OpenModelica Users Guide

Version 2011-11-21
for OpenModelica 1.8

November 2011

Peter Fritzson
Adrian Pop, Martin Sjölund, Per Östlund, Peter Aronsson,
Adeel Asghar, Mikael Axin, Bernhard Bachmann, Vasile Baluta, Robert Braun, Willi Braun, David Broman, Stefan Brus, Francesco Casella, Filippo Donida, Jens Frenkel, Mahder Gebremedhin, Pavel Grozman, Daniel Hedberg, Michael Hanke, Alf Isaksson, Kim Jansson, Daniel Kanth, Tommi Karhela, Juha Kortelainen, Abhinn Kothari, Petter Krus, Alexey Lebedev, Oliver Lenord, Ariel Liebman, Rickard Lindberg, Håkan Lundvall, Abhi Raj Metkar, Eric Meyers, Tuomas Miettinen, Afshin Moghadam, Maroun Nemer, Hannu Niemistö, Peter Nordin, Kristoffer Norling, Lennart Ochel, Karl Pettersson, Pavol Privitzer, Reino Ruusu, Per Sahlin,Wladimir Schamai, Gerhard Schmitz, Anton Sodja, Ingo Staack, Kristian Stavåker, Sonia Tariq, Mohsen Torabzadeh-Tari, Parham Vasaiely, Niklas Worschech, Robert Wotzlaw, Björn Zackrisson,
Azam Zia

This document is part of OpenModelica: http://www.openmodelica.org
Contact: OpenModelica@ida.liu.se

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 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.

1.1 System Overview

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

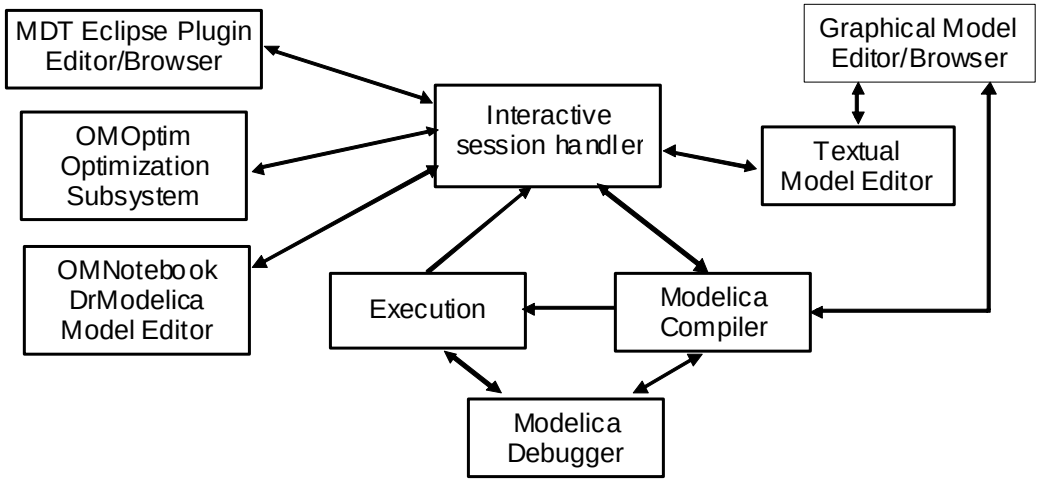


Figure 1-1-1. The architecture of the OpenModelica environment. Arrows denote data and control flow. The interactive session handler receives commands and shows results from evaluating commands and expressions that are translated and executed. Several subsystems provide different forms of browsing and textual editing of Modelica code. The debugger currently provides debugging of an extended algorithmic subset of Modelica

The 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.

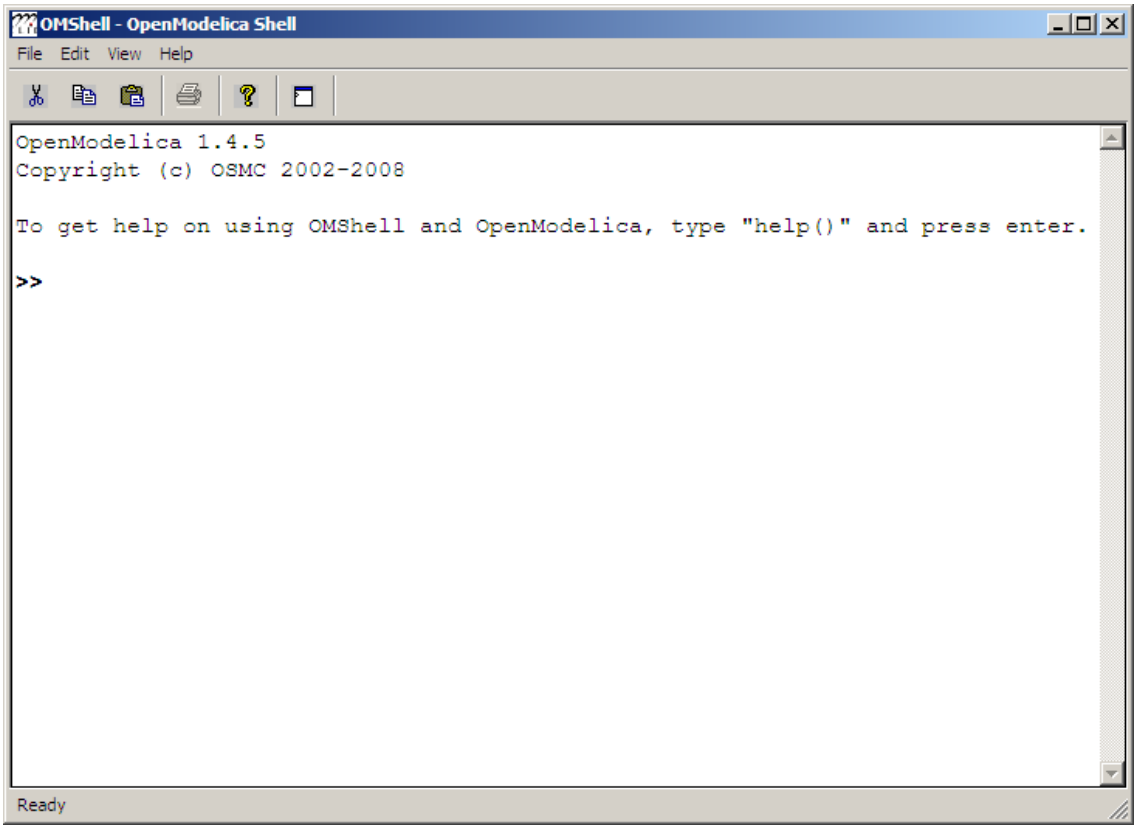
- *Eclipse plugin editor/browser*. The Eclipse plugin called MDT (Modelica Development Tooling) provides file and class hierarchy browsing and text editing capabilities, rather analogous to previously described Emacs editor/browser. Some syntax highlighting facilities are also included. The Eclipse framework 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 facilities are yet available in the cells of this notebook editor.
- *Graphical model editor/browser OMEdit.*. 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 also includes a textual editor for editing model class definitions, and a window for interactive Modelica command evaluation.
- *Optimization subsystem OMEdit.*. This is an optimization subsystem for OpenModelica, currently for design optimization choosing an optimal set of design parameters for a model. The current version has a graphical user interface, provides genetic optimization algorithms and Pareto front optimizaiton, works integrated with the simulators and automatically accesses variables and design parameters from the Modelica model.
- *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 Eclipse 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.

1.2 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` (C:/OpenModelica1.8.0/share/doc/omc/testmodels/) directory, see also Chapter 4.

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 Using Compiler Debug Trace Flags in Interactive Mode

When running OMC in interactive mode (for instance using OMShell) one can make use of some of the compiler debug trace flags defined in section 2.1.2 in the System Documentation. Here we give a few example sessions.

Example Session 1

```
OpenModelica 1.8.0
Copyright (c) OSMC 2002-2011
To get help on using OMShell and OpenModelica, type "help()" and press enter.

>> setDebugFlags("failtrace")
true

>> model A Integer t = 1.5; end A;  //The type is Integer but 1.5 is of Real Type
{A}

>> instantiateModel(A)
"/*- CevalScript.cevalGenerateFunctionDAEs failed( instantiateModel )*/
```

```
/*- CevalScript.cevalGenerateFunction failed(instantiateModel)*/
- Inst.makeBinding failed
- Inst.instElement failed: COMPONENT(t in/out: mod: = 1.5 tp: Integer var :VAR,
baseClass: <nothing>)
Scope: A
- Inst.instClassdef failed
class :A
- Inst.instClass: A failed
Inst.instClassInProgram failed
"
Error: Type mismatch in modifier, expected type Integer, got modifier =1.5 of type Real
Error: Error occured while flattening model A
Error: Type mismatch in modifier, expected type Integer, got modifier =1.5 of type Real
Error: Error occured while flattening model A
```

Example Session 2

```
OpenModelica 1.8.0
Copyright (c) OSMC 2002-2011
To get help on using OMShell and OpenModelica, type "help()" and press enter.

>> setDebugFlags("dump")
true
---DEBUG(dump)---
IEXP(Absyn.CALL(Absyn.CREF_IDENT("setDebugFlags", []),
FUNCTIONARGS(Absyn.STRING("dump"), )))
---/DEBUG(dump)---
"
---DEBUG(dump)---
IEXP(Absyn.CALL(Absyn.CREF_IDENT("getErrorString", []), FUNCTIONARGS(, )))
---/DEBUG(dump)—

>> model B Integer k = 10; end B;
{B}
---DEBUG(dump)---
Absyn.PROGRAM([
Absyn.CLASS("B", false, false, false, Absyn.R_MODEL,
Absyn.PARTS([Absyn.PUBLIC([Absyn.ELEMENTITEM(Absyn.ELEMENT(false, _, Absyn.UNSPECIFIED
, "component", Absyn.COMPONENTS(Absyn.ATTR(false, false, Absyn.VAR, Absyn.BIDIR,
[]),Integer,[Absyn.COMPONENTITEM(Absyn.COMPONENT("k",[], SOME(Absyn.CLASSMOD([],
SOME(Absyn.INTEGER(10))))), NONE())]), Absyn.INFO("", false, 1, 9, 1, 23)), NONE))])],
NONE()), Absyn.INFO("", false, 1, 1, 1, 30))
],Absyn.TOP)
---/DEBUG(dump)---
"
---DEBUG(dump)---
IEXP(Absyn.CALL(Absyn.CREF_IDENT("getErrorString", []), FUNCTIONARGS(, )))
---/DEBUG(dump)—

>> instantiateModel(B)
"fclass B
Integer k = 10;
end B;
"
---DEBUG(dump)---
IEXP(Absyn.CALL(Absyn.CREF_IDENT("instantiateModel", []),
FUNCTIONARGS(Absyn.CREF(Absyn.CREF_IDENT("B", [])), )))
---/DEBUG(dump)---
"
```

```
---DEBUG(dump)---
IEXP(Absyn.CALL(Absyn.CREF_IDENT("getErrorString", []), FUNCTIONARGS(, )))
---/DEBUG(dump)—

>> simulate(B, startTime=0, stopTime=1, numberOfIntervals=500, tolerance=1e-4)
record SimulationResult
resultFile = "B_res.plt"
end SimulationResult;
---DEBUG(dump)---
#ifdef __cplusplus
extern "C" {
#endif
#ifdef __cplusplus
}
#endif
IEXP(Absyn.CALL(Absyn.CREF_IDENT("simulate", []),
FUNCTIONARGS(Absyn.CREF(Absyn.CREF_IDENT("B", [])), startTime = Absyn.INTEGER(0),
stopTime = Absyn.INTEGER(1), numberOfIntervals = Absyn.INTEGER(500), tolerance =
Absyn.REAL(0.0001))))
---/DEBUG(dump)---
"
---DEBUG(dump)---
IEXP(Absyn.CALL(Absyn.CREF_IDENT("getErrorString", []), FUNCTIONARGS(, )))
---/DEBUG(dump)--
```

Example Session 3

```
OpenModelica 1.8.0
Copyright (c) OSMC 2002-2011
To get help on using OMShell and OpenModelica, type "help()" and press enter.

>> setDebugFlags("failtrace")
true

>> model C Integer a; Real b; equation der(a) = b; der(b) = 12.0; end C;
{C}

>> instantiateModel(C)
"/*- CevalScript.cevalGenerateFunctionDAEs failed( instantiateModel )*/
/*- CevalScript.cevalGenerateFunction failed(instantiateModel)*/
- Static.elabCall failed
function: der posargs: a
- Static.elabExp failed: der(a)
Scope: C
- instEquationCommon failed for eqn: der(a) = b; in scope:C
- instEquation failed eqn:der(a) = b;
- Inst.instClassdef failed
class :C
- Inst.instClass: C failed
Inst.instClassInProgram failed
"
Error: Illegal derivative. der(a) where a is of type Integer, which is not a subtype of
Real
Error: Wrong type or wrong number of arguments to der(a)'.
Error: Error occured while flattening model C
Error: Illegal derivative. der(a) where a is of type Integer, which is not a subtype of
Real
Error: Wrong type or wrong number of arguments to der(a)'.
```

```
Error: Error occured while flattening model C
```

1.2.3 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:/OpenModelica1.8.0/share/doc/omc/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:/OpenModelica1.8.0/share/doc/omc/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;
```

1.2.4 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:/OpenModelica1.8.0/share/doc/omc/testmodels/"

>> cd("..")
" C:/OpenModelica1.8.0/share/doc/omc/"

>> cd("C:/OpenModelica1.8.0/share/doc/omc/testmodels/")
" C:/OpenModelica1.8.0/share/doc/omc/testmodels/"
```

1.2.5 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:/OpenModelica1.8.0/share/doc/omc/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:

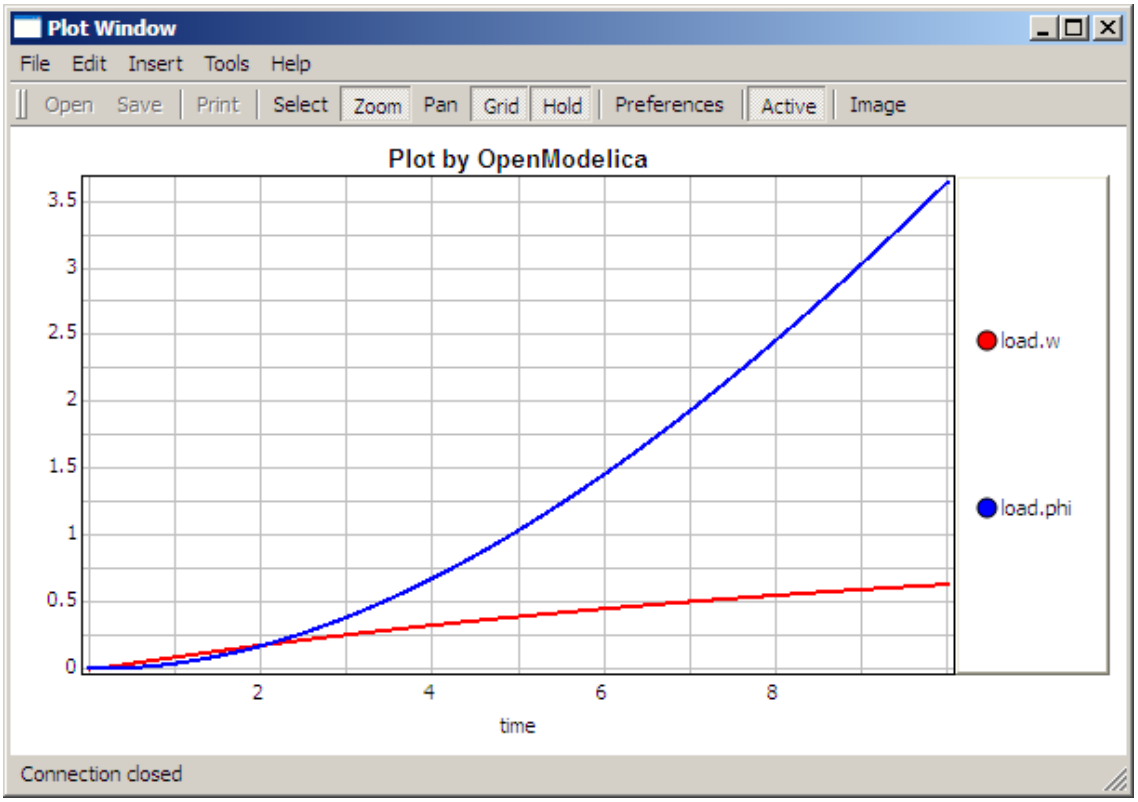
```
>> instantiateModel(dcmotor)
```

```
"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
```



1.2.6 The val() function

The `val`(*variableName*,*time*) scription function can be used to retrieve the interpolated value of a simulation result variable at a certain point in the simulation time, see usage in the BouncingBall simulation below.

1.2.7 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):

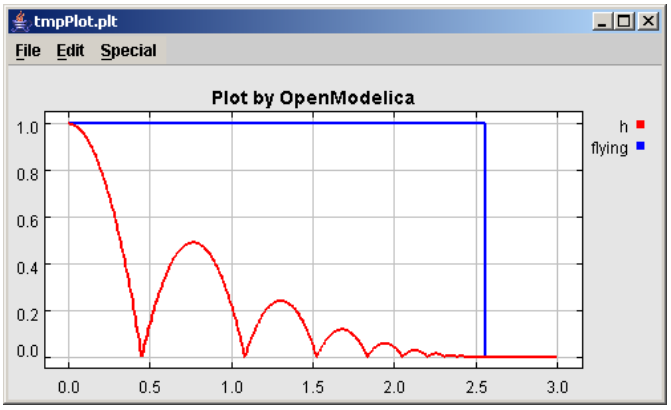
```
>> loadFile("C:/OpenModelica1.8.0/share/doc/omc/testmodels/BouncingBall.mo")
true
>> list(BouncingBall)
"model BouncingBall
  parameter Real e=0.7 "coefficient of restitution";
  parameter Real g=9.81 "gravity acceleration";
  Real h(start=1) "height of ball";
  Real v "velocity of ball";
  Boolean 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):

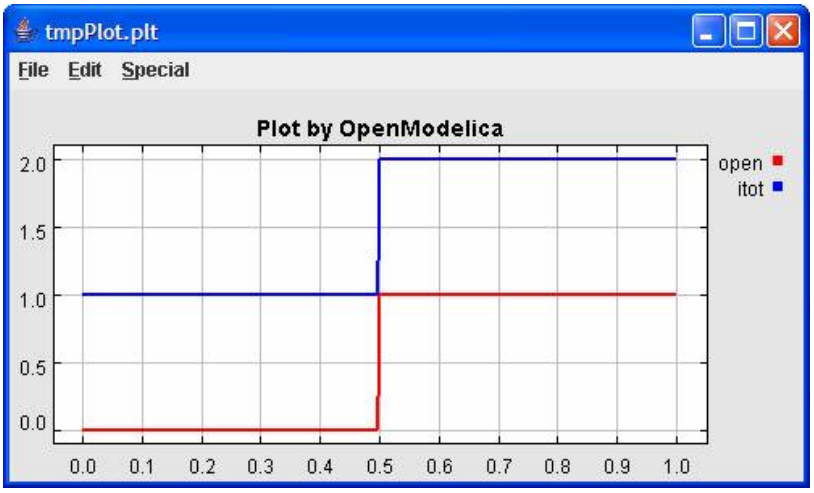
```
>> model Switch
  Real v;
  Real i;
  Real i1;
  Real itot;
  Boolean open;
equation
  itot = i + i1;
  if open then
    v = 0;
  else
    i = 0;
  end if;
   1 - i1 = 0;
   1 - v - i = 0;
   open = time >= 0.5;
end Switch;
Ok
>> simulate(Switch, startTime=0, stopTime=1);
```

Retrieve the value of `itot` at time=0 using the `val`(*variableName*,*time*) function:

```
>> val(itot,0)
1
```

Plot `itot` and `open`:

```
>> 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.

1.2.8 Clear All Models

Now, first clear all loaded libraries and models:

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

List the loaded models – nothing left:

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

1.2.9 VanDerPol Model and Parametric Plot

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

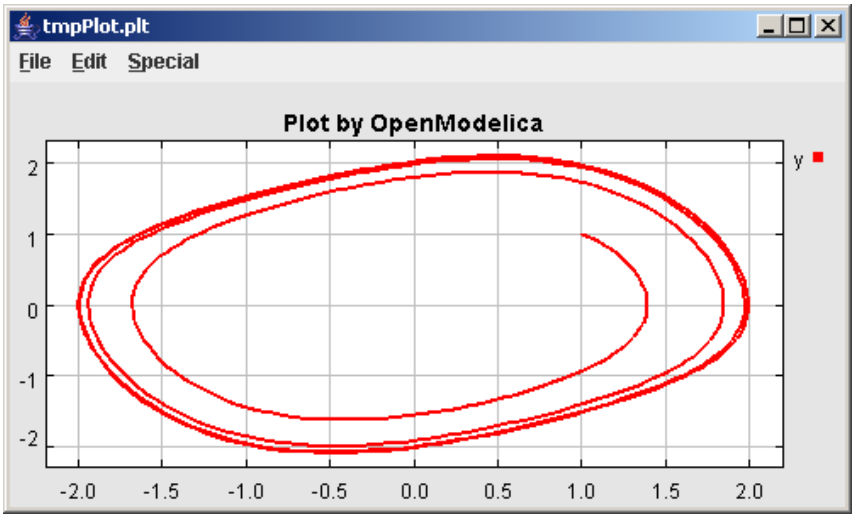
```
>> loadFile("C:/OpenModelica1.8.0/share/doc/omc/testmodels/VanDerPol.mo"))
true
```

It is simulated:

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

It is plotted:

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



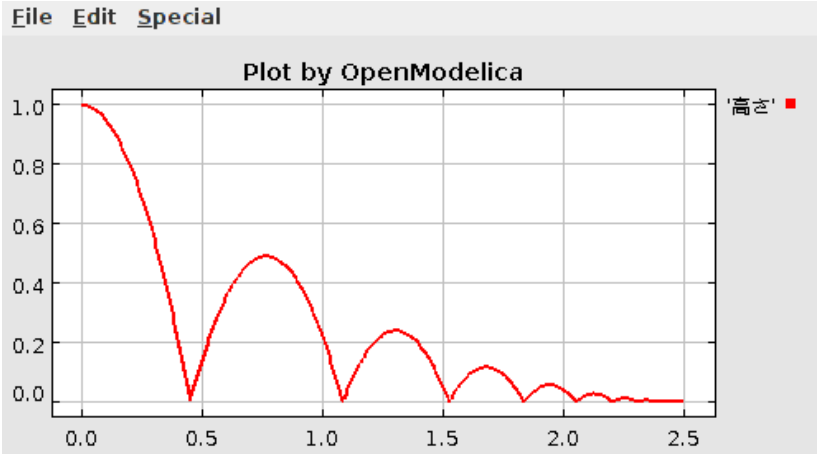
Perform code instantiation to flat forrm of the `VanDerPol` model:

```
>> 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;
"
```

1.2.10 Using Japanese or Chinese Characters

Japenese, Chinese, and other kinds of UniCode characters can be used within quoted (single quote) identifiers, see for example the variable name to the right in the plot below:



1.2.11 Scripting with For-Loops, While-Loops, and If-Statements

A simple summing integer loop (using multi-line input without evaluation at each line into OMShell requires copy-paste as one operation from another document):

```
>> k := 0;
   for i in 1:1000 loop
     k := k + i;
   end for;

>> k
500500
```

A nested loop summing reals and integers::

```
>> g := 0.0;
   h := 5;
   for i in {23.0,77.12,88.23} loop
     for j in i:0.5:(i+1) loop
       g := g + j;
       g := g + h / 2;
     end for;
     h := h + g;
   end for;
```

By putting two (or more) variables or assignment statements separated by semicolon(s), ending with a variable, one can observe more than one variable value:

```
>> h;g
1997.45
1479.09
```

A for-loop with vector traversal and concatenation of string elements:

```
>> i:="";
   lst := {"Here ", "are ","some ","strings."};
   s := "";
   for i in lst loop
     s := s + i;
   end for;

>> s
"Here are some strings."
```

Normal while-loop with concatenation of 10 "abc " strings:

```
>> s:="";
   i:=1;
   while i<=10 loop
     s:="abc "+s;
     i:=i+1;
   end while;

>> s
"abc abc abc abc abc abc abc abc abc abc "
```

A simple if-statement. By putting the variable last, after the semicolon, its value is returned after evaluation:

```
>> if 5>2 then a := 77; end if; a
77
```

An if-then-else statement with elseif:

```
>> if false then
```

```
    a := 5;
  elseif a > 50 then
    b:= "test"; a:= 100;
  else
    a:=34;
  end if;
```

Take a look at the variables a and b:

```
>> a;b

100
"test"
```

1.2.12 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
```

1.2.13 Getting Information about Error Cause

Call the function getErrorString in order to get more information about the error cause after a simulation failure:

```
>> getErrorString()
```

1.2.14 Alternative Simulation Output Formats

There are several output format possibilities, with `mat` being the default. `plt` and `mat` are the only formats that allow you to use the `val()` or `plot()` functions after a simulation. Compared to the speed of `plt`, `mat` is roughly 5 times for small files, and scales better for larger files due to being a binary format. The `csv` format is roughly twice as fast as `plt` on data-heavy simulations. The `plt` format allocates all output data in RAM during simulation, which means that simulations may fail due applications only being able to address 4GB of memory on 32-bit platforms. Empty does no output at all and should be by far the fastest. The `csv` and `plt` formats are suitable when using an external scripts or tools like `gnuplot` to generate plots or process data. The `mat` format can be post-processed in MATLAB[1] or Octave[2].

