breakpoints
sh show	Show all breakpoints
bt backtrace	Print the backtrace (stack)
fbt fbacktrace <i>filter</i>	Print filtered backtrace (stack)
mb maxbacktrace <i>int</i> (0=full, default=0)	Set the maximum of backtrace entries (stack).
ca callchain	Print the call chain
fca fcallchain <i>filter</i>	Print filtered call chain
mc maxcallchain <i>integer</i>	Set the maximum of callchain entries. (0=full, default=100)

[set] de depth <i>integer</i>	Set the depth of variable printing. (0=full, default=10)
[set] ms maxstring <i>integer</i>	Set how many 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 <i>var_name</i>	Print the live variable.
sz size sizeof <i>var_name</i>	Print sizeof the live variable.
di display <i>var_name</i>	Display the live variable each step.
ud undisplay <i>var_name</i>	Un-display the live variable.
di display	Show display variables.
ud undisplay	Un-display ALL display variables.
gr graph <i>var_name</i>	Send the live variable to external viewer.
pty printtype <i>identifier</i>	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.

Appendix A

Contributors to OpenModelica

This Appendix lists the individuals who have made significant contributions to OpenModelica, in the form of software development, design, documentation, project leadership, tutorial material, etc. The individuals are listed for each year, from 1998 to the current year: the project leader and main author/editor of this document followed by main contributors followed by contributors in alphabetical order.

A.1 OpenModelica Contributors 2006

Peter Fritzson, PELAB, Linköping University, Linköping, Sweden.

Peter Aronsson, MathCore Engineering AB, Linköping, Sweden.

Adrian Pop, PELAB, Linköping University, Linköping, Sweden.

Bernhard Bachmann, Fachhochschule Bielefeld, Bielefeld, Germany.

David Broman, PELAB, Linköping University, Linköping, Sweden.

Anders Fernström, PELAB, Linköping University, Linköping, Sweden.

Elmir Jagudin, PELAB, Linköping University, Linköping, Sweden.

Håkan Lundvall, PELAB, Linköping University, Linköping, Sweden.

Kaj Nyström, PELAB, Linköping University, Linköping, Sweden.

Lucian Popescu, MathCore Engineering AB, Linköping, Sweden.

Andreas Remar, PELAB, Linköping University, Linköping, Sweden.

Anders Sandholm, PELAB, Linköping University, Linköping, Sweden.

A.2 OpenModelica Contributors 2005

Peter Fritzson, PELAB, Linköping University, Linköping, Sweden.

Peter Aronsson, PELAB, Linköping University and MathCore Engineering AB, Linköping, Sweden.

Adrian Pop, PELAB, Linköping University, Linköping, Sweden.

Håkan Lundvall, PELAB, Linköping University, Linköping, Sweden.

Ingemar Axelsson, PELAB, Linköping University, Linköping, Sweden.

David Broman, PELAB, Linköping University, Linköping, Sweden.

Daniel Hedberg, MathCore Engineering AB, Linköping, Sweden.

Håkan Lundvall, PELAB, Linköping University, Linköping, Sweden.

Kaj Nyström, PELAB, Linköping University, Linköping, Sweden.

Lucian Popescu, MathCore Engineering AB, Linköping, Sweden.

Levon Saldamli, PELAB, Linköping University, Linköping, Sweden.

A.3 OpenModelica Contributors 2004

Peter Fritzson, PELAB, Linköping University, Linköping, Sweden.

Peter Aronsson, Linköping University, Linköping, Sweden.

Bernhard Bachmann, Fachhochschule Bielefeld, Bielefeld, Germany.

Peter Bunus, PELAB, Linköping University, Linköping, Sweden.

Daniel Hedberg, MathCore Engineering AB, Linköping, Sweden.

Håkan Lundvall, PELAB, Linköping University, Linköping, Sweden.

Emma Larsdotter Nilsson, PELAB, Linköping University, Linköping, Sweden.

Kaj Nyström, PELAB, Linköping University, Linköping, Sweden.
Adrian Pop, PELAB, Linköping University, Linköping, Sweden.
Lucian Popescu, MathCore Engineering AB, Linköping, Sweden.
Levon Saldamli, PELAB, Linköping University, Linköping, Sweden.

A.4 OpenModelica Contributors 2003

Peter Fritzson, PELAB, Linköping University, Linköping, Sweden.
Peter Aronsson, Linköping University, Linköping, Sweden.
Levon Saldamli, PELAB, Linköping University, Linköping, Sweden.
Peter Bunus, PELAB, Linköping University, Linköping, Sweden.
Vadim Engelson, PELAB, Linköping University, Linköping, Sweden.
Daniel Hedberg, Linköping University, Linköping, Sweden.
Eva-Lena Lengquist-Sandelin, PELAB, Linköping University, Linköping, Sweden.
Susanna Monemar, PELAB, Linköping University, Linköping, Sweden.
Adrian Pop, PELAB, Linköping University, Linköping, Sweden.
Erik Svensson, MathCore Engineering AB, Linköping, Sweden.

A.5 OpenModelica Contributors 2002

Peter Fritzson, PELAB, Linköping University, Linköping, Sweden.
Levon Saldamli, PELAB, Linköping University, Linköping, Sweden.
Peter Aronsson, Linköping University, Linköping, Sweden.
Daniel Hedberg, Linköping University, Linköping, Sweden.
Henrik Johansson, PELAB, Linköping University, Linköping, Sweden
Andreas Karström, PELAB, Linköping University, Linköping, Sweden

A.6 OpenModelica Contributors 2001

Peter Fritzson, PELAB, Linköping University, Linköping, Sweden.
Levon Saldamli, PELAB, Linköping University, Linköping, Sweden.
Peter Aronsson, Linköping University, Linköping, Sweden.

A.7 OpenModelica Contributors 2000

Peter Fritzson, PELAB, Linköping University, Linköping, Sweden.

A.8 OpenModelica Contributors 1999

Peter Fritzson, PELAB, Linköping University, Linköping, Sweden

—

Peter Rönnquist, PELAB, Linköping University, Linköping, Sweden.

A.9 OpenModelica Contributors 1998

Peter Fritzson, PELAB, Linköping University, Linköping, Sweden.
David Kågedal, PELAB, Linköping University, Linköping, Sweden.
Vadim Engelson, PELAB, Linköping University, Linköping, Sweden.

Index

literate programming 31