```
simulate(... , outputFormat="mat")
simulate(... , outputFormat="csv")
simulate(... , outputFormat="plt")
simulate(... , outputFormat="empty")
```

It is also possible to specify which variables should be present in the result-file. This is done by using POSIX Extended Regular Expressions[3]. The given expression must match the full variable name (`^` and `$` symbols are automatically added to the given regular expression).

```
// Default, match everything
simulate(... , variableFilter=".*")
// match indices of variable myVar that only contain the numbers using combinations
// of the letters 1 through 3
simulate(... , variableFilter="myVar\\[[1-3]*\\]")
// match x or y z
simulate(... , variableFilter="x|y|z")
```

1.2.15 Using External Functions

See Chapter 11 for more information about calling functions in other programming languages.

1.2.16 Using Parallel Simulation via OpenMP Multi-Core Support

Faster simulations on multi-core computers can be obtained by using a new OpenModelica feature that automatically partitions the system of equations and schedules the parts for execution on different cores using

[1] http://www.mathworks.com/products/matlab/
[2] http://www.gnu.org/software/octave/
[3] http://en.wikipedia.org/wiki/Regular_expression

shared-memory OpenMP based execution. The speedup obtained is dependent on the model structure, whether the system of equations can be partitioned well. This version in the OpenModelica 1.8 release is an experimental version without load balancing. The following command, not yet available from the OpenModelica GUI, will run a parallel simulation on a model:

```
omc +d=openmp model.mo
```

1.2.17 Loading Specific Library Version

There exists many different version of Modelica libraries which are not compatible. It is possible to keep multiple versions of the same library stored in the directory given by calling `getModelicaPath()`. By calling `loadModel(Modelica,{"3.2"})`, OpenModelica will search for a directory called "Modelica 3.2" or a file called "Modelica 3.2.mo". It is possible to give several library versions to search for, giving preference for a pre-release version of a library if it is installed. If the searched version is "default", the priority is: no version name (Modelica), main release version (Modelica 3.1), pre-release version (Modelica 3.1Beta 1) and unordered versions (Modelica Special Release).

The `loadModel` command will also look at the uses annotation of the top-level class after it has been loaded. Given the following package, Complex 1.0 and ModelicaServices 1.1 will also be loaded into the AST automatically.

```
package Modelica
  annotation(uses(Complex(version="1.0"), ModelicaServices(version="1.1")))
end Modelica;
```

1.2.18 Calling the Model Query and Manipulation API

In the OpenModelica System Documentation, an external API (application programming interface) is described which returns information about models and/or allows manipulation of models. Calls to these functions can be done interactively as below, but more typically by program clients to the OpenModelica Compiler (OMC) server. Current examples or such clients are the OpenModelica MDT Eclipse plugin, OMNotebook, the OMEdit graphic model editor, etc. This API is untyped for performance reasons, i.e., no type checking and minimal error checking is done on the calls. The results of a call is returned as a text string in Modelica syntax form, which the client has to parse. An example parser in C++ is available in the OMNotebook source code, whereas another example parser in Java is available in the MDT Eclipse plugin.

Below we show a few calls on the previously simulated BouncingBall model. The full documentation on this API is available in the system documentation. First we load and list the model again to show its structure:

```
>>loadFile("C:/OpenModelica1.8.0/share/doc/omc/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;
"
```

Different kinds of calls with returned results:

```
>>getClassRestriction(BouncingBall)
"model"
```

```
>>getClassInformation(BouncingBall)
{"model","","",{false,false,false},{"writable",1,1,18,17}}
```

```
>>isFunction(BouncingBall)
false
```

```
>>existClass(BouncingBall)
true
```

```
>>getComponents(BouncingBall)
{{Real,e,"coefficient of restitution", "public", false, false, false,
"parameter", "none", "unspecified"},
{Real,g,"gravity acceleration",
"public", false, false, false, "parameter", "none", "unspecified"},
{Real,h,"height of ball", "public", false, false, false,
"unspecified", "none", "unspecified"},
{Real,v,"velocity of ball",
"public", false, false, false, "unspecified", "none", "unspecified"},
{Boolean,flying,"true, if ball is flying", "public", false, false,
false, "unspecified", "none", "unspecified"},
{Boolean,impact,"",
"public", false, false, false, "unspecified", "none", "unspecified"},
{Real,v_new,"", "public", false, false, false, "unspecified", "none",
"unspecified"}}
```

```
>>getConnectionCount(BouncingBall)
0
```

```
>>getInheritanceCount(BouncingBall)
0
```

```
>>getComponentModifierValue(BouncingBall,e)
0.7
```

```
>>getComponentModifierNames(BouncingBall,e)
{}
```

```
>>getClassRestriction(BouncingBall)
"model"
```

```
>>getVersion() // Version of the currently running OMC
"1.6"
```

1.2.19 Quit OpenModelica

Leave and quit OpenModelica:

```
>> quit()
```

1.2.20 Dump XML Representation

The command `dumpXMLDAE` dumps an XML representation of a model, according to several optional parameters.

```
dumpXMLDAE(modelname[,asInSimulationCode=<Boolean>] [,filePrefix=<String>]
[,storeInTemp=<Boolean>] [,addMathMLCode =<Boolean>])
```

This command dumps the mathematical representation of a model using an XML representation, with optional parameters. In particular, `asInSimulationCode` defines where to stop in the translation process (before dumping the model), the other options are relative to the file storage: `filePrefix` for specifying a different name and `storeInTemp` to use the temporary directory. The optional parameter `addMathMLCode` gives the possibility to don't print the MathML code within the xml file, to make it more readable.Usage is trivial, just: `addMathMLCode`=true/false (default value is false).

1.2.21 Dump Matlab Representation

The command export dumps an XML representation of a model, according to several optional parameters.

```
exportDAEtoMatlab(modelname);
```

This command dumps the mathematical representation of a model using a Matlab representation. Example:

```
$ cat daequery.mos
loadFile("BouncingBall.mo");
exportDAEtoMatlab(BouncingBall);
readFile("BouncingBall_imatrix.m");

$ omc daequery.mos
true
"The equation system was dumped to Matlab file:BouncingBall_imatrix.m"
"
% Incidence Matrix
% ===================================
% number of rows: 6
IM={[3,-6],[1,{'if', 'true','==' {3},{},}],[2,{'if', 'edge(impact)'
{3},{5},}],[4,2],[5,{'if', 'true','==' {4},{},}],[6,-5]};
VL = {'foo','v_new','impact','flying','v','h'};


EqStr = {'impact = h <= 0.0;','foo = if impact then 1 else 2;','when {h <= 0.0 AND v
<= 0.0,impact} then v_new = if edge(impact) then (-e) * pre(v) else 0.0; end
when;','when {h <= 0.0 AND v <= 0.0,impact} then flying = v_new > 0.0; end
when;','der(v) = if flying then -g else 0.0;','der(h) = v;'};

OldEqStr={'fclass BouncingBall','parameter Real e = 0.7 "coefficient of
restitution";','parameter Real g = 9.81 "gravity acceleration";','Real h(start = 1.0)
"height of ball";','Real v "velocity of ball";','Boolean flying(start = true) "true,
if ball is flying";','Boolean impact;','Real v_new;','Integer foo;','equation','
impact = h <= 0.0;','  foo = if impact then 1 else 2;','  der(v) = if flying then -g
else 0.0;','  der(h) = v;','  when {h <= 0.0 AND v <= 0.0,impact} then','  v_new = if
edge(impact) then (-e) * pre(v) else 0.0;','  flying = v_new > 0.0;','
reinit(v,v_new);','  end when;','end BouncingBall;',''};"
```

1.3 Summary of 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*`>][,outputInterval=<`*Real*`>][,method=<`*String*`>] [,tolerance=<`*Real*`>][,fixedStepSize=<`*Real*`>] [,outputFormat=<`*String*`>])`	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 intervales will give higher time resolution, but occupy more space and take longer to compute. The default number of intervals is 500. It is possible to choose solving method, default is "dassl", "euler" and "rungekutta" are also available. Output format "plt" is default and the only one that works with th val() command, "csv" (comma separated values) and "empty" (no output) are also available.
`plot(`*vars*`)`	Plot the variables given as a vector or a scalar, e.g. `plot({x1,x2})` or `plot(x1)`.
`plotParametric(`*var1*, *var2*`)`	Plot var2 relative to var1 from the most recently simulated model, e.g. `plotParametric(x,y)`.
`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.
`dumpXMLDAE(`*modelname*, ...`)`	Dumps an XML representation of a model, according to several optional parameters.
`exportDAEtoMatlab(`*name*`)`	Dumps an Matlab representation of a model.
`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 the path indicated by the environment variable `OPENMODELICALIBRARY`.
`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*.
`val(`*variable*`,`*timePoint*`)`	Return the value of the *variable* at time *timePoint*.
`help()`	Print this helptext (returned as a string).
`quit()`	Leave and quit the OpenModelica environment

1.4 References

Peter Fritzson, Peter Aronsson, Håkan Lundvall, Kaj Nyström, Adrian Pop, Levon Saldamli, and David Broman. The OpenModelica Modeling, Simulation, and Software Development Environment. In Simulation News Europe, 44/45, December 2005. See also: http://www.openmodelica.org.

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

The Modelica Association. The Modelica Language Specification Version 3.0, Sept 2007. http://www.modelica.org.

Chapter 2

OMEdit – The OpenModelica Connection Editor

OMEdit – the OpenModelica Connection Editor is the new Graphical User Interface for graphical model editing in OpenModelica. It is implemented in C++ using the Qt 4.7 graphical user interface library and supports the Modelica Standard Library version 3.1 that is included in the latest OpenModelica installation. This chapter gives a brief introduction to OMEdit and also demonstrates how to create a `DCmotor` model using the editor.

OMEdit provides several user friendly features for creating, browsing, editing, and simulating models:

- *Modeling* – Easy model creation for Modelica models.
- *Pre-defined models* – Browsing the Modelica Standard library to access the provided models.
- *User defined models* – Users can create their own models for immediate usage and later reuse.
- *Component interfaces* – Smart connection editing for drawing and editing connections between model interfaces.
- *Simulation* – Subsystem for running simulations and specifying simulation parameters start and stop time, etc.
- *Plotting* – Interface to plot variables from simulated models.

2.1 Starting OMEdit

2.1.1 Microsoft Windows

OMEdit can be launched using the executable placed in `OpenModelicaInstallationDirectory/bin/OMEdit/OMEdit.exe`. Alternately, choose `OpenModelica > OpenModelica Connection Editor` from the start menu in Windows. A splash screen similar to the one shown in Figure 2-1 will appear indicating that it is starting OMEdit. After the splash screen the main OMEdit window will appear; see Figure 2-2.

Figure 2-1: OMEdit Splash Screen.

2.1.2 Linux

?? fill in

2.1.3 Mac OS X

?? fill in

2.2 Introductory Modeling in OMEdit

In this section we will demonstrate how one can create Modelica models in OMEdit, e.g. a `DCmotor`.

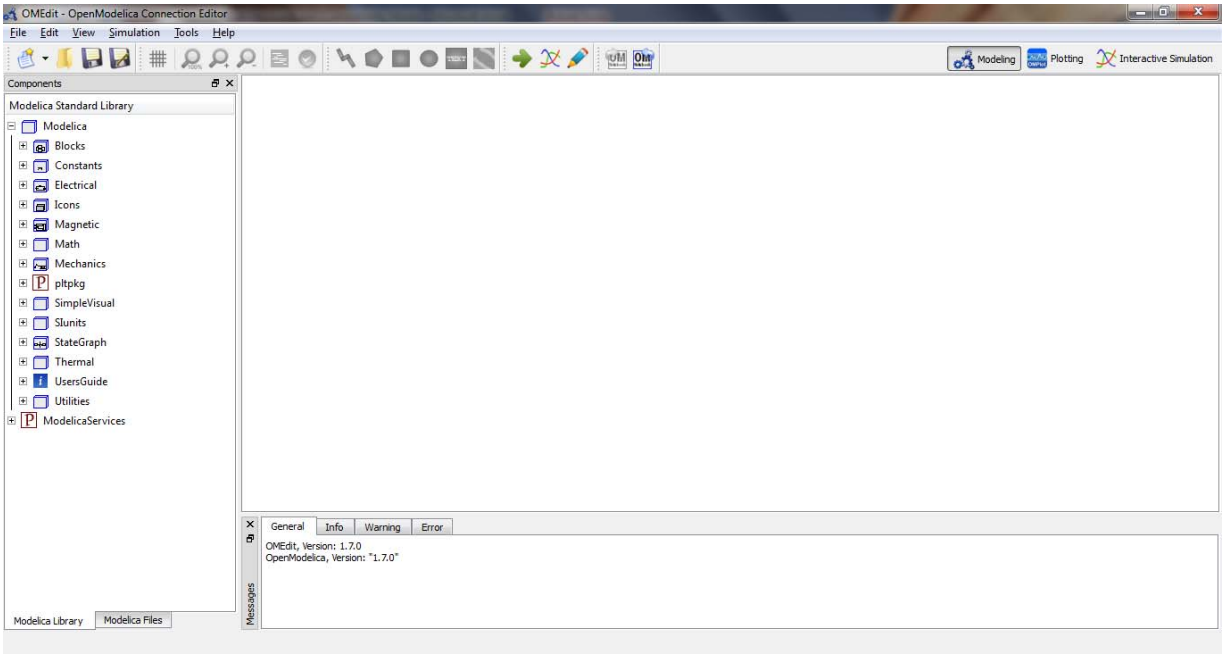


Figure 2-2: OMEdit Main Window.

2.2.1 Creating a New File

Creating a new file/model in OMEdit is rather straightforward. In OMEdit the new file can be of type `model`, `class`, `connector`, `record`, `block`, `function` and `package`. User can create any of the file types mentioned above by selecting `File > New` from the menu. Alternatively, you can also click on the drop down button beside `new icon` shown in toolbar right below the File menu. See Figure 2-4.

For this introductory example we will create a new model named `DCmotor`. By default the newly created model will open up in the tabbed view of OMEdit, also called Section 2.5.2), and become visible. The models are created in the OMC global scope unless you specify the parent package for it.

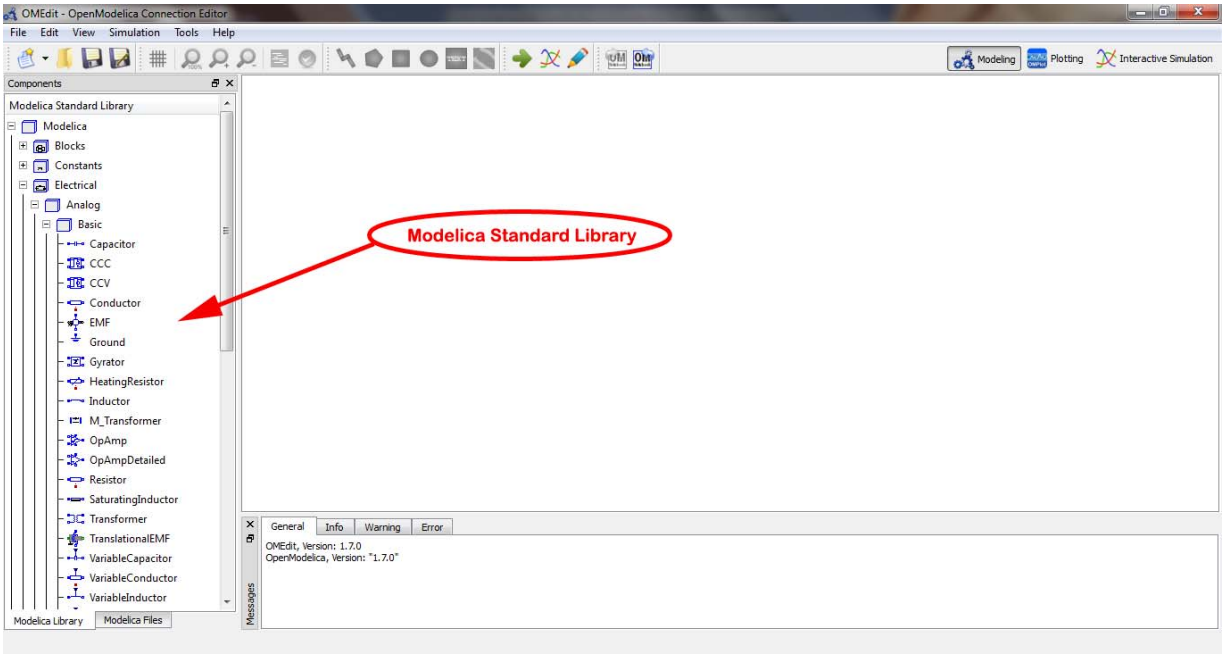


Figure 2-3: Modelica Standard Library.

2.2.2 Adding Component Models

The Modelica standard library is loaded automatically and is available in the left dock window. The library is retrieved through the `loadModel(Modelica)` API call and is loaded in the OMC symbol table and workspace after the command execution is completed. Component models available in the Modelica standard library are added to the models by doing a drag and drop from the `Library Window` (see Figure 2-3 and Section 2.5.1). Navigate to the component model in the library tree, click on it, drag it to the model you are building while pressing the mouse left button, and drop the component where you want to place it in the model.

Similarly, the component models present in the Modelica Tree View, i.e. the Custom Modelica Models also can be added into some other custom Modelica models by a similar drag and drop. The dropped component keeps getting updated as soon as we make any changes in the original model.

For this example we will add four components as instances of the models `Ground`, `Resistor`, `Inductor` and `EMF` from the `Modelica.Electrical.Analog.Basic` package, an instance of the model `SignalVoltage` from the `Modelica.Electrical.Analog.Sources` package, one instance of the model `Inertia` from the `Modelica.Mechanics.Rotational.Components` package and one last instance of the model `Step` from the `Modelica.Blocks.Sources` package.

2.2.3 Making Connections

In order to connect one component model to another the user simply clicks on any of the ports. Then it will start displaying a connection line. Then move the mouse to the target component where you want to finish the connection and click on the component port where the connection should end. You do not need to hold the mouse left button down for drawing connections.

In order to have a functioning `DCmotor` model, connect the `Resistor` to the `Inductor` and the `SignalVoltage`, `EMF` to `Inductor` and `Inertia`, `Ground` to `SignalVoltage` and `EMF`, and finally `Step` to `SignalVoltage`. Check Figure 2-7 to see how the `DCmotor` model looks like after connections.

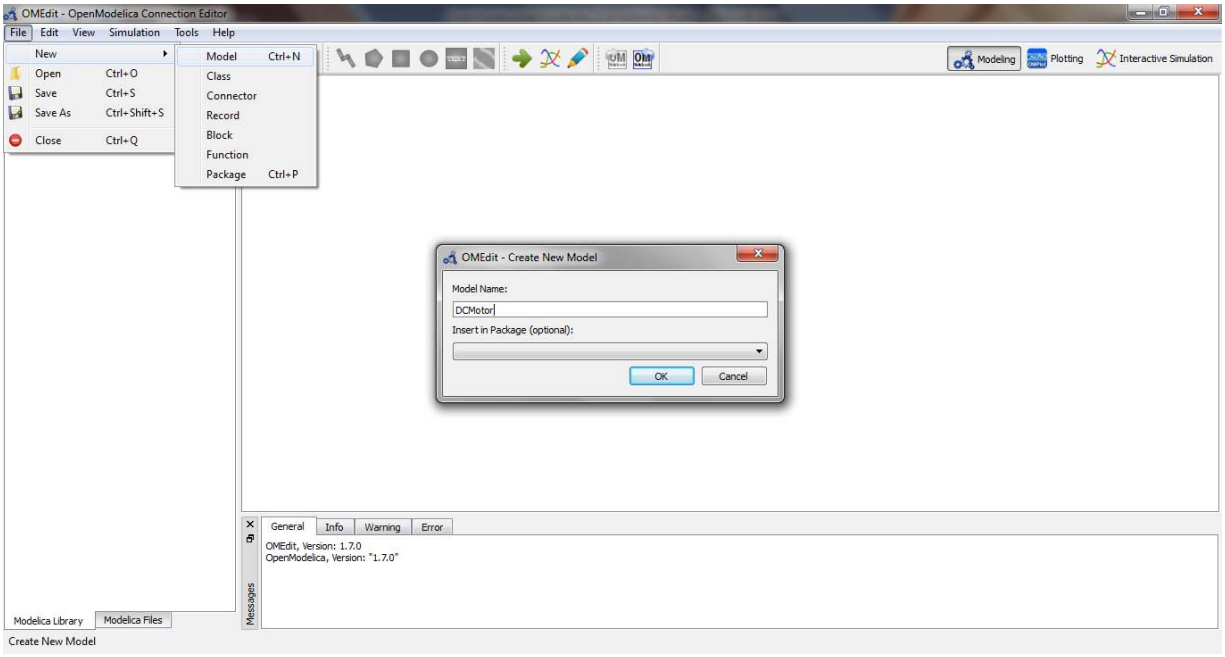


Figure 2-4: Creating a new model.

In order to connect one component model to another the user first enables the connect mode from the toolbar to make the connect mode active. See **Figure 2-5**.

Figure 2-5: Connect mode button.

The different kinds of connections are:

- *Connections for Models* – If the connect mode is active then user then simply clicks on any of the ports.
- *Connection for Connector Types* – If the connect mode is active then we can also connect two components of connector types to each other if they are of the same type. Since, it is a connector type, to start or end a connection, you can just click anywhere on the component icon and it will start.
- *Connection for Connector Array Type* – If any of the start port or the end port of the connection is of an array type, the user also needs to add indices at which the connection is to be made. For this a dialog box will pop up, as soon as the user clicks on the end port of the connection asking the user for indices of the array for whichever port it is necessary. For example, let's say, the user wants to connect, the x component instance of `Modelica.Electrical.Digital.Converters.BooleanToLogic` to the y component instance of `Modelica.Blocks.Sources.BooleanConstant`. Now since, `Modelica.Electrical.Digital.Converters.BooleanToLogic.x` is a connector array, as soon as the user clicks on the end port for this connection, a dialog box appears asking user the index of the start instance. See Figure 2-6.

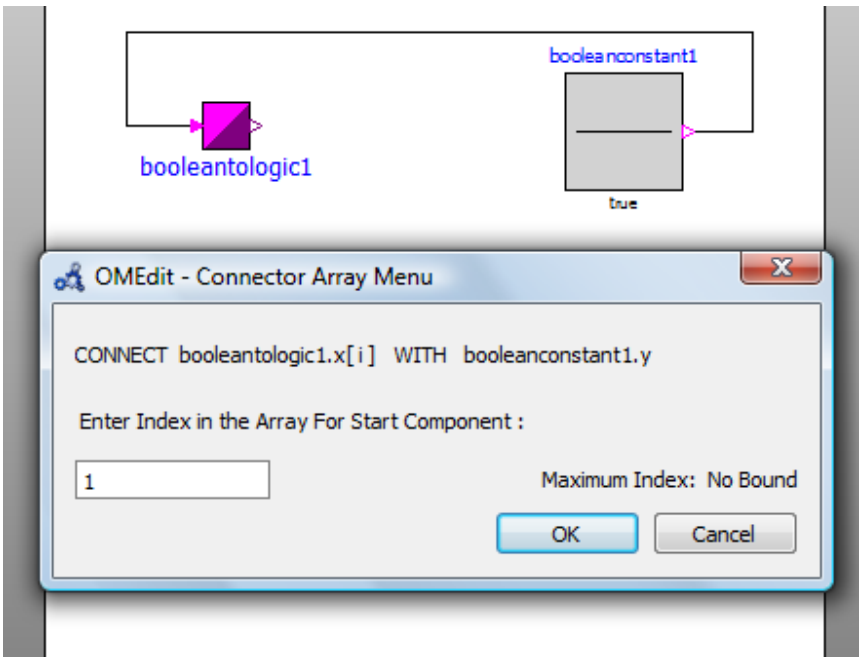


Figure 2-6: Connector Array Menu

2.2.4 Simulating the Model

The OMEdit Simulation dialog (see Figure 2-8, Section 2.6.2) can be launched either from `Simulation > Simulate` or by clicking the `simulate icon` from the toolbar. Once the user clicks on `Simulate!` button, OMEdit starts the simulation process, at the end of the simulation process the Plot Variables Window (Figure 2-9, Section 2.5.3) useful for plotting will appear at the right side. Figure 2-8 shows the simulation dialog.

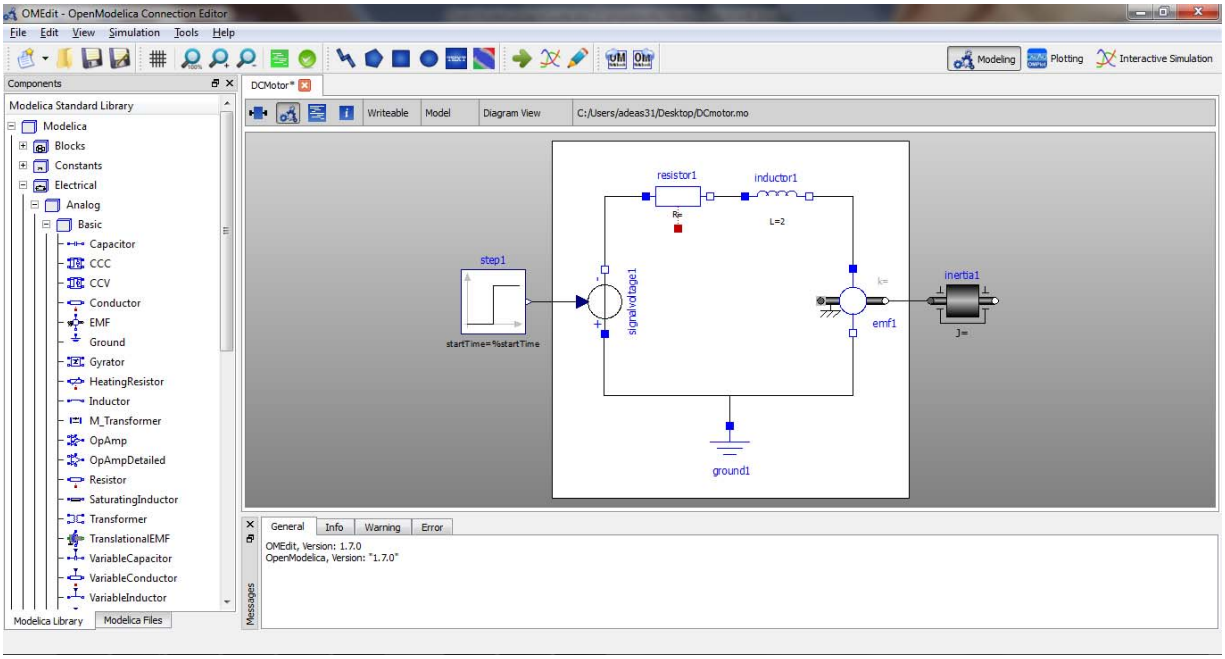


Figure 2-7: DCmotor model after connections.

2.2.5 Plotting Variables from Simulated Models

The instance variables that are candidate for plotting are shown in the right dock window. This window is automatically launched once the user simulates the model; the user can also launch this window manually either from `Simulation > Plot Variables` or by clicking on the `plot icon` from toolbar. It contains the list of variables that are possible to use in an OpenModelica plot. The plot variables window contains a tree structure of variables; there is a checkbox beside each variable. The user can launch the plotted graph window by clicking the checkbox.

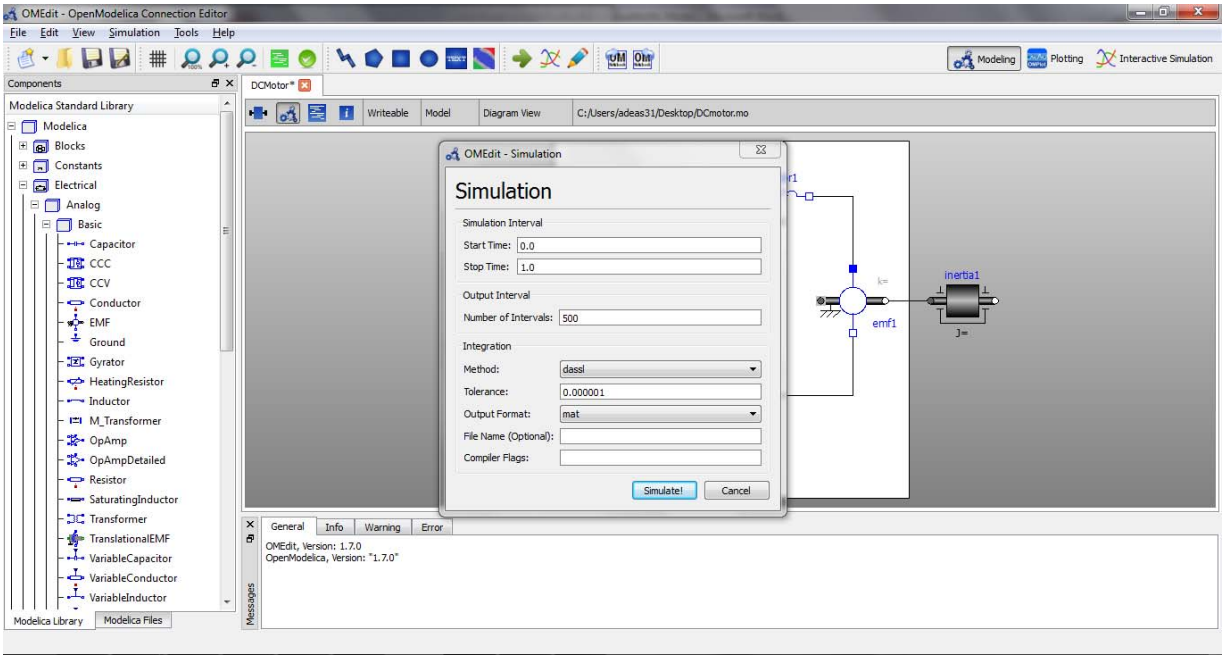


Figure 2-8: Simulation Dialog.

Figure 2-9 shows the complete `DCmotor` model along with the list of plot variables and an example plot window.

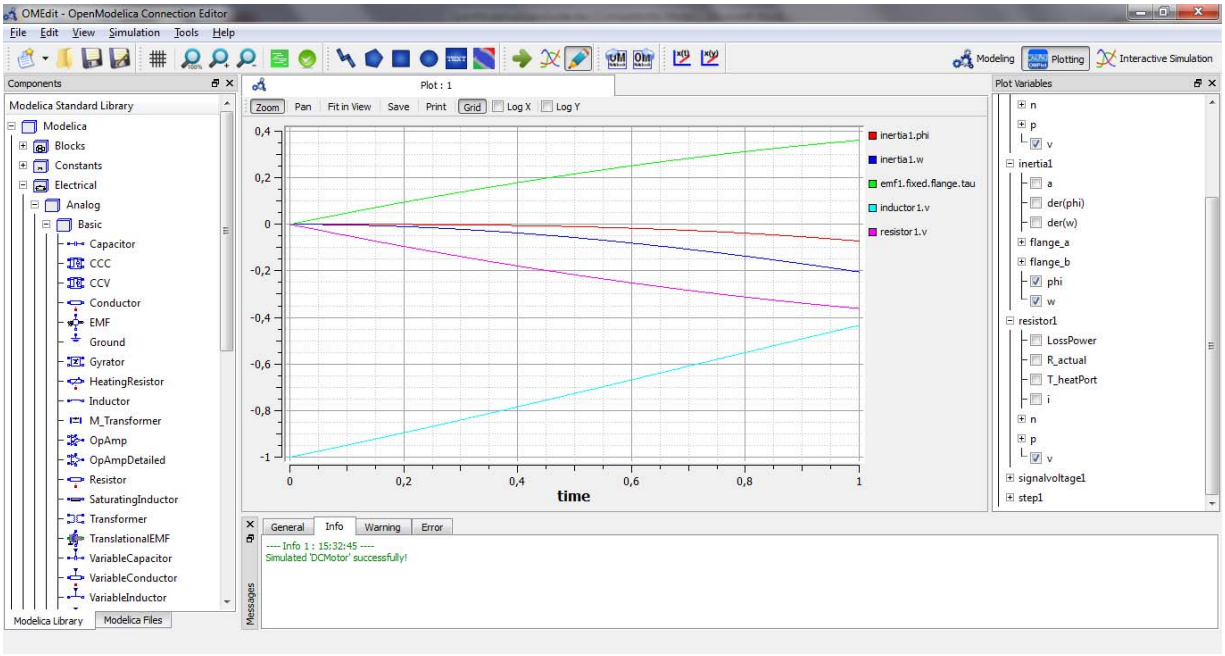


Figure 2-9: Plotted variables.

2.3 How to Create User Defined Shapes – Icons

Users can create shapes of their own by using the shape creation tools available in OMEdit.

- *Line Tool* – Draws a line. A line is created with a minimum of two points. In order to create a line, the user first selects the line tool from the toolbar and then click on the `Designer Window`; this will start creating a line. If a user clicks again on the `Designer Window` a new line point is created. In order to finish the line creation, user has to double click on the `Designer Window`.
- *Polygon Tool* – Draws a polygon. A polygon is created in a similar fashion as a line is created. The only difference between a line and a polygon is that, if a polygon contains two points it will look like a line and if a polygon contains more than two points it will become a closed polygon shape.
- *Rectangle Tool* – Draws a rectangle. The rectangle only contains two points where first point indicates the starting point and the second point indicates the ending the point. In order to create rectangle, the user has to select the rectangle tool from the toolbar and then click on the `Designer Window`, this click will become the first point of rectangle. In order to finish the rectangle creation, the user has to click again on the `Designer Window` where he/she wants to finish the rectangle. The second click will become the second point of rectangle.
- *Ellipse Tool* – Draws an ellipse. The ellipse is created in a similar way as a rectangle is created.
- *Text Tool* – Draws a text label.
- *Bitmap Tool* – Draws a bitmap container.

The shape tools are located at the top in the toolbar. See Figure 2-10.

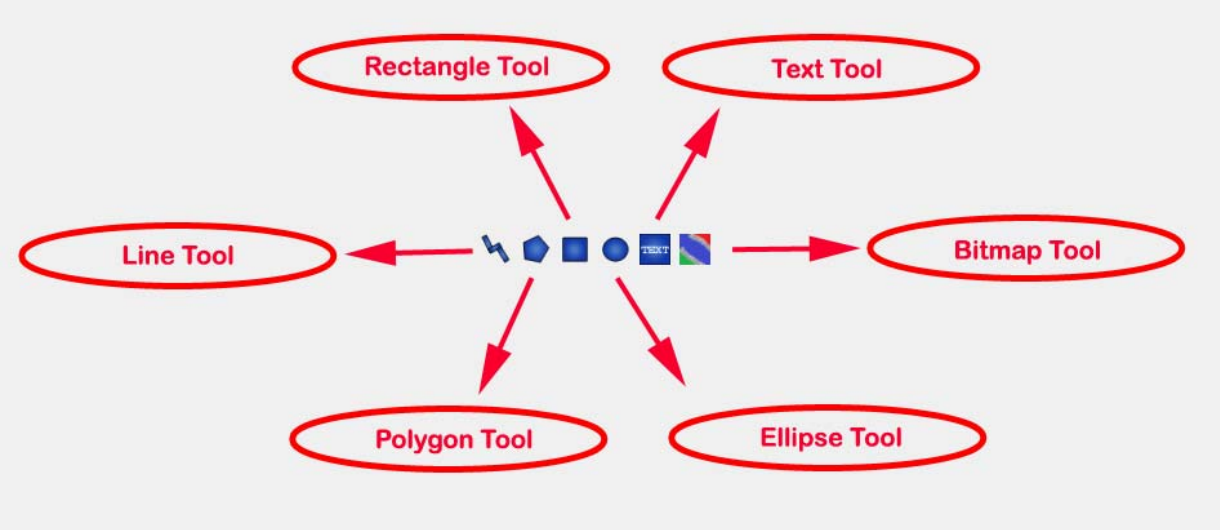


Figure 2-10: User defined shapes.

The user can select any of the shape tools and start drawing on the `Designer Window`. The shapes created on the `Diagram View` of `Designer Window` are part of the diagram and the shapes created on the `Icon View` will become the icon representative of the model.

For example, if a user creates a model with name `testModel` and add a rectangle using the rectangle tool and a polygon using the polygon tool, in the `Icon View` of the model. The model's `Modelica Text` will appear as follows:

```
model testModel
```

```
annotation(Icon(graphics = {Rectangle(rotation = 0, lineColor = {0,0,255}, fillColor =
{0,0,255}, pattern = LinePattern.Solid, fillPattern = FillPattern.None, lineThickness
= 0.25, extent = {{ -64.5,88},{63, -22.5}}),Polygon(points = {{ -47.5, -29.5},{52.5, -
29.5},{4.5, -86},{ -47.5, -29.5}}, rotation = 0, lineColor = {0,0,255}, fillColor =
{0,0,255}, pattern = LinePattern.Solid, fillPattern = FillPattern.None, lineThickness
= 0.25)}));
end testModel;
```

In the above code snippet of `testModel`, the rectangle and a polygon are added to the icon annotation of the model. Similarly, any user defined shape drawn on a `Diagram View` of the model will be added to the diagram annotation of the model.

2.4 OMEdit Views

OMEdit has three kinds of views.

2.4.1 Modeling View

This is the default view. This view shows the `Designer Window` and allows users to create their models.

2.4.2 Plotting View

This view is used for showing plot graphs. The user can launch this view anytime by using the views button in the tool bar. This view also becomes active automatically when user simulates the model successfully.

2.4.3 Interactive Simulation View

This view is quite similar to `Plotting View`. One of the primary differences is that `Plotting View` is used to show graphs of pre-built models that cannot be changed. However, in the `Interactive Simulation View` the user can change the values of variables and parameters of the model at runtime.

2.5 OMEdit Windows/Tabs

The OMEdit GUI contains several windows that shows different views to users:

- *Library Window* for the Modelica Standard Library.
- Drawing interface in the form of *Designer Window*.
- *Plot Window* contains the list of instance variables.
- *Messages Window* displays the informational, warning and error messages.
- *Documentation Window* displays the Modelica annotations based documentation in a QWebView.
- *Model Browser Window* displays the component hierarchy of the model.

2.5.1 Library Window

The Modelica Standard Library is automatically loaded in OMEdit and is located on the left dock window. Once a Modelica model has been created then the user can just drag and drop components into the model from the MSL, the `Library Window`. The available libraries in the MSL are:

- Blocks
- Constant
- Electric
- Icons
- Magnetic
- Math
- Mechanics
- SIunits
- Thermal
- UsersGuide
- Utilities

The `Library Window` consists of two tabs one shows the Modelica Standard Library and is selected by default the other tab shows the Modelica files that user creates in OMEdit.

2.5.1.1 Viewing Models Description

In order to view the model details, double click the component and details will be opened in `Designer Window`. Alternative way is to right click on the component and press `Show Component,` it will do the same.

2.5.1.2 Viewing Models Documentation

Right click the model in the `Library Window` and select `View Documentation`; it will launch the `Documentation Window`. See Figure 2-11.

2.5.1.3 How to Open an Existing Model?

- Go to `File->Open`. An Open Model Dialog Appears. The user can go to the required file location and click on it to open the model in OMEdit.
- The user can also drag any model from an outside window and drop it into the Main Window of OMEdit to open that model.

2.5.1.4 How to create a Copy of an Existing Model?

To copy a model, the user can simply right click on the model in the `Library Window`, and then select `copy`. To paste the copied model the user can click on any empty space in the `Library Window` and click on the `paste` Option. A copy of the model will be created initialized with a new name.

2.5.1.5 How to Check a Model?

Right click the component in the library window and select ***Check***; it will launch the ***Check Dialog***. See Figure 2-11.

2.5.1.6 How to Instantiate a Model?

Right click the component in the library window and select `Instantiate Model`; it will launch the Instantiate Model `Dialog`. See Figure 2-11.

2.5.1.7 How to Rename a Model?

Right click the model in the `Library Window` and select `Rename`; it will launch the `Rename Dialog`. See Figure 2-11.

2.5.1.8 How to Delete a Model?

Right click the model in the `library window` and select `Delete`; a popup will appear asking "Are you sure you want to delete?"

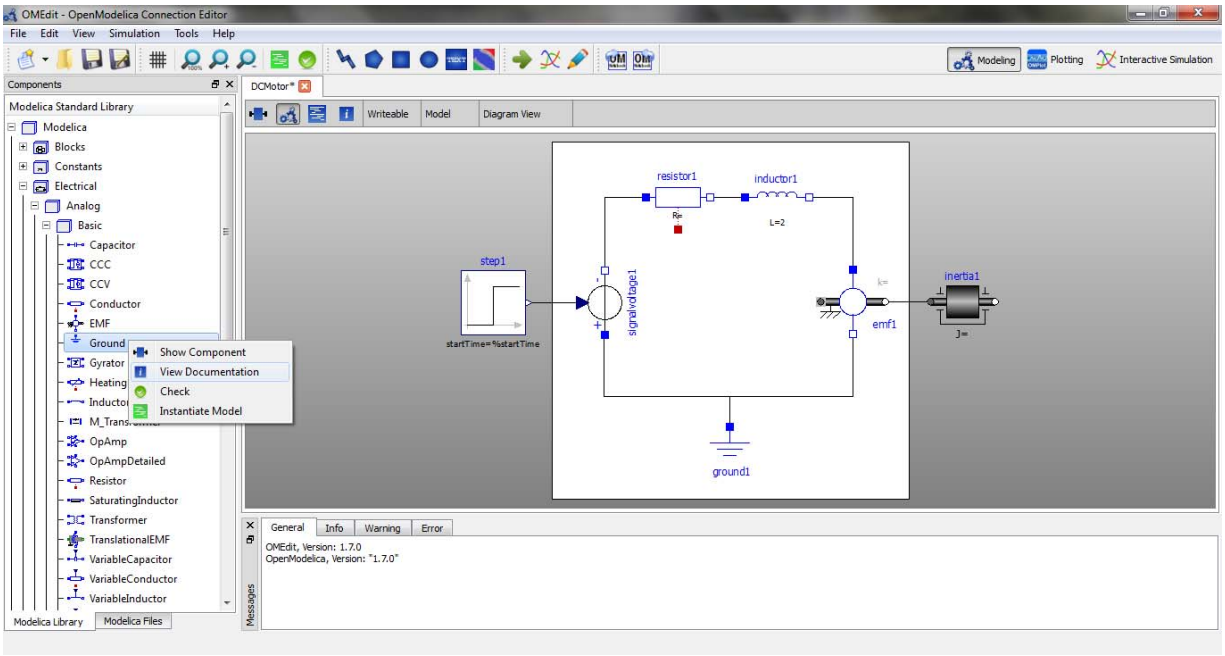


Figure 2-11: Context menu to view component model details.

2.5.2 Designer Window

The Designer Window is the main window of OMEdit. It consists of three views,

- *Icon View* - Shows the model icon view.
- *Diagram View* - Shows the diagram of the model created by the user.
- *Modelica Text View* - Shows the Modelica text of the model.

2.5.3 Plot Variables Window

The right dock window represents the `Plot Variables Window`. It consists of a tree containing the list of instance variables that are extracted from the simulation result. Each item of the tree has a checkbox beside it. The user can click on the check box to launch the plot graph window. The user can add/remove the variables from the plot graph window by marking/unmarking the checkbox beside the plot variable.

2.5.4 Messages Window

`Messages Window` is located at the bottom of the application. The Messages Window consists of 4 types of messages,

- *General Messages* – Shown in black color.
- *Informational Messages* – Shown in green color.
- *Warning Messages* – Shown in orange color.
- *Error Messages* – Shown in red color.

2.5.5 Documentation Window

This window is shown when a user right clicks the component in the library window and selects `View Documentation`. This shows the OpenModelica documentation of components in a web view. All externals links present in the documentation window are opened in the default browser of the user. All local links are opened in the same window. Figure 2-12 shows the `Documentation Window` view.

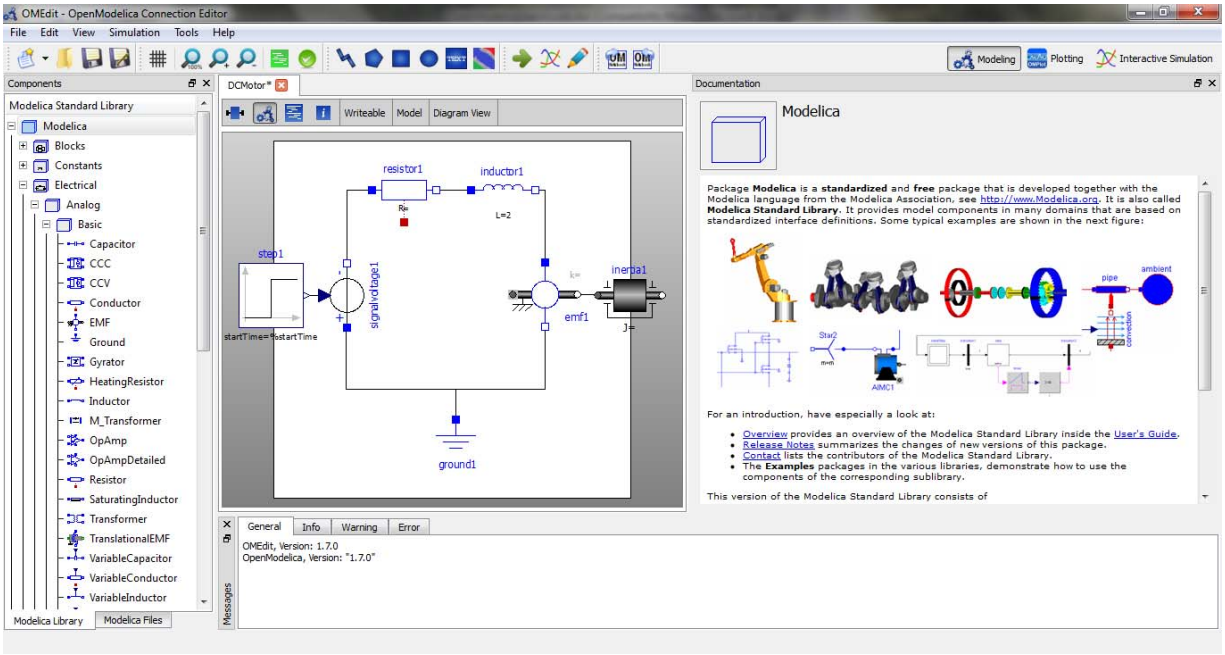


Figure 2-12: Documentation Window.

2.5.6 Model Browser Window

The Model Browser Window is located on the left bottom dock window below the Library Window. It lays the outline of the currently opened model and show all the component heirarchy in a tree format. See Figure 2-13.

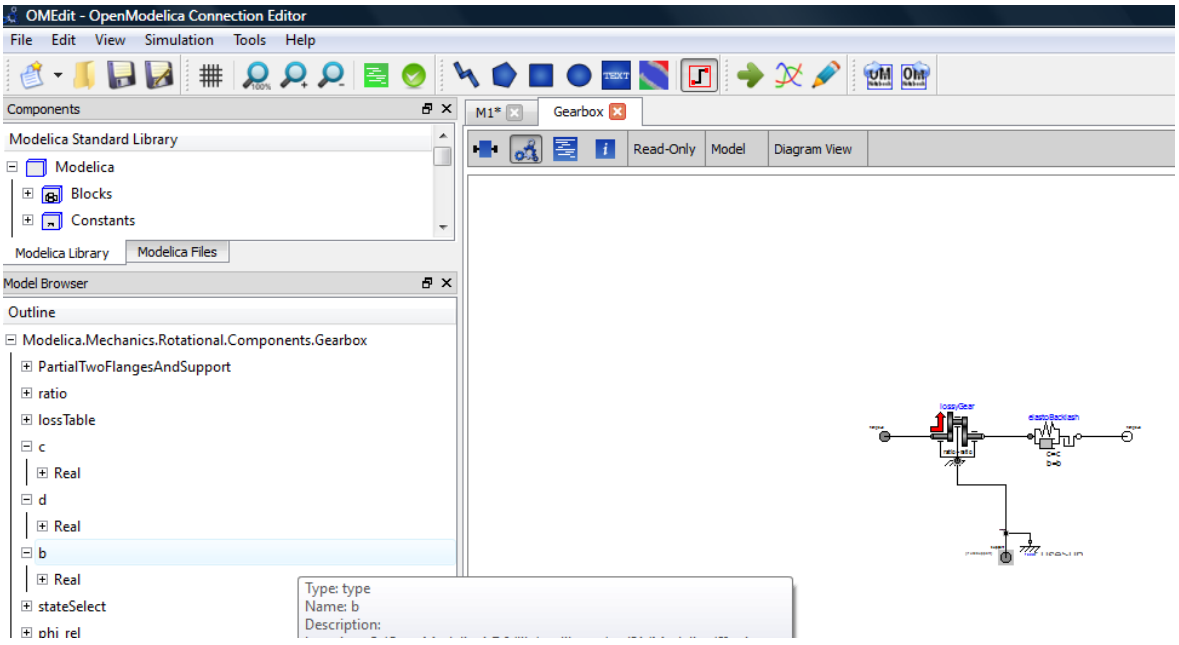


Figure 2-13: Model Browser Window.

2.6 Dialogs

Dialogs are a kind of window that is not visible by default. The user has to launch them or they will automatically appear due to some user action. The following dialogs are available:

- *New Dialog* for creating new Modelica models.
- *Simulation Dialog* for simulating Modelica models.
- *Model Properties Dialog.*
- *Model Attributes Dialog.*

2.6.1 New Model Dialog

The `New Dialog` can be launch from `File > New > Model Type`**.** Model type can be `model`, `class`, `connector`, `record`, `function` and `package`.

2.6.2 Simulation Dialog

`Simulation Dialog` can be launched either from `Simulation > Simulate` or by clicking on the `Simulate!` button in the toolbar. Figure 2-8 shows a simulation dialog. The simulation dialog consists of simulation variables. You can set the value of any variable, depending on the simulation requirement. Simulation variables are,

- Simulation Interval
  - Start Time
  - Stop Time
- Output Interval
  - Number of Intervals
  - Output Interval
- Integration
  - Method
  - Tolerance
  - Fixed Step Size

2.6.3 Model Properties Dialog

The models that are placed in the `Designer Window` can be modified by changing the properties. In order to launch the `Model Properties Dialog` of a particular model right click the model and select `Properties`. See Figure 2-14. The properties dialog contains the name of the model, class name the model belongs to and the list of parameters of the component.

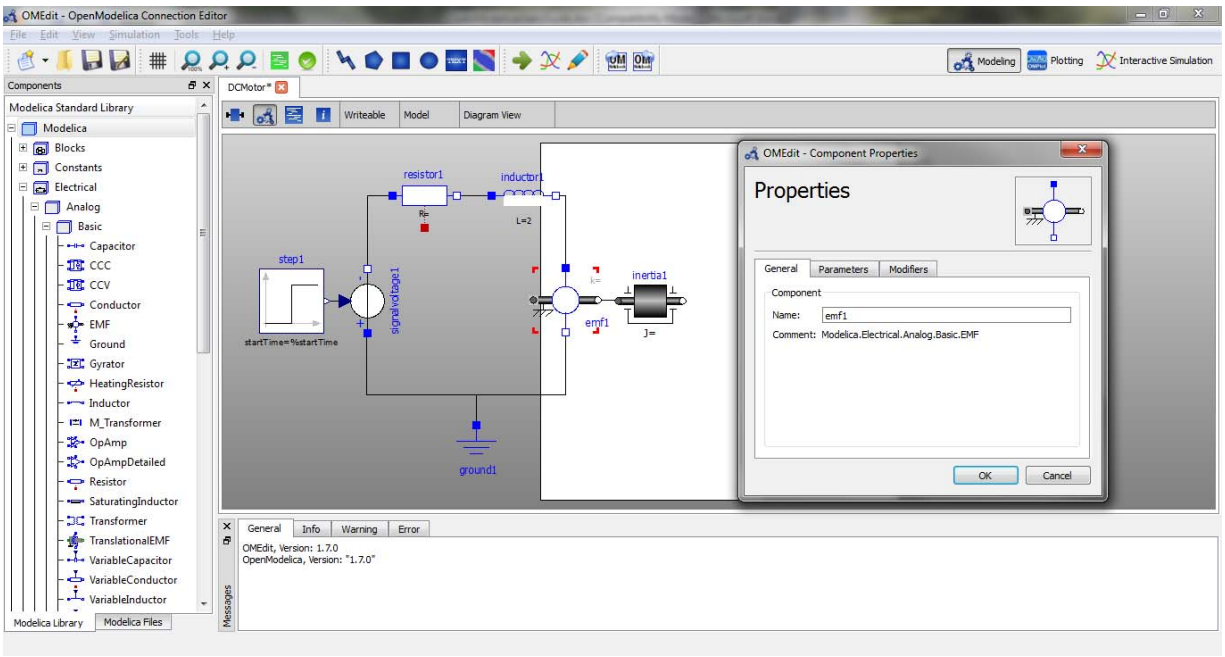


Figure 2-14: Properties Dialog.

2.6.4 Model Attributes Dialog

Right click the model placed in the `Designer Window` and select `Attributes`. It will launch the attributes dialog. Figure 2-15 shows the `Model Attributes Dialog`.

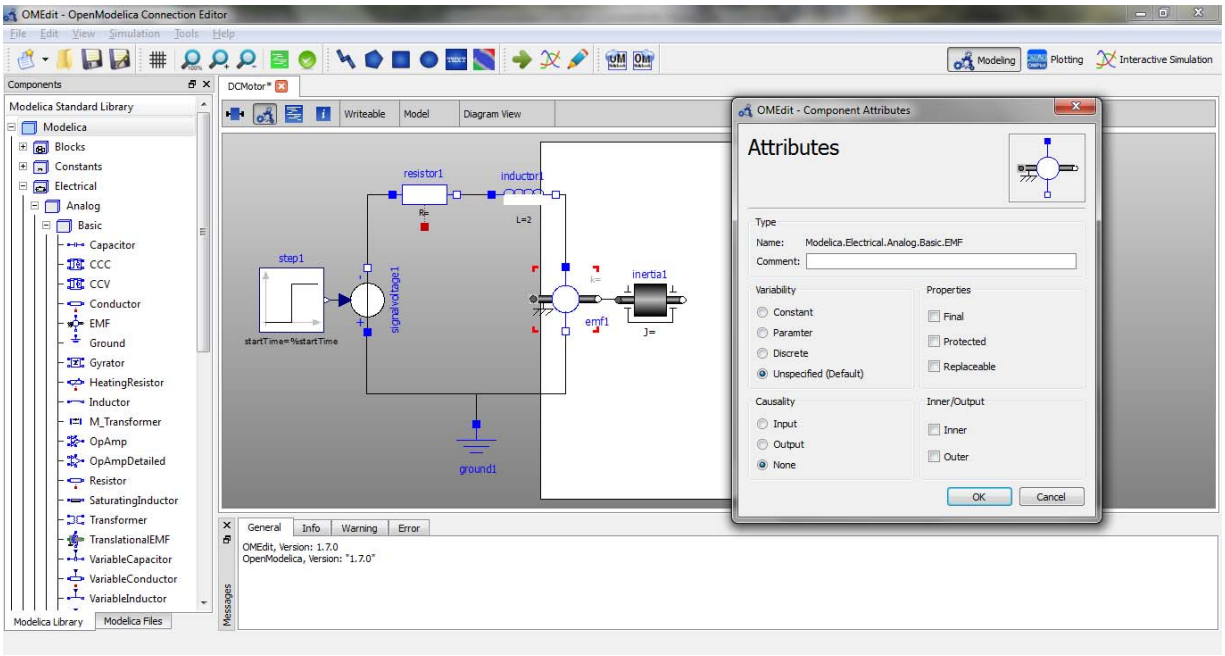


Figure 2-15: Attributes Dialog.

2.7 Interactive Simulation in OMEdit

OMEdit uses the OpenModelica Interactive (OMI) subsystem to perform the interactive plotting. The OMI uses the TCP/IP technique to transfer data back and forth. OMEdit connects with OMI through TCP sockets.

2.7.1 Invoking Interactive Simulation

`Interactive Simulation Dialog` can be launched either from `Simulation > Interactive Simulation` or by clicking on the `Interactive Simulation!` button in the toolbar. Interactive Simulation Dialog looks similar to the Simulation Dialog but it differs in functionality, instead of performing normal pre-built simulation it performs online interactive simulation where simulation responds in real-time to user input and changes to parameters.

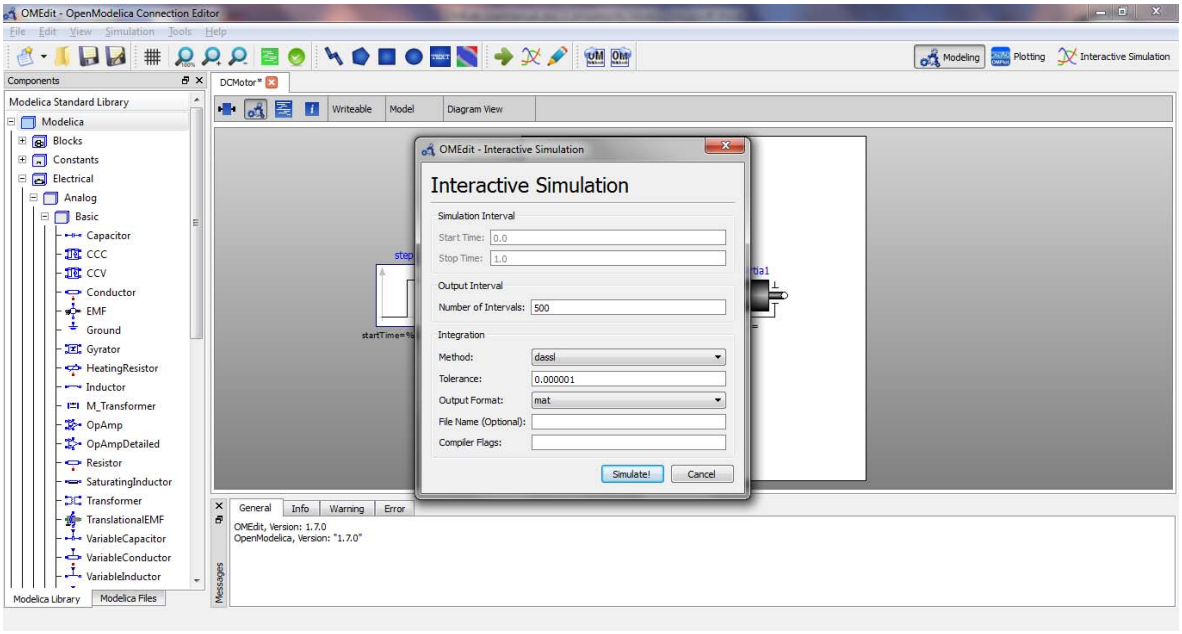


Figure 2-16: Interactive Simulation Dialog.

2.7.2 Interactive Simulation View

Once your model was successfully built using the `Interactive Simulation Dialog`, the `Interactive Simulation View` will become active automatically. `Interactive Simulation View` contains,

- *Graph* – It contains a graph which is used to display the values of selected variables over the time.
- *Parameters* – The parameters of the model are shown on the right top section with the default values.
- *Variables* – The right bottom section contains the list of variables that user can select for interactive plotting.
- *Initialize Button* – This button is used to send the information of changed parameters and checked variables to the OpenModelica Interactive subsystem.

- *Start Button* – Once the parameters and variables are initialized and sent to the OMI. Then the user can click on start button and start the interactive plotting.
- *Pause Button* – This button pauses the running interactive plotting.
- *Stop Button* – Clears everything but does not remove the connection with OMI. After clicking stop button user has to reinitialize everything and start the interactive plotting again.
- *Shut Down Button* – Disconnects from OMI and closes the interactive simulation session.
- *Show OMI Log Button* – Pops up a log window which displays the messages exchanged between OMEdit and OMI.

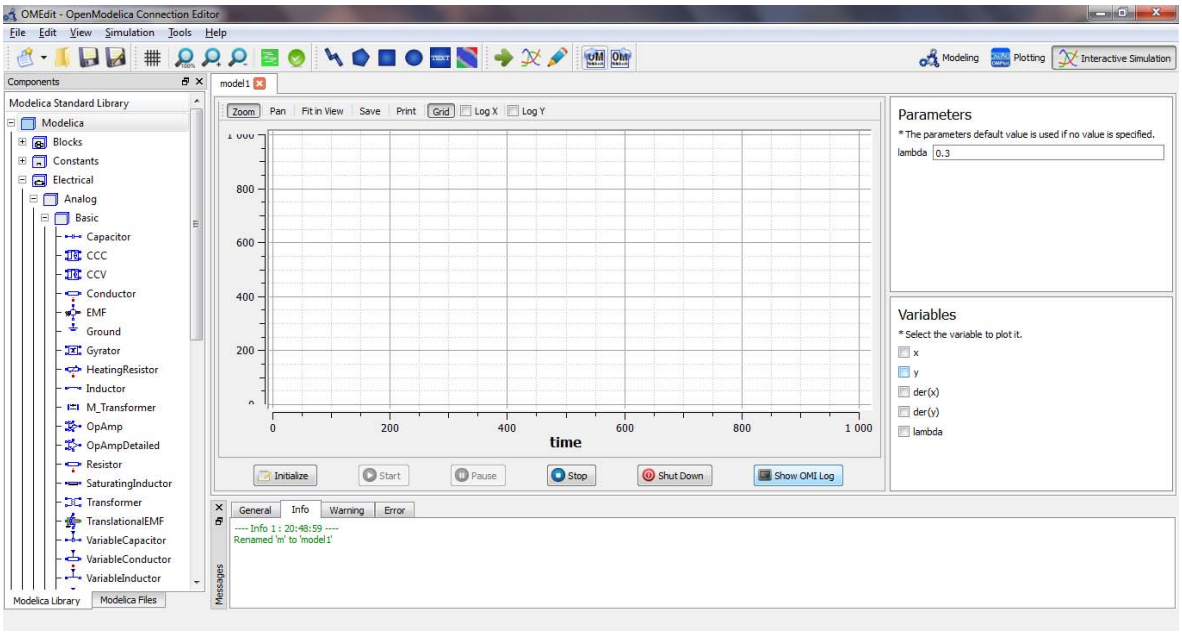


Figure 2-17: Interactive Simulation View.

Chapter 3

2D Plotting and 3D Animation

This chapter covers 2D plotting available from OMNotebook, OMShell or programmable plotting from your own Modelica model. The 3D animation is currently turned off by default, but will be available in an enhanced version in a future release.

3.1 Enhanced Qt-based 2D Plot Functionality

Starting with OpenModelica 1.4.5, enhanced plotting functionality is available (Eriksson, 2008). The enhanced plotting is implemented based on a Qt-based (Trolltech, 2007) GUI package. This new plotting functionality has additional features compared to the old Java-based PtPlot plotting. The simulation data is sent directly to the plotting window in OMNotebook (or a popup window if called from OMShell), which handles the presentation (see Figure 3-2). As OMNotebook now has access to all source data it is now possible to manipulate diagrams, e.g. zoom or change scales.

To allow the use of graphics functions from within Modelica models a new Modelica API has been developed. This utilizes an external library to communicate with OMNotebook. In addition to this, a number of new functions that can be used for drawing geometric objects like circles, rectangles and lines have been added.

The following is a summary of the capabilities of the new 2D graphics package:

- *Interaction with OMNotebook.* The graphics package has been developed to be fully integrated with OMNotebook and allow modifications of diagrams that have been previously created.
- *Usage without OMNotebook*. If the functionality of the graphics package is used without OMNotebook, a new window should be opened to present the resulting graphics.
- *Logarithmic scaling*. Some applications of OpenModelica produce simulation data with large value ranges, which is hard to make good plots of. One solution to this problem is to scale the diagram logarithmically, and this is allowed by the graphics package.
- *Zoom*. To allow studying of small variations the user is allowed to zoom in and out in a diagram.
- *Support for graphic programming*. To allow creation of Modelica models that are able to draw illustrations, show diagrams and suchlike, it is possible to use the graphics package not only from the external API of OMC, but also from within Modelica models. To accomplish this a new Modelica interface for the graphics package has been created.
- *Programmable Modelica API*. The Modelica API is defined by a number of Modelica functions, located in the package `Modelica.Graphics.Plot`, which use external libraries to access functionality of the graphics package.

The programmable Modelica API functions include the following:

- `plot`(x). Draws a two-dimensional line diagram of *x* as a function of time.
- `plotParametric(x,y)`. Draws a two-dimensional parametric diagram with *y* as a function of *x*.
- `plotTable`([x1, .., y1; .. ; x*n*, .., y*n*]). Draws a two-dimensional parametric diagram with *y* as a function of *x*.
- `drawRect`(x1, x2, y1, y2). Draws a rectangle with vertices in (x1, y1) and (x2, y2).
- `drawEllipse`(x1, x2, y1, y2). Draws an ellipse with the size of a rectangle with vertices in (x1, y1) and (x2, y2).
- `drawLine`(x1, x2, y1, y2). Draws a line from (x1, y1) to (x2, y2).

A faster and more stable plot implementation is available from the OpenModelica 1.7 release. Currently these commands can be used by giving the `plot3`(…) *command instead of* `plot`(…)*. In future releases the faster plot implementation will be the default.*

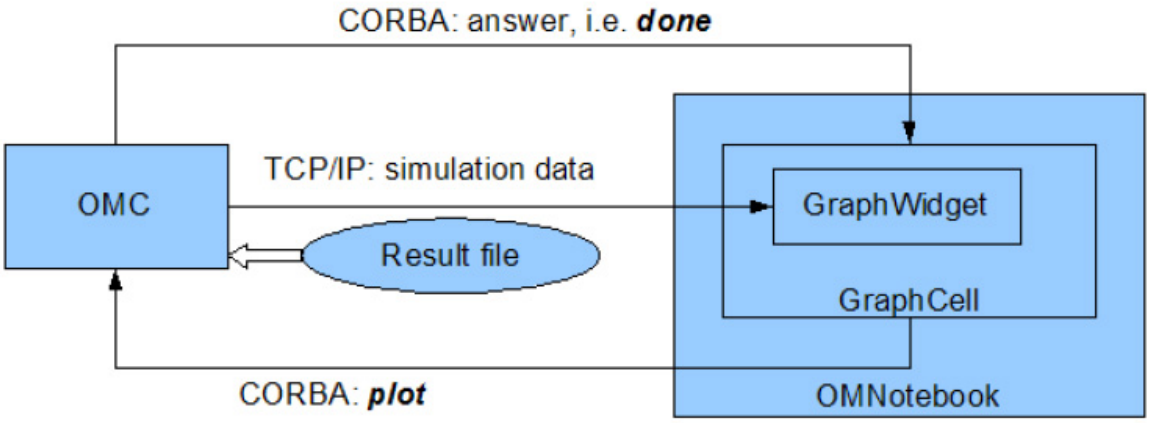


Figure 3-1. Plotting architecture with the new 2D graphics package.

3.2 Simple 2D Plot

To create a simple time plot the model `HelloWorld` defined in DrModelica is simulated. To reduce the amount of simulation data in this example the number of intervals is limited with the argument `numberOfIntervals=10`. The simulation is started with the command below.

```
simulate(HelloWorld, startTime=0, stopTime=4, numberOfIntervals=10);
```

When the simulation is finished the file `HelloWorld res.plt` contains the simulation data. The contents of the file is the following (some formatting has been applied).

```
0                       1
4.440892098500626e-013 0.9999999999995559
0.4444444444444444     0.6411803884299349
0.8888888888888888     0.411112290507163
1.333333333333333      0.2635971381157249
1.777777777777778      0.1690133154060587
2.222222222222222      0.1083680232218813
```

```
2.666666666666667       0.06948345122279623
3.111111111111112       0.04455142624447787
3.555555555555556       0.02856550078454138
4                       0.01831563888872685
```

Diagrams are now created with the new graphics package by using the following command.

```
plot(x);
```

seems to correspond well with the data.

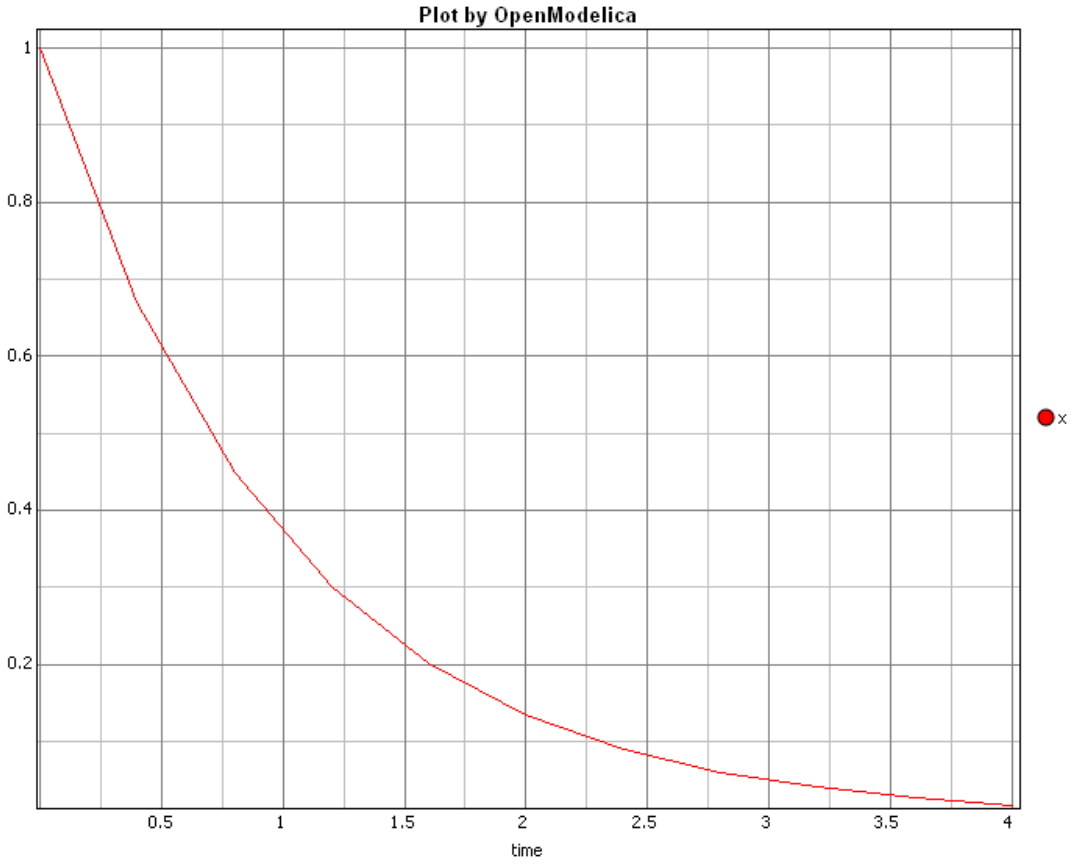


Figure 3-2. Simple 2D plot of the HelloWorld example.

By re-simulating and saving results at many more points, e.g. using the default 500 intervals, a much smoother plot can be obtained.

```
simulate(HelloWorld, startTime=0, stopTime=4, numberOfIntervals=500);
plot(x);
```

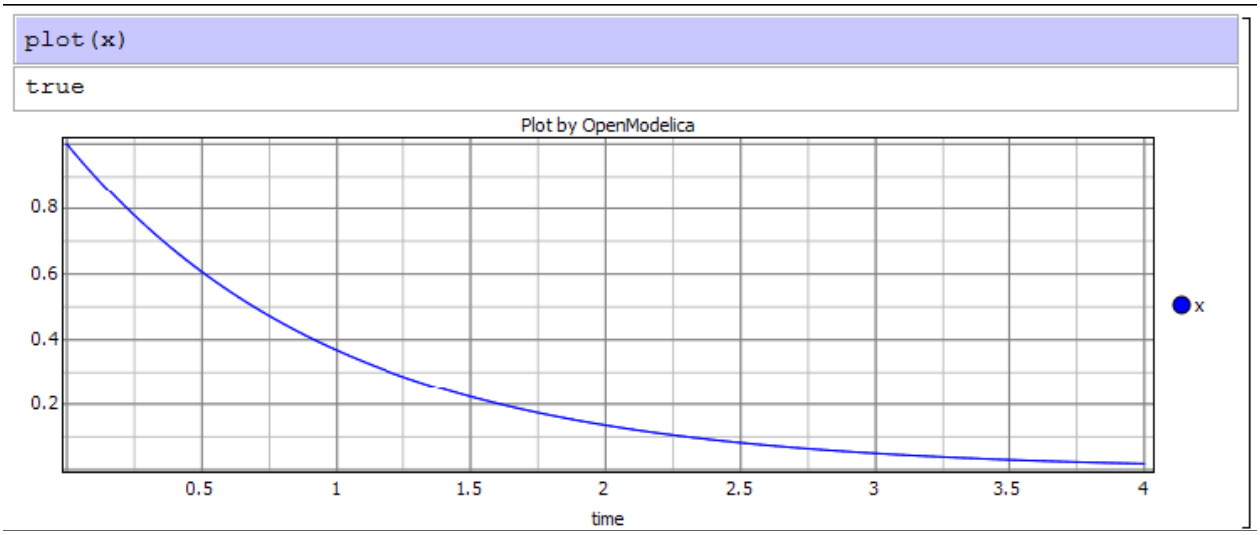


Figure 3-3. Simple 2D plot of the HelloWorld example with larger number of points.

Additional features of the new plotting are shown in Figure 3-4 and Figure 3-5.

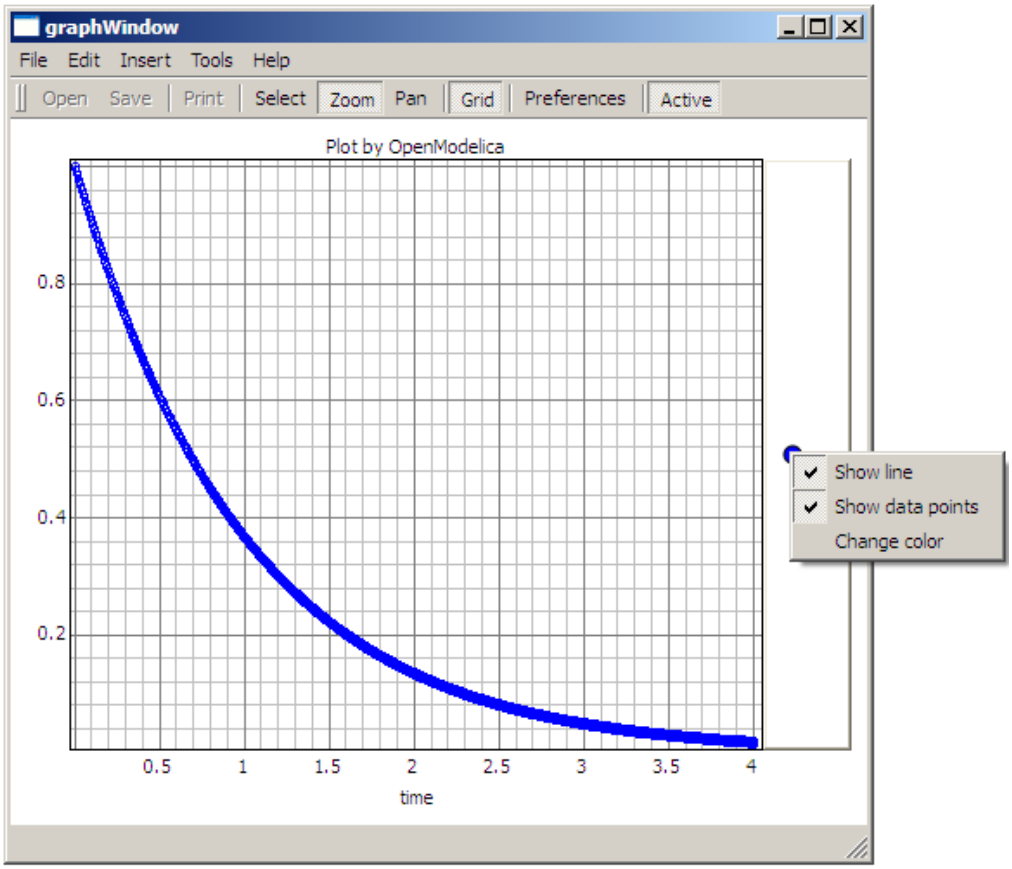


Figure 3-4. Features of the new Qt-based Plotting Package: Show data points, Change line colors, etc.

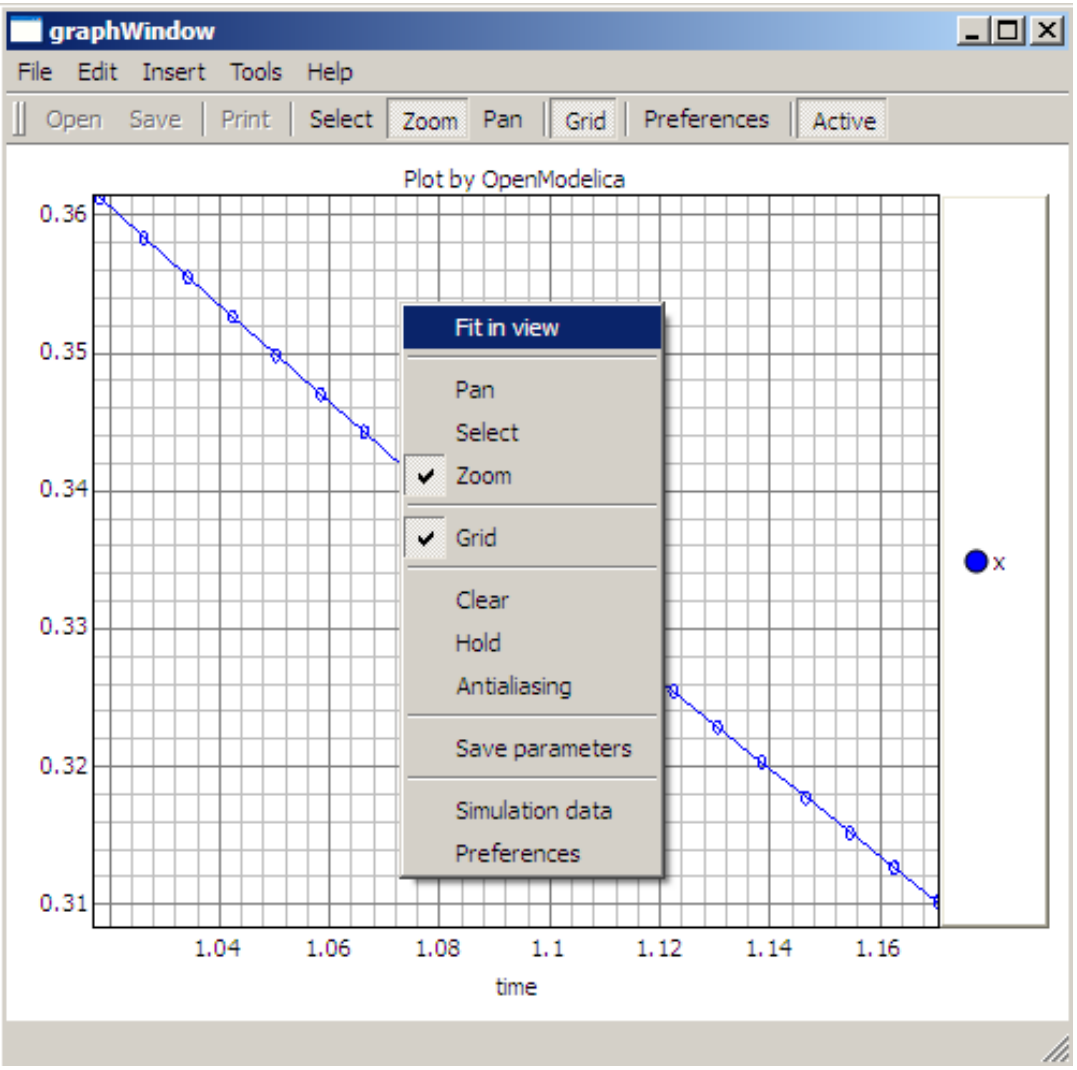


Figure 3-5. Features of the new Qt-based Plotting Package: Zoom, Fit in view, Grid, etc.

3.2.1 Plot Functions and Their Options

All plot functions are part of `ModelicaBuiltin.mo`[4], with additional explanation below.

[4] OPENMODELICAHOME/lib/omc/ModelicaBuiltin.mo

Command	Description
`plot(x)`	Creates a diagram with data from the last simulation that had a variable named x.
`plot({x,y,..., z})`	Like the previous command, but with several variables.
`plotParametric(x, y)`	Creates a parametric diagram with data from the last simulated variables named x and y.
`plotParametric(x, {y1,y2})`	Like the previous command, but with several variables.
`plotAll()`	Creates a diagram with all variables from the last simulated model as functions of time.

All of these commands can have any number of optional arguments to further customize the the resulting diagram. The available options and their allowed values are listed below.

Option	Default value	Description
`fileName`	`The result of the last simulation`	The name of the result-file containing the variables to plot
`grid`	`true`	Determines whether or not a grid is shown in the diagram.
`title`	`"Plot by OpenModelica"`	This text will be used as the diagram title.
`interpolation`	`linear`	Determines if the simulation data should be interpolated to allow drawing of continuous lines in the diagram. "linear" results in linear interpolation between data points, "constant" keeps the value of the last known data point until a new one is found and "none" results in a diagram where only known data points are plotted.
`legend`	`true`	Determines whether or not the variable legend is shown.
`points`	`true`	Determines whether or not the data points should be indicated by a dot in the diagram.
`logX`	`false`	Determines whether or not the horizontal axis is logarithmically scaled.
`logY`	`false`	Determines whether or not the vertical axis is logarithmically scaled.
`xRange`	`{0, 0}`	Determines the horizontal interval that is visible in the diagram. {0, 0} will select a suitable range.
`yRange`	`{0, 0}`	Determines the vertical interval that is visible in the diagram. {0, 0} will select a suitable range.
`antiAliasing`	`false`	Determines whether or not antialiasing should be used in the diagram to improve the visual quality.
`vTitle`	`""`	This text will be used as the vertical label in the diagram.
`hTitle`	`"time"`	This text will be used as the horizontal label in the diagram.

3.2.2 Zooming

The left mouse button can for instance be used for zooming in on interesting parts of the diagram.The same result can be achieved by using the optional parameters `xRange` and `yRange`. The `plotParametric` command would then look like the following.

```
plotParametric(x, y, xRange={0.9, 1.95}, yRange={-1.5, 1.35})
```

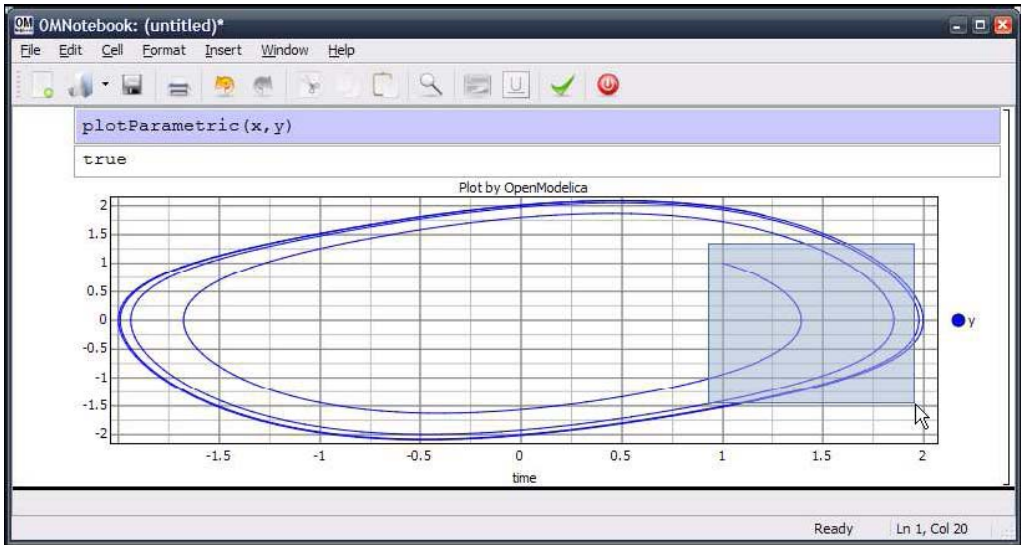


Figure 3-6. Zooming in an Input cell.

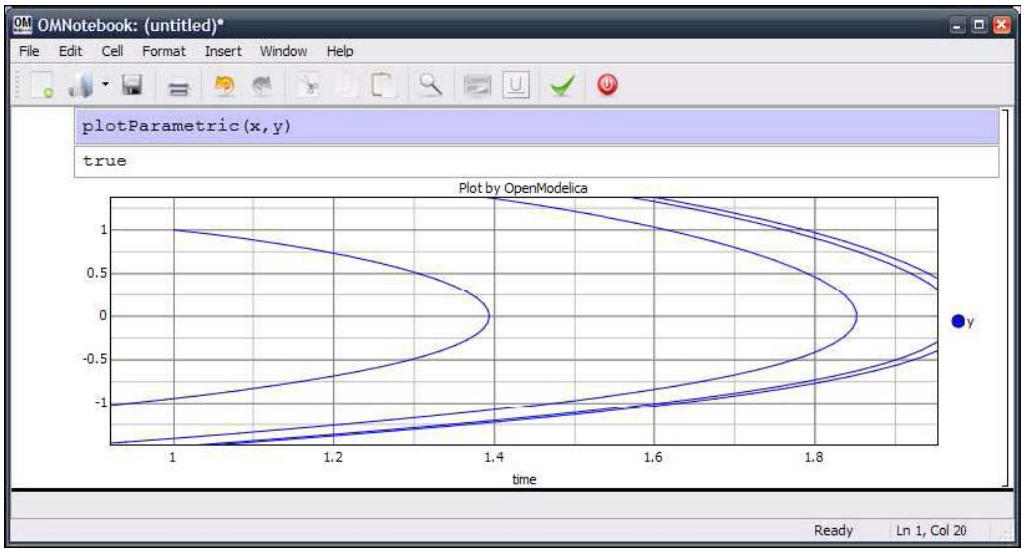


Figure 3-7. Magnified input cell.

3.2.3 Plotting all variables of a model

A command, `plotAll`, has been introduced to plot all the variables of a model. This can be useful if a model contains many interesting variables, as it might be easier to remove variables that are not important

than to list all those who are. The command available for this is `plotAll()`.The command below applies `plotAll` to the model HelloWorld. The result is shown in Figure 3-8. The simplest way to remove unimportant variables is to use the `Remove` command in the `Legend` menu.

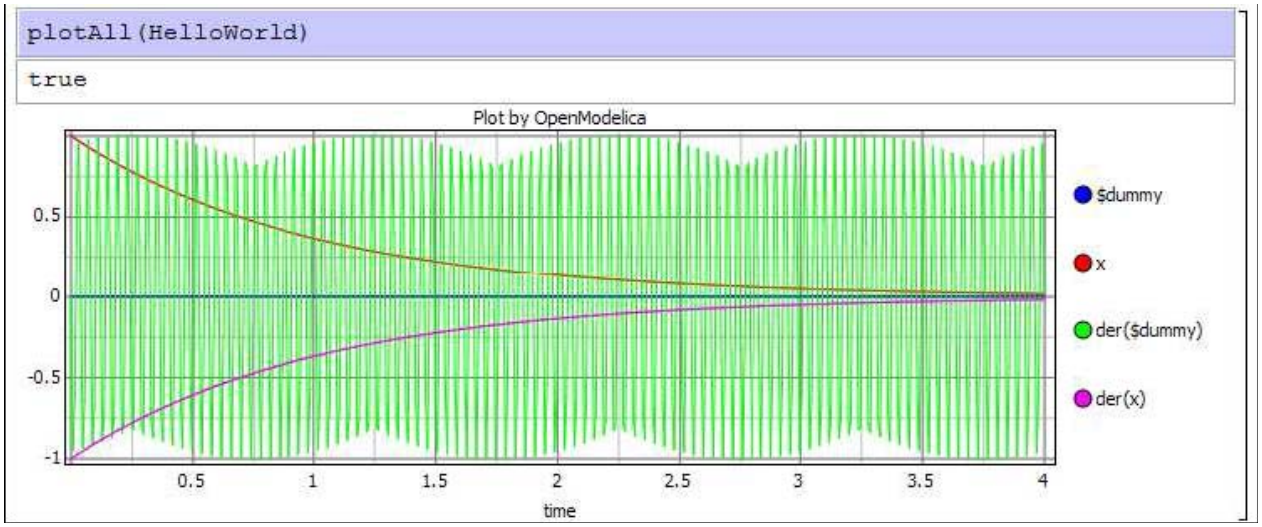


Figure 3-8. Result of the plotAll command.

3.2.4 Plotting During Simulation

When running long simulations, or if plotting without need for commands like `plot` or `plotParametric` is desired, the interface for transfer of simulation data during running simulations can be used. This is enabled by running the following command.

```
enableSendData(true)
```

The same command, but with the parameter *false*, is used to disable the interface. Enabling of the interface has some drawbacks though. The simulation time will be longer as the transfer of data will require some resources.

If the simulation data would have been plotted anyway, some of this time will be saved later however. To reduce the amount of data that has to be transferred, and thereby reduce the time needed to do so, the interesting variables in the model can be specified with the `variableFilter` option of the `simulate` command (see Section 1.2.14 for details). If for instance the model `HelloWorld` is to be simulated the following commands can be used.

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

enableSendData(true);
simulate(HelloWorld, startTime=0, stopTime=25, variableFilter="x");
```

When the simulation data has been transferred the button *D* will appear to the right of the input field. By pressing this dialog `Simulation data` will appear, where new curves can be created.

3.2.5 Programmable Drawing of 2D Graphics

The graphics package provides functions for drawing of basic geometrical objects in the graphics area. These can be used from Modelica models and are executed when the model is simulated. To avoid name conflicts, the functions have been put in the package `Modelica.pltpkg`. The functions of the Modelica programmable plotting interface are described below.

- `plot(model, "x"). Creates a diagram with data from the variable x in the previously simulated model model.`
- `plot(model`, "x, y"). Like the function above, but with more than one variable.
- `plotParametric(model`, "x", "y"). Creates a parametric diagram with data from the variables x and y in the previously simulated model `model`.
- plotTable([x1, y1, z1, ...; x2, y2, z2, ...;...]). Draws y and z as functions of x..
- `clear`().Clears the active GraphWidget.
- `rect`(x_1, x_2, y_1, y_2). Draws a rectangle with vertices in (x_1, y_1) and (x_2, y_2).
- `ellipse`(x_1, x_2, y_1, y_2). Draws an ellipse with the size of a rectangle with vertices in (x_1, y_1) and (x_2, y_2).
- `line`(x1, x2, y1, y2). Draws a line from (x_1, y_1) to (x_2, y_2).
- `hold`(Boolean on). Determines whether or not the active GraphWidget should be cleared before new graphics is drawn.
- `wait`(ms). Waits for (at least) `ms` milliseconds.

The following model shows how these functions can be used to draw ellipses, rectangles, and lines.

```
model testGeom
  parameter Integer n=10;
  protected
  Boolean b[n,n];
equation
  for x in 1:n loop
    for y in 1:n loop
      when initial() then
        if((y == 1) or (y == 10) or (x == 1) or (x == 10)) then
          b[x,y] = pltpkg.rect(x, y, x+1, y+1, fillColor = "blue",
          color = "green");
        else if(y >= 4 and y <= 5 and x >= 4 and x <= 5) then
            b[x,y] = pltpkg.line(x, y, x+1, y+1, color = "red");
          else
            b[x,y] = pltpkg.ellipse(x, y, x+1, y+1, fillColor = "yellow",
            color = "black");
          end if;
        end if;
      end when;
    end for;
  end for;
end testGeom;
```

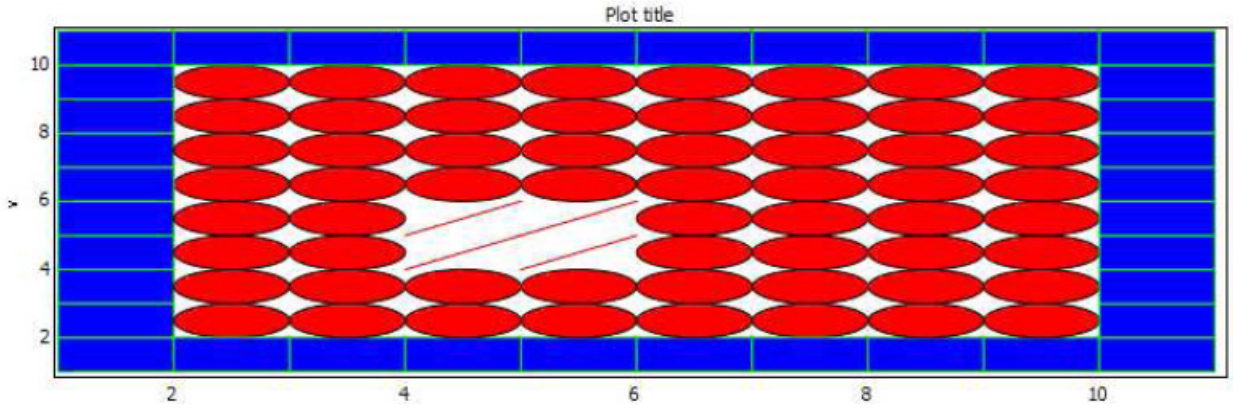


Figure 3-9. Programmable drawing of rectangles and ellipses.

3.2.6 Plotting of Table Data

Another way to visualize data provided by the graphics package is plotting of table data. This is done by using the command `pltpkg.plotTable`, which expects a matrix of Real values as a parameter. The rows of this matrix represent variable values. The first column is the time variable and the other columns contains values at these points in time. The names of the variables can be specified with the argument `variableNames`, which is a String list. The following model demonstrates how this command can be used.

```
model table
  protected
  Boolean b;
algorithm
  b := pltpkg.pltTable([ 0, 0.95, 0.92, 20, 25;
    10, 0.94, 0.92, 23, 28;
    20, 0.94, 0.91, 32, 35;
    30, 0.93, 0.90, 43, 46] );
end table;
```

The result is shown in Figure 3-10 below.

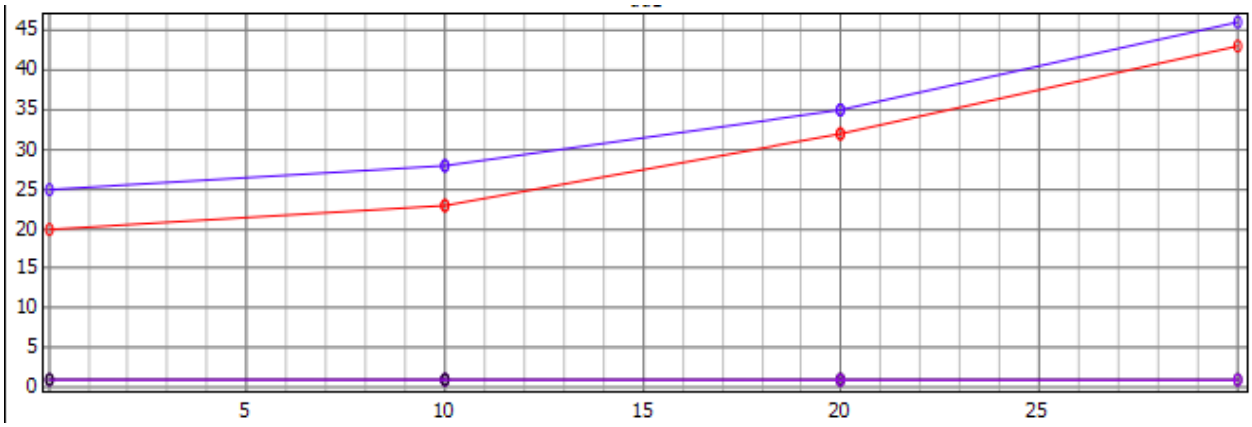


Figure 3-10. Plotting of table data.

3.3 Java-based PtPlot 2D plotting

The plot functionality in OpenModelica 1.4.4 and earlier was based on PtPlot (Lee, 2006), a Java-based plot package produced within the Ptolemy project. To plot one uses plot commands within input cells which it evaluates. Available plotting commands which call Java-based plotting are as follows, still available but renamed with a suffix 2:

```
// normal one variable plotting, time on the X axis
plot2( variable );
// normal multiple variable plotting, time on the X axis
plot2( {variable1, variable2, variable3, … variableN} );

// to plot dependent values
plotParametric2( variableX, variableY );
```

For example:

```
simulate(HelloWorld, startTime=9, stopTime=4);
plot(x);
```

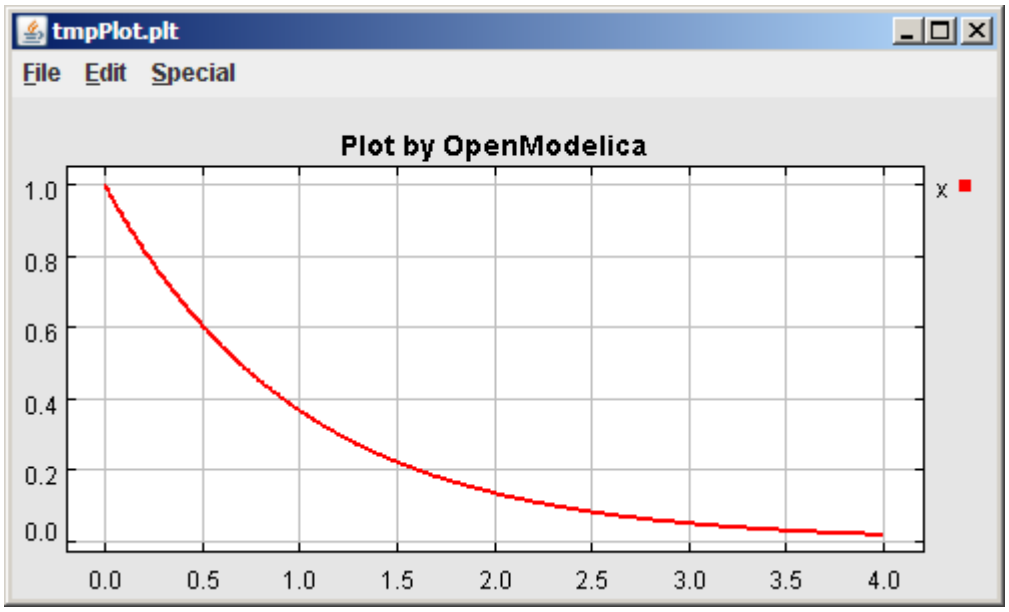


Figure 3-11. Java-based PtPlot plot window.

3.4 3D Animation

There are two main approaches to add 3D graphics information to Modelica objects:

- Graphical annotations
- Graphical objects

Both of these approaches were investigated, but the second was finally chosen.

3.4.1 Object Based Visualization

Since one important goal of this work is to come up with a system for visualization that might be used for simulations done with the Modelica MultiBody library (Otter, 2008), it follows that much can be learned from investigating currently available solutions. There are commercial software packages available that can visualize MultiBody simulations.

The MultiBody package is well suited for visualization. Entities in a MultiBody simulation correspond to physical entities in a real world and as such have many of the properties needed to correctly display them within a visualization of the simulation, such as position and rotation. Other properties such as color and shape can easily be added as properties or be decided based on the object type.

Instead of using annotations to encode information about how a certain object is supposed to look when visualized, object based visualization creates additional Modelica objects of a predetermined type that can be known to the client actually doing the visualization. These objects contain variables such as position, rotation and size that can be connected to the simulated variables using ordinary Modelica equations. When asked to visualize a model, the OpenModelica compiler can find variables in the model that are in the visualization package and only send only those datasets over to the client doing the visualization, in this case OMNotebook.

Taking inspiration from the MultiBody library, a small package has been designed that provides a minimal set of classes that can be connected to variables in the simulation. It is created as a Modelica package and can be included in the Modelica Library. The package is called `SimpleVisual`, and consists of a small hierarchy of classes that in increasing detail can describe properties of a visualized object. It is implemented on top of the Qt graphics package called Coin3D (Coin3D, 2008). More information is available in (Magnusson, 2008). A comprehensive earlier work on integrating and generating 3D graphics from Modelica models is reported in (Engelson, 2000).

This section gives a short introduction to how the `SimpleVisual` package is used.

3.4.2 BouncingBall

The bouncing ball model is a simple example to the Modelica language. Adding visualization of the bouncing ball using the SimpleVisual package is very straightforward.

```
model BouncingBall
  parameter Real e=0.7 "coefficient of restitution";
  parameter Real g=9.81 "gravity acceleration";
  Real h(start=10) "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;
```

To run a simulation of the bouncing ball, create a new InputCell and call the simulate command. The simulate command takes a model, start time, and an end time as arguments.

```
simulate(BouncingBall, startTime = 0, stopTime = 5);
```

3.4.2.1 Adding Visualization

The bouncing ball will be simulated with a red sphere. We will let the variable h control the y position of the sphere. Since the ball has a size and the model describes the bouncing movement of a point, we will use that size to translate the visualization slightly upwards. First, we must import the `SimpleVisual` package and create an object to visualize. That is done by adding a few lines to the beginning of the `BouncingBall` model, which we rename to `BouncingBall3D` to emphasize that we have made some changes:

```
model BouncingBall3D
import SimpleVisual.*
SimpleVisual.PositionSize ball "color=red;shape=sphere;";
...
```

The string "color=red;" is used to set the color parameter of the object and the shape parameter controls how we will display this object in the visualization.

The next step is to connect the position of the ball object to the simulation. Since Modelica is an equation based language, we must have the same number of variables as equations in the model. This

means that even though the only aspect of the ball that is really interesting is its y-position, each variable in the ball object must be assigned to an equation. Setting a variable to be constant zero is a valid equation. The SimpleVisual library contains a number of generic objects which gives the user an increasing amount of control.

```
SimpleVisual.Position
SimpleVisual.PositionSize
SimpleVisual.PositionRotation
SimpleVisual.PositionRotationSize
SimpleVisual.PositionRotationSizeO_set
```

Since we are really only interested in the position of the ball, we could use `SimpleVisual`.Position, but to make it a little bit more interesting we use `SimpleVisual.PositionSize` and make the ball a little bigger.

```
obj.size[1]=5;  obj.size[2]=5;  obj.size[3]=5;
obj.frame_a[1]=0;  obj.frame_a[2]=h+obj.size[2]/2;  obj.frame_a[3]=0;
```

A `SimpleVisual.PositionSize` object has two properties; `size` and `frame_a`. All are three dimensional real numbers, or Real[3] in Modelica.

- `size` controls the size of the visual representation of the object.
- `frame_a` contains the position of the object.

3.4.2.2 Running the Simulation and Starting Visualization

To be able to simulate the model with the added visualization, OpenModelica must load the SimpleVisual package.

```
loadLibrary(SimpleVisual)
```

Now, call simulate once more. This time the simulation will generate values for the added SimpleVisual object that can be read by the visualization in OMNotebook.

```
simulate(BouncingBall3D, start=0, end=5s);
```

To display the visualization, create an input cell and call the visualize in the input part of the cell.

```
visualize(BouncingBall3D);
```

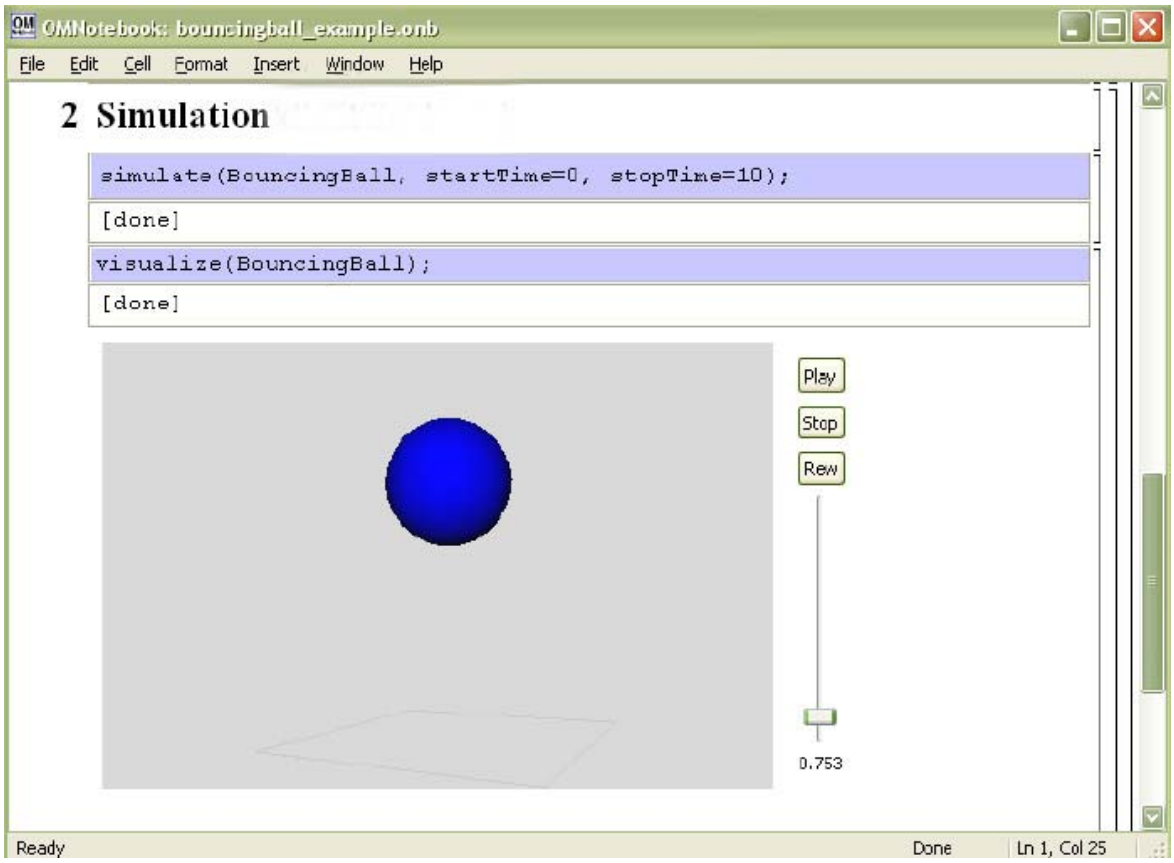


Figure 3-12. 3D animation of the bouncing ball model.

3.4.3 Pendulum 3D Example

This example explores a slightly more complex scenario where the visualization uses all the properties of a SimpleVisual object. The model used is a simple ideal 2D pendulum, not modeling properties like friction, air resistance etc.

```
class MyPendulum3D "Planar Pendulum"
constant Real PI=3.141592653589793;
parameter Real m=1, g=9.81, L=5;
Real F;
Real x(start=5),y(start=0);
Real vx,vy;
equation
  m*der(vx)=-(x/L)*F;
  m*der(vy)=-(y/L)*F-m*g;
  der(x)=vx;
  der(y)=vy;
  x^2+y^2=L^2;
end MyPendulum3D;
```

Start by identifying the variables in the model that will be needed to create a visual representation of the simulation.

- Real x and Real y hold the current position of the pendulum.
- Real L is a parameter which holds the length of the pendulum.

3.4.3.1 Adding the Visualization

As before, to be able to use the SimpleVisual package we must import it.

```
class MyPendulum3D "Planar Pendulum"
import Modelica.SimpleVisual;
...
```

Adding a sphere to represent the weight of the pendulum is done in the same way the BouncingBall was visualized. The variables x and y hold the position.

```
...
Real vx,vy;
SimpleVisual.PositionSize ball "color=red;shape=ball;";
equation
  ball.size[1]=1.5;  ball.size[2]=1.5;  ball.size[3]=1.5;
  ball.frame_a[1]=x;  ball.frame_a[2]=y;  ball.frame_a[3]=0;
  m*der(vx)=-(x/L)*F;
...
```

The next step is to create a visualization of the "thread" that holds the pendulum. It will be represented by a small elongated cube connected to the ball in one end and in the fixed center of the pendulum movement. We will want the object to rotate with the pendulum motion so create a `SimpleVisual.PositionRotationSize` object.

```
SimpleVisual.PositionRotationSize thread "shape=cube;"
```

To specify the rotation of an object, the visualization package uses two points. One is the position of the object, frame_a, that has been demonstrated earlier. The other position, `frame_b`, is interpreted as the end point of a vector from frame_a. This vector is used as the new up direction for the object. In this example, defining `frame_b` is simple. The cube that represents the thread will always be pointing to (0, 0, 0). We already know the length of the thread from the parameter L.

```
thread.size[1]=0.05;  thread.size[2]=L;  thread.size[3]=0.05;
thread.frame_a[1]=x;  thread.frame_a[2]=y;  thread.frame_a[3]=0;
thread.frame_b[1]=0;  thread.frame_b[2]=0;  thread.frame_b[3]=0;
```

Running this simulation and starting the visualization, we notice that everything is not quite right. The thread is centered around the pendulum. We could calculate a new position by translating the x and y coordinates along the rotation vector, but there is a better way. Change the object type to `SimpleVisual.PositionRotationSizeOffset`. The offset parameter is a translation within the local coordinate system of the object. To shift the center of the object to be at the bottom of the thread we add an offset of L/2 to the y component of offset.

```
thread.size[1]=0.05;  thread.size[2]=L;  thread.size[3]=0.05;
thread.frame_a[1]=x;  thread.frame_a[2]=y;  thread.frame_a[3]=0;
thread.frame_b[1]=0;  thread.frame_b[2]=0;  thread.frame_b[3]=0;
thread.offset[1]=0;  thread.offset[2]=L/2;  thread.offset[3]=0;
```

In the final model, a simple static fixture has also been added.

```
class MyPendulum3D "Planar Pendulum"
  import Modelica.SimpleVisual;
  constant Real PI=3.141592653589793;
  parameter Real m=1, g=9.81, L=5;
  Real F;
  Real x(start=5),y(start=0);
  Real vx,vy;
  SimpleVisual.PositionSize ball "color=red;shape=ball;";
  SimpleVisual.PositionSize fixture "shape=cube;";
```

```
  SimpleVisual.PositionRotationSizeOffset thread "shape=cube;";
equation
  fixture.size[1]=0.5; fixture.size[2]=0.1; fixture.size[3]=0.5;
  fixture.frame_a[1]=0; fixture.frame_a[2]=0; fixture.frame_a[3]=0;
  ball.size[1]=1.5; ball.size[2]=1.5; ball.size[3]=1.5;
  ball.frame_a[1]=x;  ball.frame_a[2]=y;  ball.frame_a[3]=0;
  thread.size[1]=0.05;  thread.size[2]=L; thread.size[3]=0.05;
  thread.frame_a[1]=x; thread.frame_a[2]=y; thread.frame_a[3]=0;
  thread.frame_b[1]=0; thread.frame_b[2]=0; thread.frame_b[3]=0;
  thread.offset[1]=0; thread.offset[2]=L/2; thread.offset[3]=0;
  m*der(vx)=-(x/L)*F;
  m*der(vy)=-(y/L)*F-m*g;
  der(x)=vx;
  der(y)=vy;
  x^2+y^2=L^2;
end MyPendulum3D;
```

We simulate and visualize as previously:

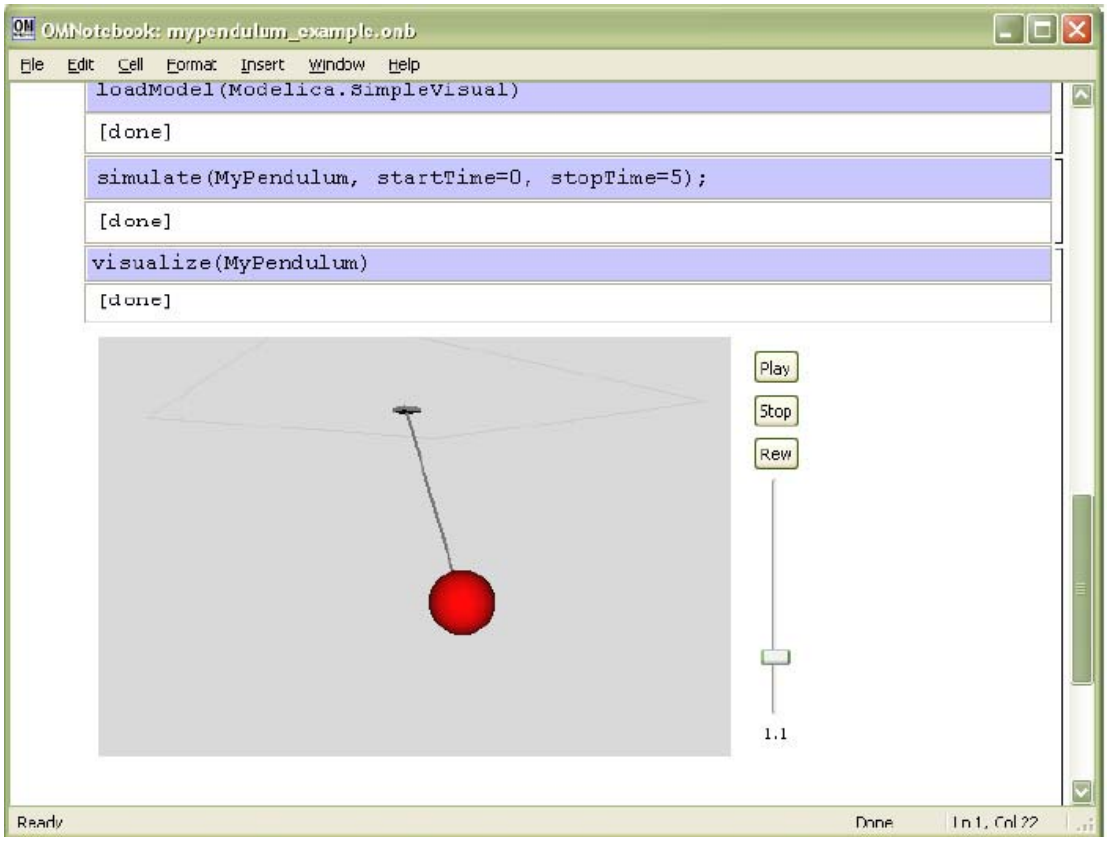


Figure 3-13. Visualization with animation of 3D pendulum.

3.5 References

Trolltech. Qt. http://www.trolltech.com/, accessed July 2007.

Coin3D. www.coin3d.org, accessed August 2008.

Henrik Eriksson. Advanced OpenModelica Plotting Package for Modelica. Master Thesis, LIU-IDA/LITH-EX-A-08/036-SE, Linköping University Electronic Press, www.ep.liu.se, June 22, 2008.

Henrik Magnusson. Integrated Generic 3D visualization of Modelica Models. Master Thesis, LIU-IDA/LITH-EX-A-08/035-SE, Linköping University Electronic Press, www.ep.liu.se, June 27, 2008.

Martin Otter. The Modelica MultiBody Library. http://www.modelica.org/libraries/Modelica, Modelica.Mechanics.MultiBody, accessed August 2008.

Vadim Engelson. Tools for Design, Interactive Simulation, and Visualization of Object-Oriented Models in Scientific Computing. Ph.D. Thesis. Linköping Studies in Science and Technology, Dissertation No. 627, http://www.ida.liu.se/~vaden/thesis/, 2000.

Edward Lee et al. The PtPlot package The Ptolemy Project. http://ptolemy.berkeley.edu/body.htm, accessed July 2007.

Chapter 4

OMNotebook with DrModelica and DrControl

This chapter covers the OpenModelica electronic notebook subsystem, called OMNotebook, together with the DrModelica tutoring system for teaching Modelica, and DrControl for teaching control together with Modelica. Both are using such notebooks.

4.1 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.

4.1.1 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 4-1below and Chapter 19 in (Fritzson 2004)

4.1.2 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 4-1.

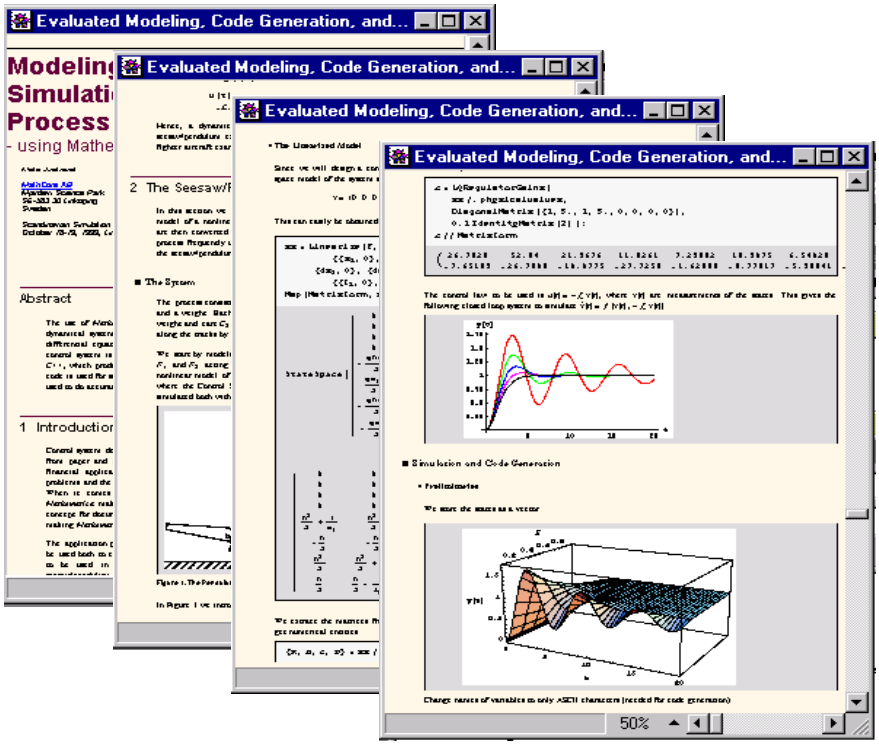


Figure 4-1. Examples of Mathematica notebooks in the MathModelica modeling and simulation environment.

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.

4.2 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 4-2).

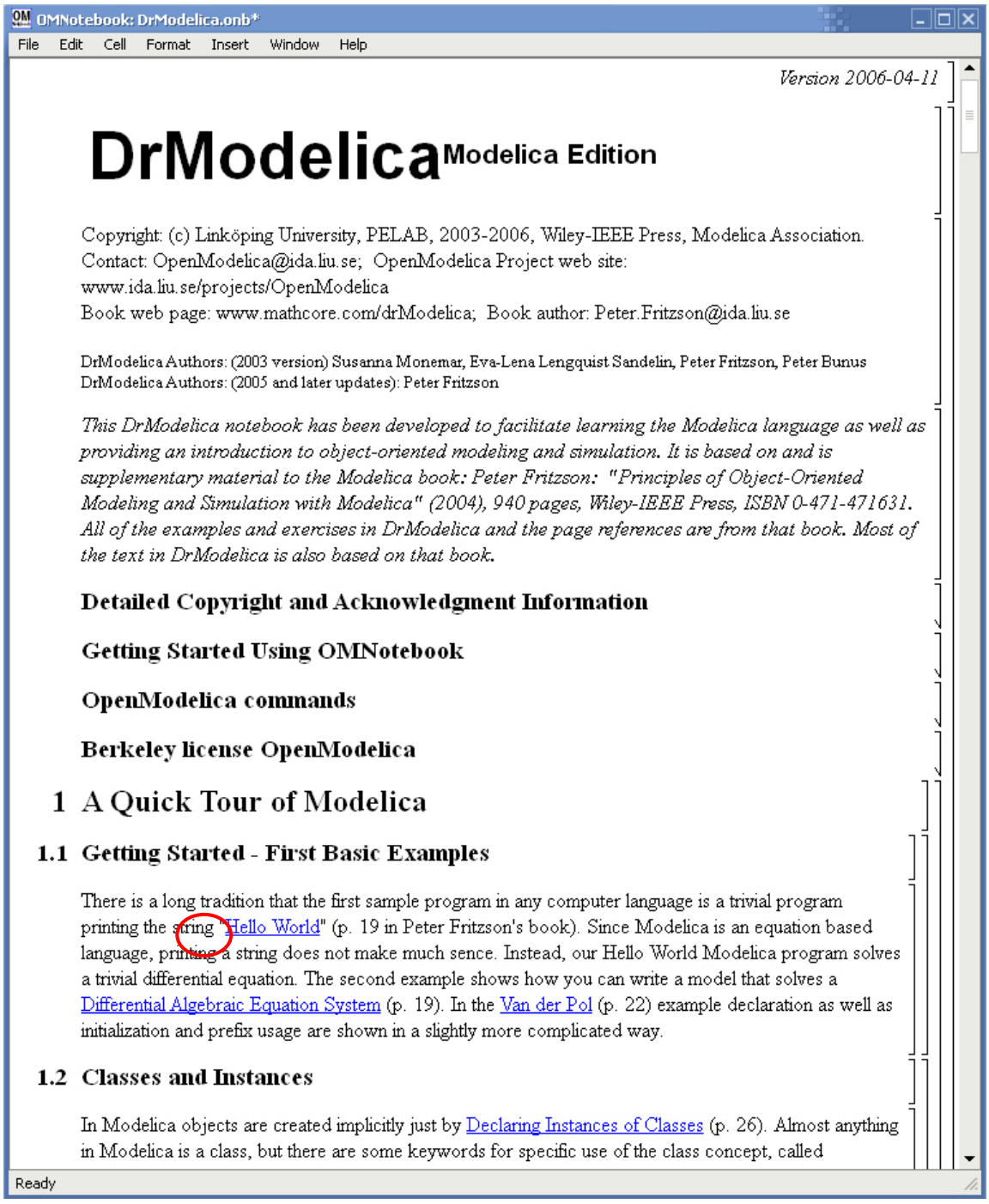


Figure 4-2. The front-page notebook of the OMNotebook version of the DrModelica tutoring system.

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

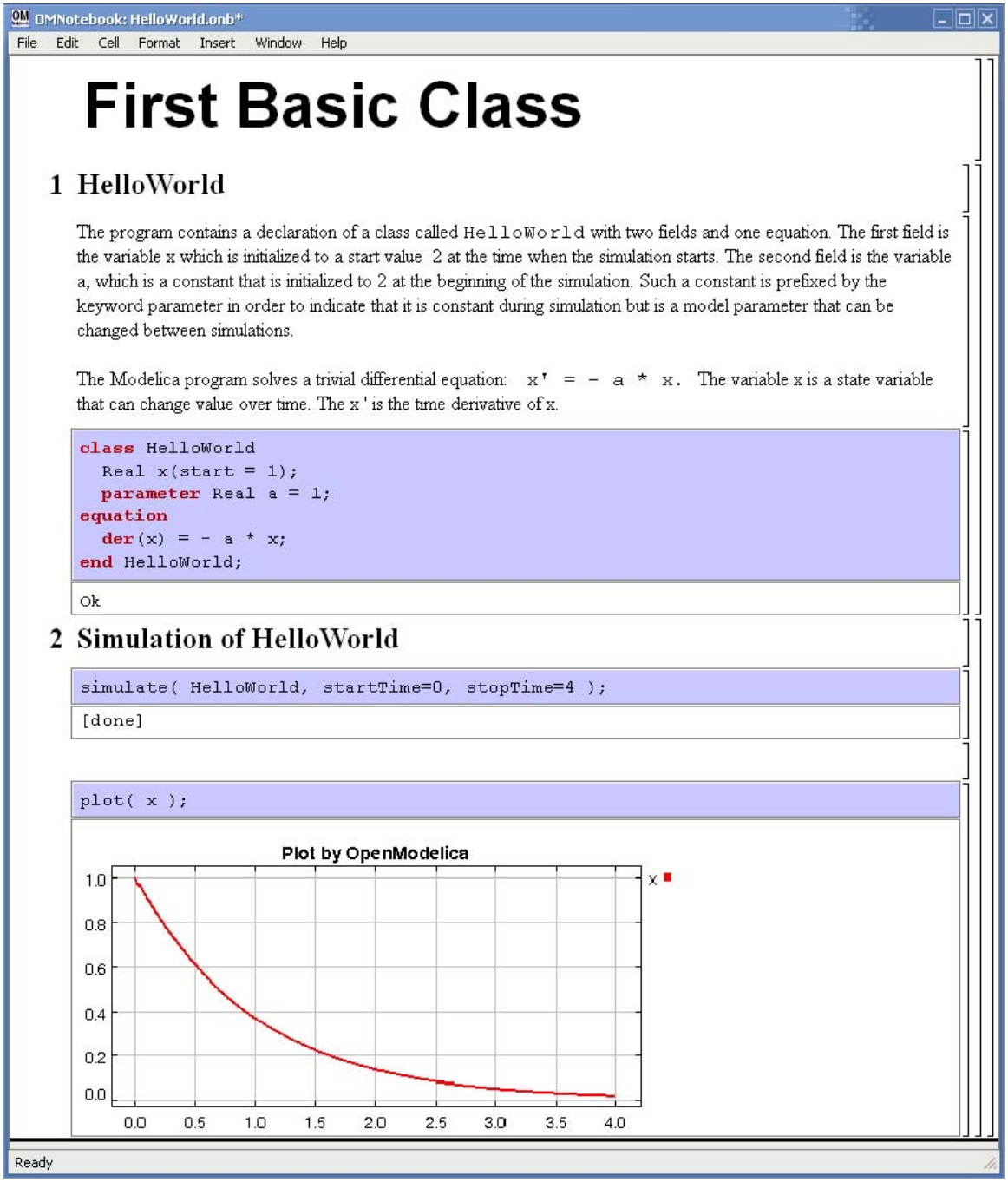


Figure 4-3. The **`HelloWorld`** class simulated and plotted using the OMNotebook version of DrModelica.

Now, let us consider that the link "*HelloWorld*" in DrModelica Section is clicked by the user. The new HelloWorld notebook (see Figure 4-3), 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.

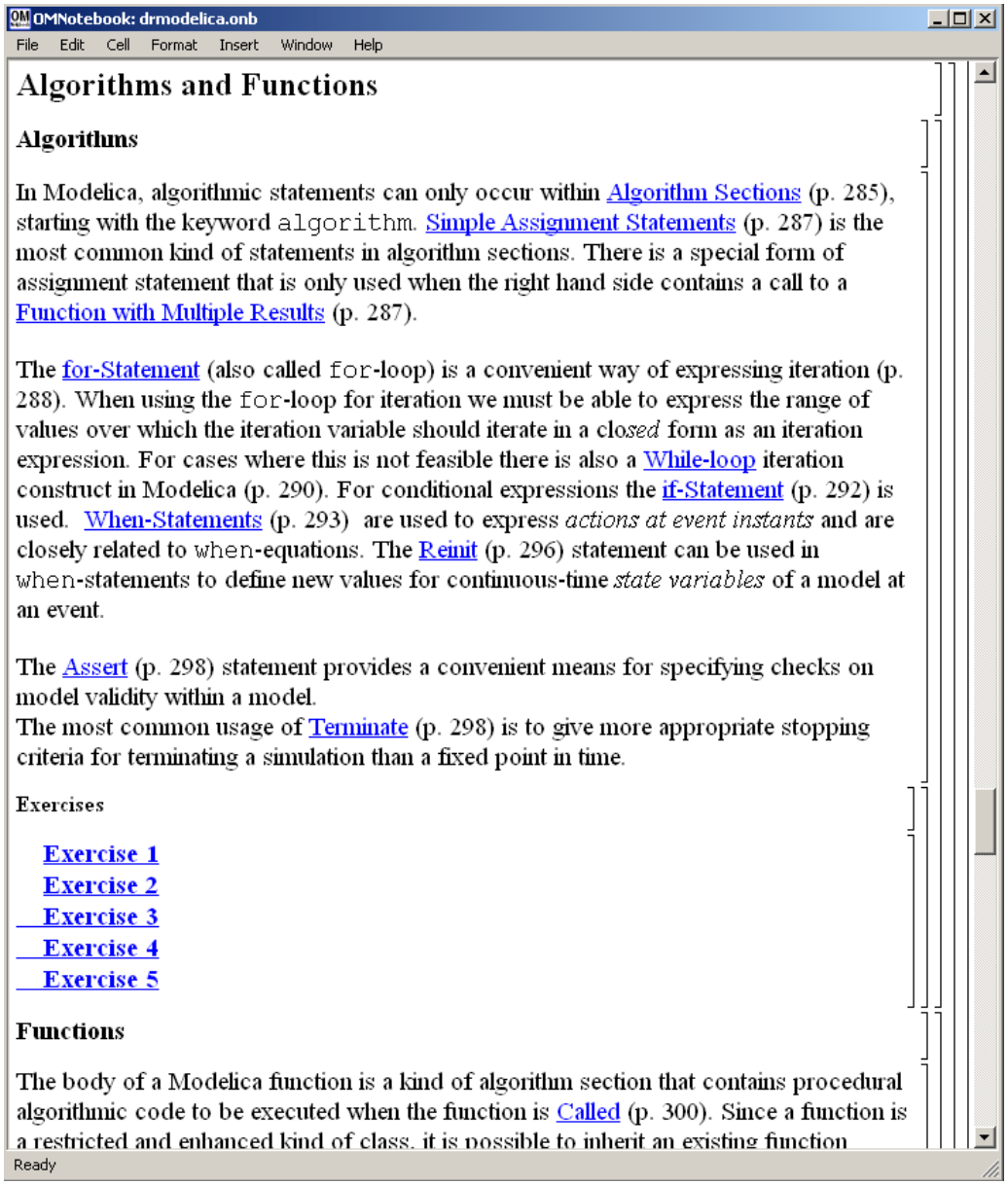


Figure 4-4. DrModelica Chapter on Algorithms and Functions in the main page of the OMNotebook version of DrModelica.

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

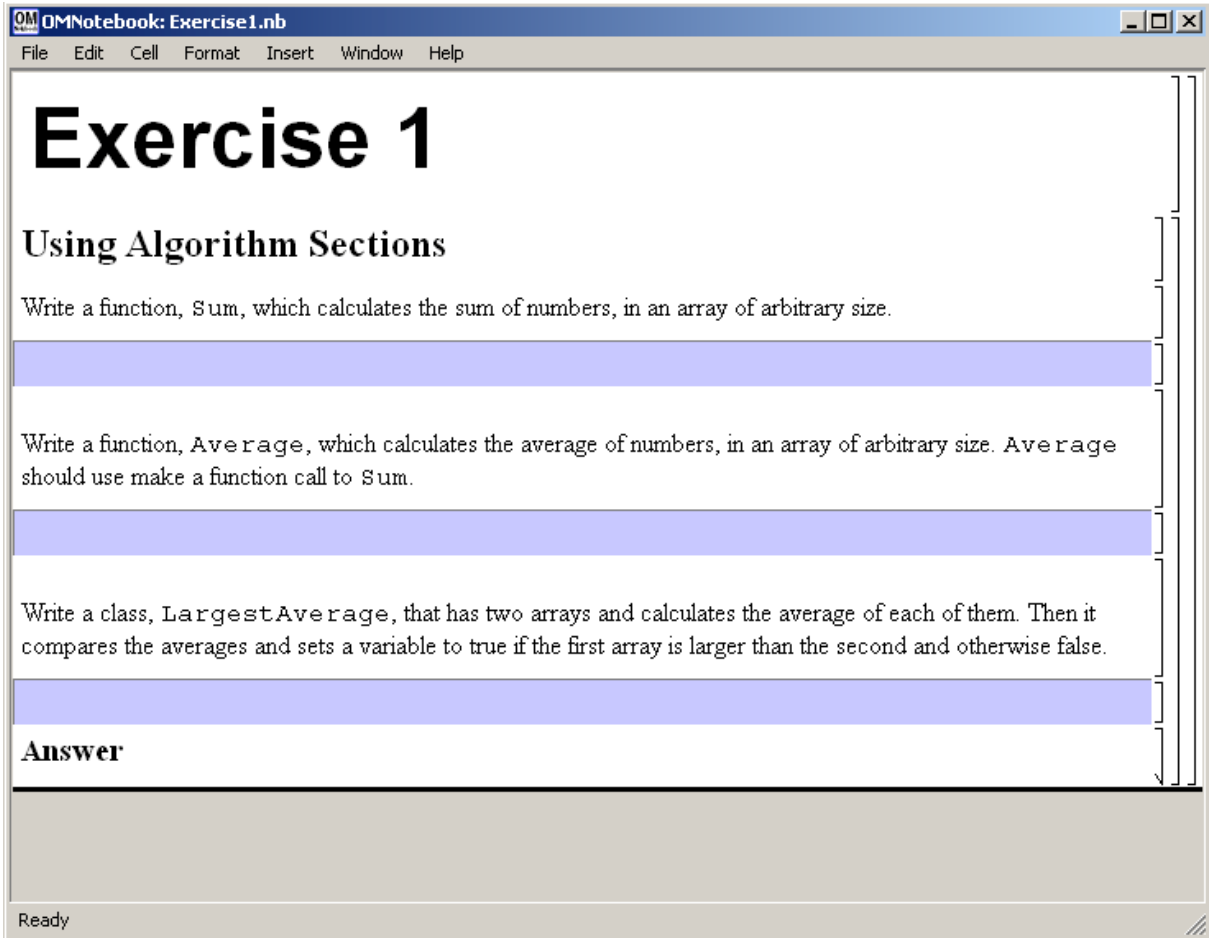


Figure 4-5. Exercise 1 in Chapter 9 of DrModelica

Exercise 1 from Chapter 9 is shown in . 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

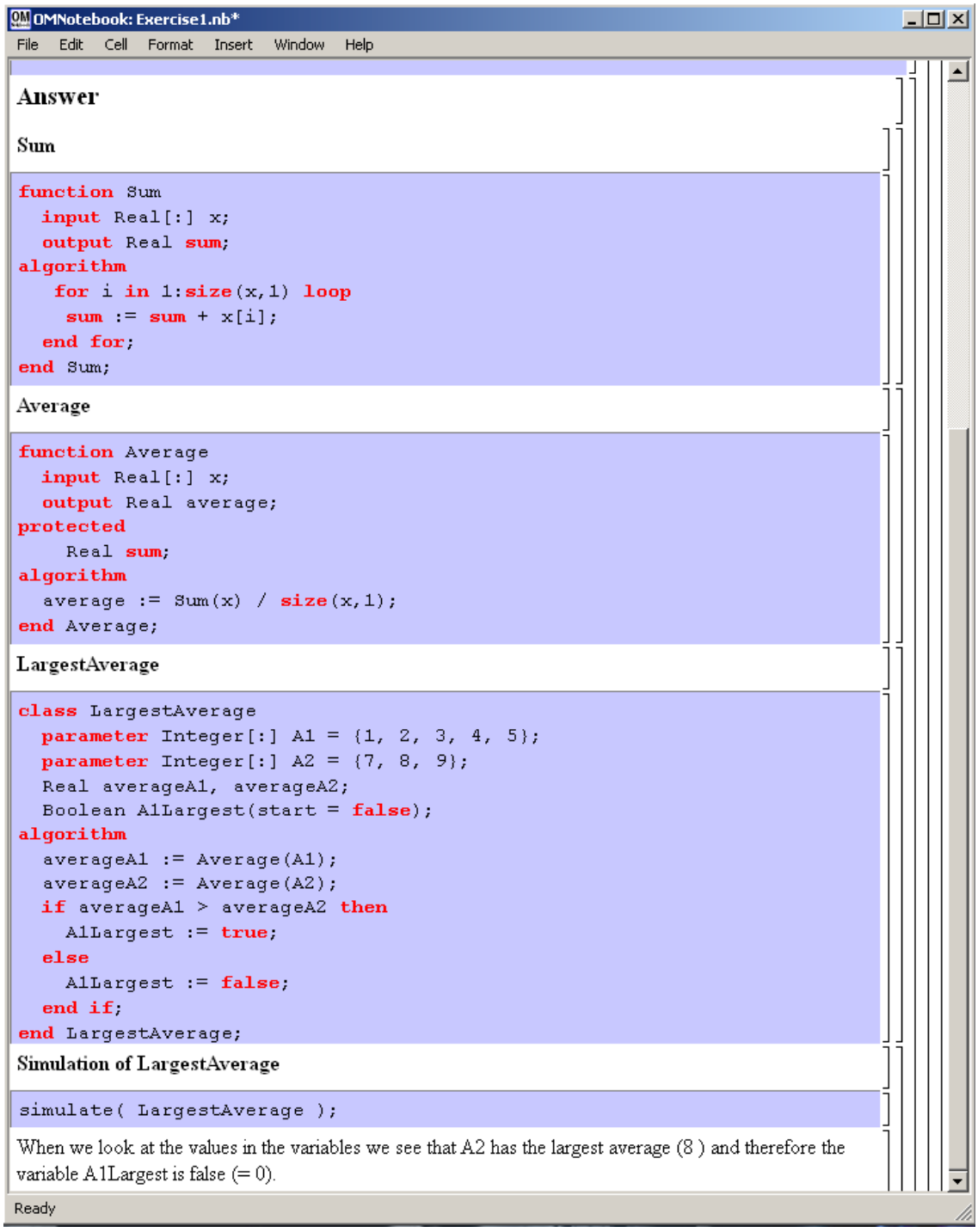


Figure 4-6. The answer section to Exercise 1 in Chapter 9 of DrModelica.

4.3 DrControl Tutorial for Teaching Control Theory

DrControl is an interactive OMNotebook document aimed at teaching control theory. It is included in the OpenModelica distribution and appears under the directory `OpenModelica1.8.0/share/omnotebook/drcontrol`.

The front-page of DrControl resembles a linked table of content that can be used as a navigation center. The content list contains topics like:

- Getting started
- The control problem in ordinary life
- Feedback loop
- Mathematical modeling
- Transfer function
- Stability
- Example of controlling a DC-motor
- Feedforward compensation
- State-space form
- State observation
- Closed loop control system.
- Reconstructed system
- Linear quadratic optimization
- Linearization

Each entry in this list leads to a new notebook page where either the theory is explained with Modelica examples or an exercise with a solution is provided to illustrate the background theory. Below we show a few sections of DrControl.

4.3.1 Feedback Loop

One of the basic concepts of control theory is using feedback loops either for neutralizing the disturbances from the surroundings or a desire for a smoother output.

In Figure 4-7, control of a simple car model is illustrated where the car velocity on a road is controlled, first with an open loop control, and then compared to a closed loop system with a feedback loop. The car has a mass m, velocity y, and aerodynamic coefficient α. The θ is the road slope, which in this case can be regarded as noise.

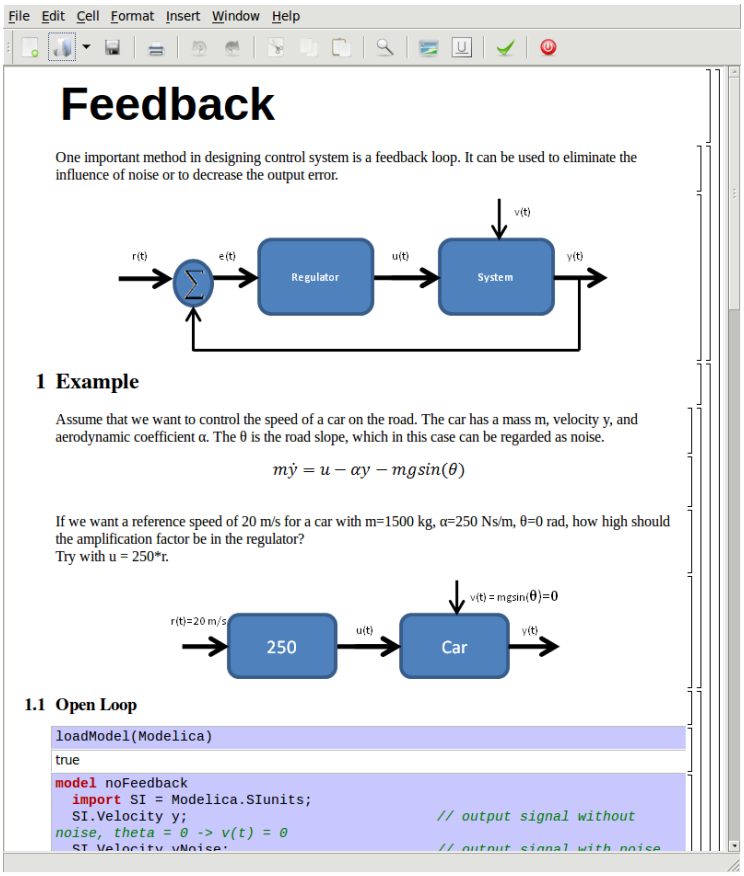


Figure 4-7. Feedback loop

Lets look at the Modelica model for the open loop controlled car:

$$m\dot{y} = u - \alpha y - mgsin(\theta)$$

```
model NoFeedback
  import SI = Modelica.SIunits;
  SI.Velocity y        "No noise";
  SI.Velocity yNoise   "With noise";
  parameter SI.Mass m = 1500;
  parameter Real alpha = 200;
  parameter SI. ngle theta = 5*3.14/180;
  parameter SI.Acceleration g = 9.82;
  SI.Force u;
  SI.Velocity r = 20 "Reference signal";
equation
  m*der(y)=u - alpha*y;
  m*der(yNoise)= u - alpha*yNoise –
     m*g*sin(theta);
  u = 250A*r;
end NoFeedback;
```

By applying a road slope angle different from zero the car velocity is influenced which can be regarded as noise in this model. The output signal in Figure 4-8 is stable but an overshoot can be observed compared to the reference signal. Naturally the overshoot is not desired and the student will in the next exercise learn how to get rid of this undesired behavior of the system.

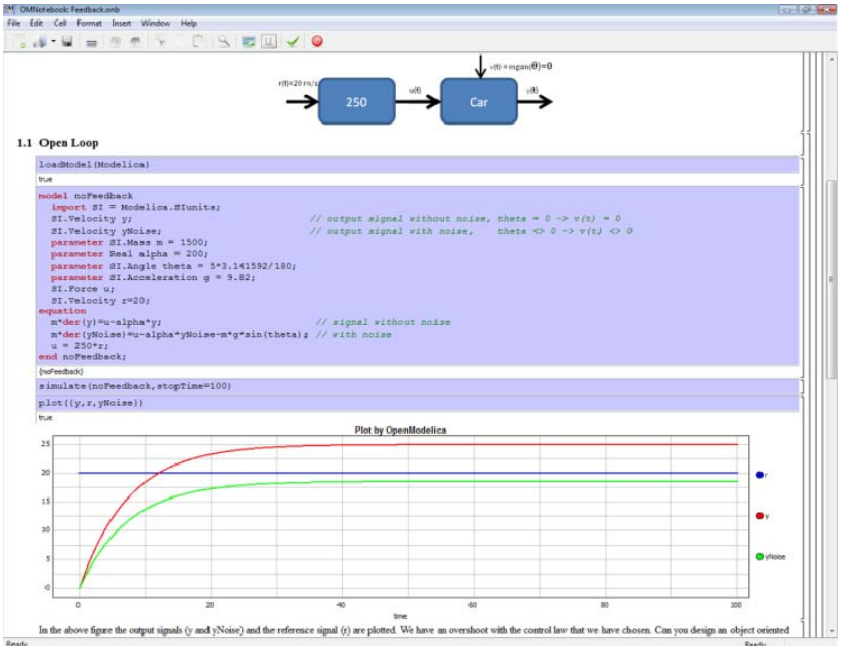


Figure 4-8. Open loop control example.

The closed car model with a proportional regulator is shown below:

$$u = K * (r - y)$$

```
model WithFeedback
  import SI = Modelica.SIunits;
  SI.Velocity y       "Output, No noise";
  SI.Velocity yNoise  "Output With noise";
  parameter SI.Mass m = 1500;
  parameter Real alpha = 250;
  parameter SI.Angle theta = 5*3.14/180;
  parameter SI.Acceleration g = 9.82;
  SI.Force u;
  SI.Force uNoise;
  SI.Velocity r = 20   "Reference signal";
equation
  m*der(y) = u - alpha*y;
  m*der(yNoise) = uNoise - alpha*yNois –
    m*g*sin(theta);
  u = 5000*(r - y);
  uNoise = 5000*(r - yNoise);
end WithFeedback;
```

By using the information about the current level of the output signal and re-tune the regulator the output quantity can be controlled towards the reference signal smoothly and without an overshoot, as shown in Figure 4-9.

In the above simple example the flat modeling approach was adopted since it was the fastest one to quickly obtain a working model. However, one could use the object oriented approach and encapsulate the car and regulator models in separate classes with the Modelica connector mechanism in between.

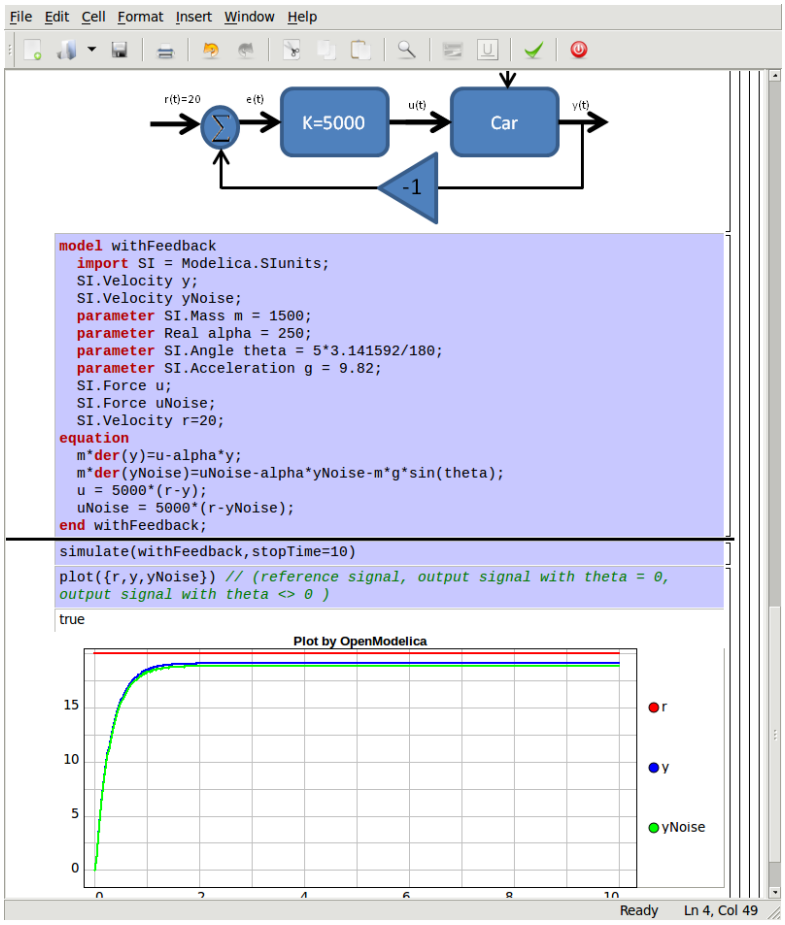


Figure 4-9. Closed loop control example.

4.3.2 Mathematical Modeling with Characteristic Equations

In most systems the relation between the inputs and outputs can be described by a linear differential equation. Tearing apart the solution of the differential equation into homogenous and particular parts is an important technique taught to the students in engineering courses, also illustrated in Figure 4-10.

$$\frac{d^n y}{dt^n} + a_1 \frac{d^{n-1} y}{dt^{n-1}} + \ldots + a_n y = b_0 \frac{d^m u}{dt^m} + \cdots + b_{m-1} \frac{du}{dt} + b_m u$$

Now let us examine a second order system:

$$\ddot{y} + a_1 \dot{y} + a_2 y = 1$$

```
model NegRoots
  Real y;
  Real der_y;
  parameter Real a1 = 3;
  parameter Real a2 = 2;
equation
  der_y = der(y);
  der(der_y) + a1*der_y + a2*y = 1;
end NegRoots;
```

Choosing different values for a_1 and a_2 leads to different behavior as shown in Figure 4-11 and Figure 4-12.

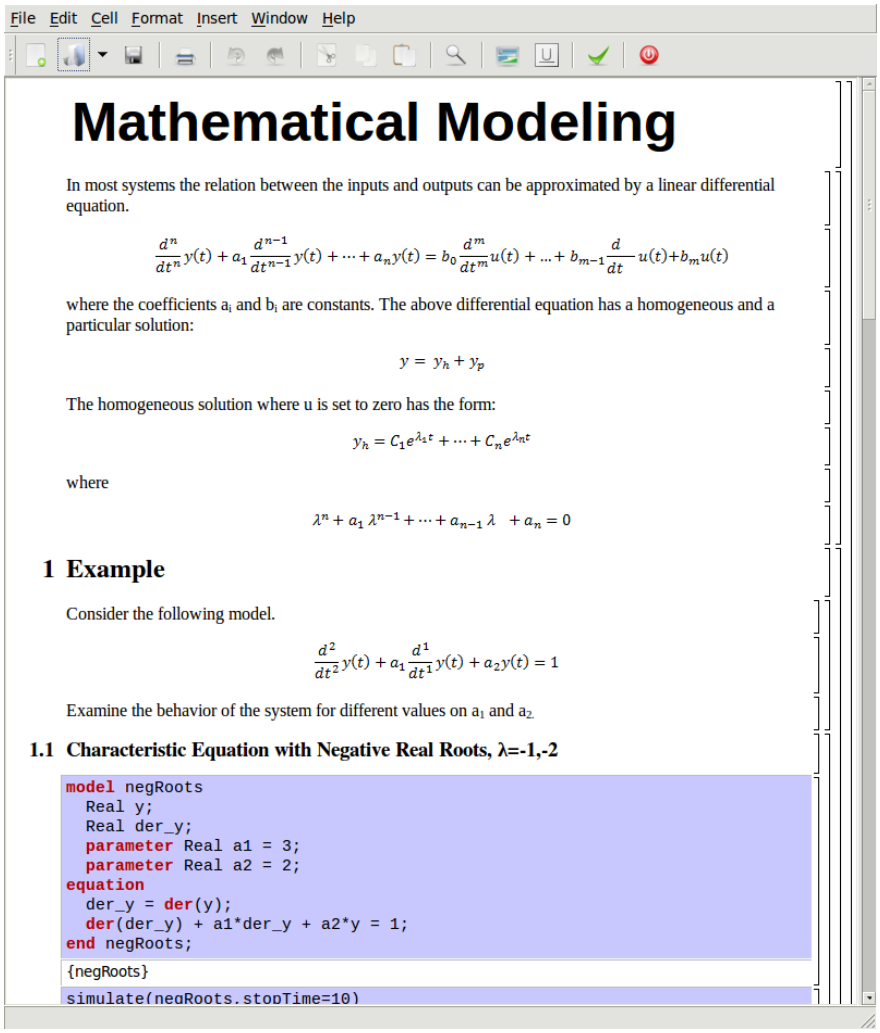


Figure 4-10. Mathematical modeling with characteristic equation.

In the first example the values of a_1 and a_2 are chosen in such way that the characteristic equation has negative real roots and thereby a stable output response, see Figure 4-11.

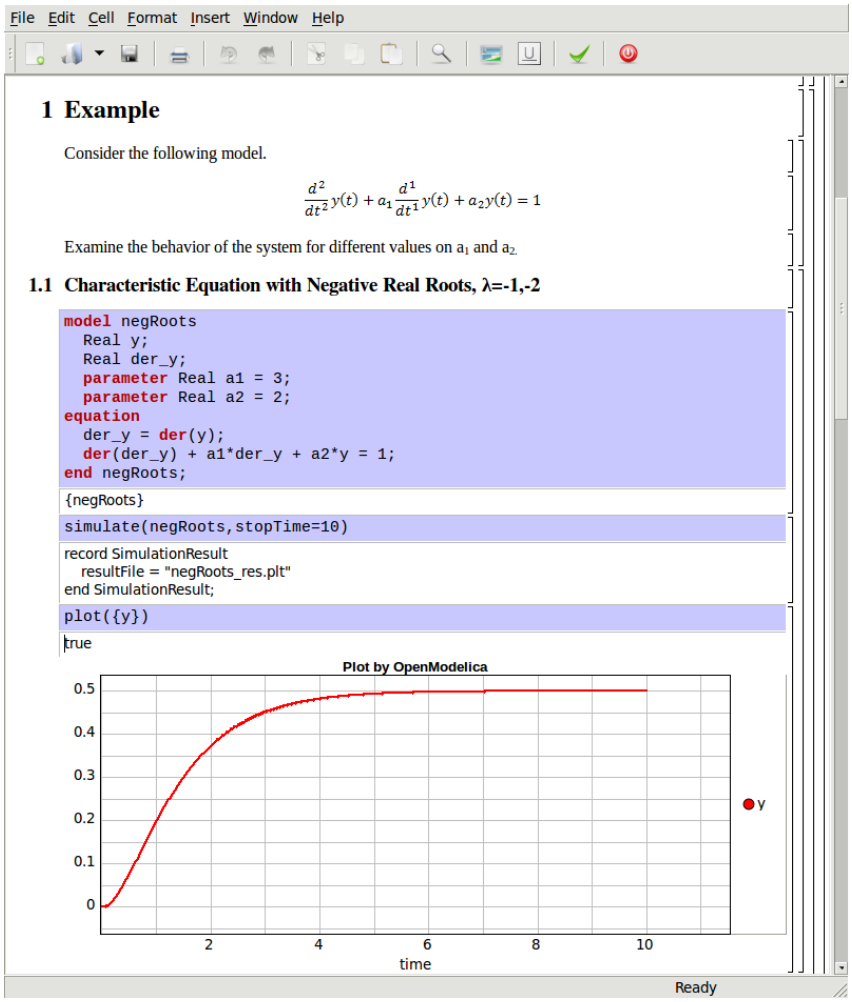


Figure 4-11. Characteristic eq. with real negative roots.

The importance of the sign of the roots in the characteristic equation is illustrated in Figure 4-11 and Figure 4-12, e.g., a stable system with negative real roots and an unstable system with positive imaginary roots resulting in oscillations.

```
model NegRoots
  Real y;
  Real der_y;
  parameter Real a1 = -2;
  parameter Real a2 = 10;
equation
  der_y = der(y);
  der(der_y) + a1*der_y + a2*y = 1;
end NegRoots;
```

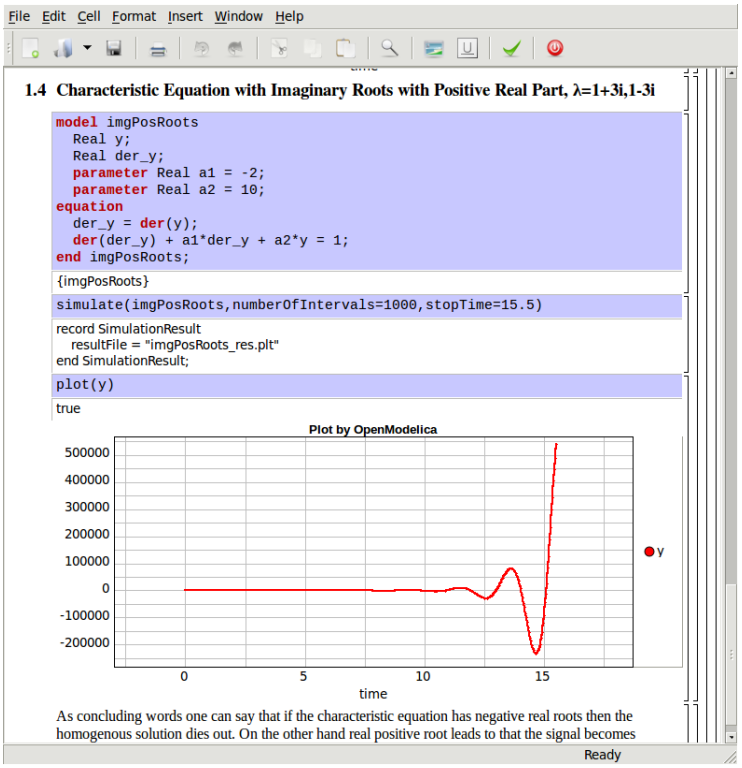


Figure 4-12. Characteristic eq. with positive imaginary roots.

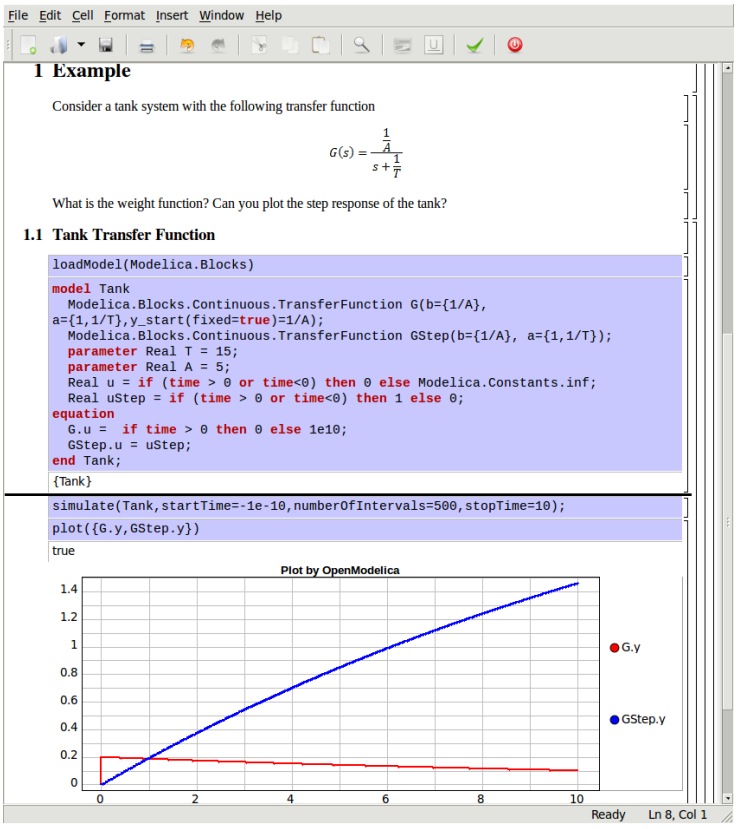


Figure 4-13. Step and pulse (weight function) response.

The theory and application of Kalman filters is also explained in the interactive course material.

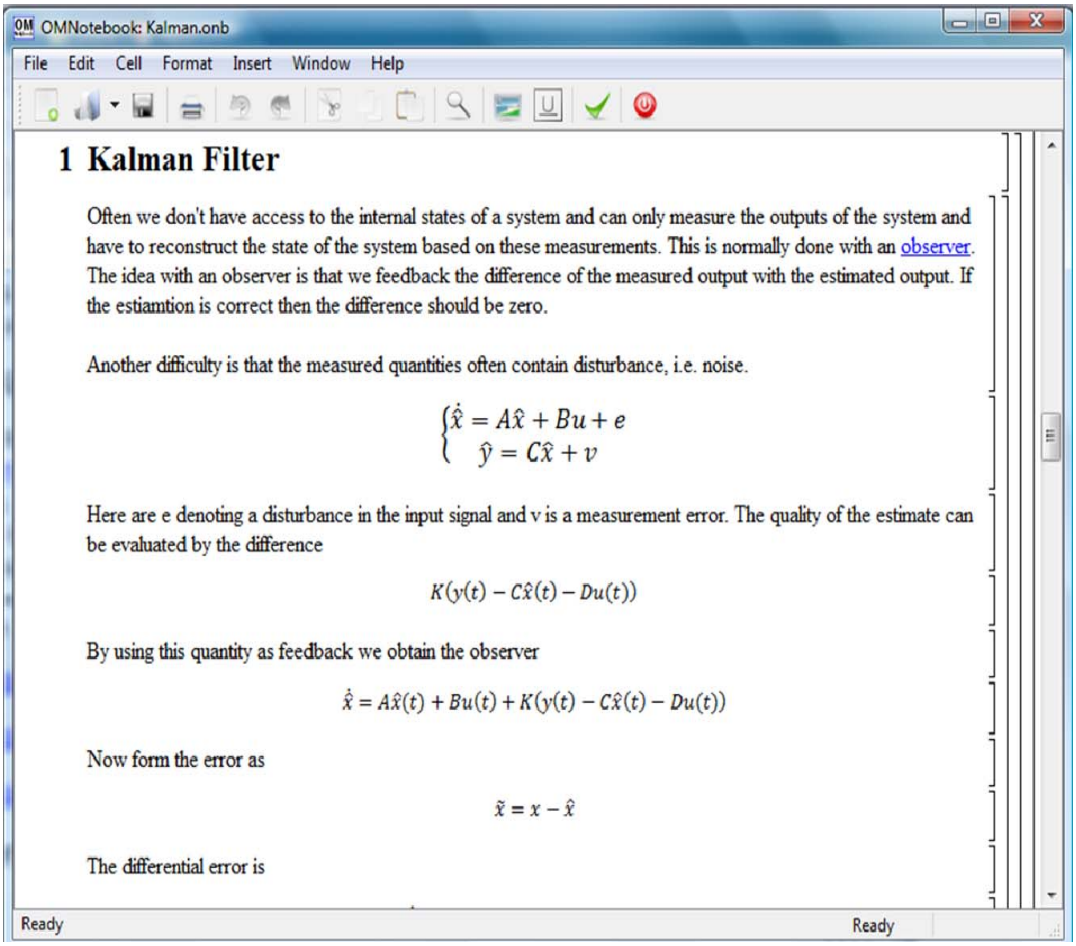


Figure 4-14. Theory background about Kalman filter.

In reality noise is present in almost every physical system under study and therefore the concept of noise is also introduced in the course material, which is purely Modelica based.

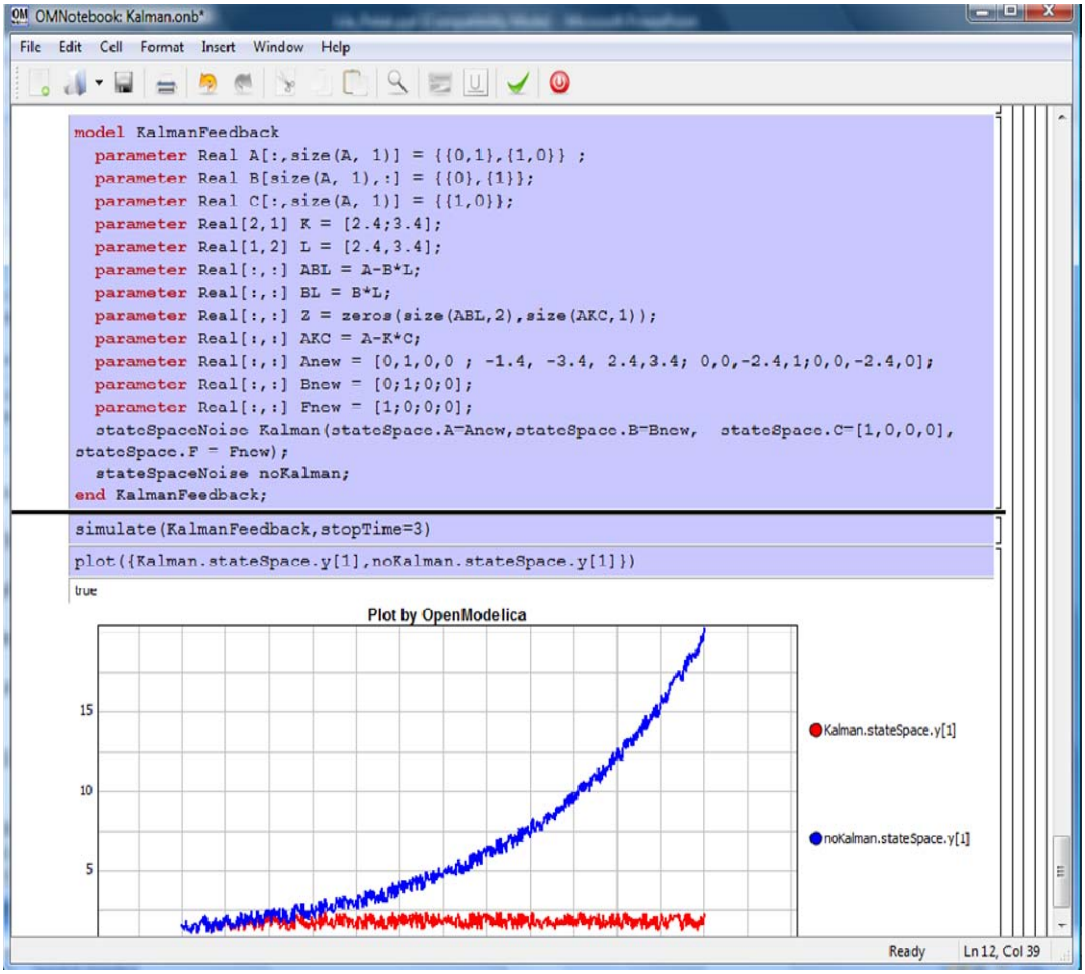


Figure 4-15. Comparison of a noisy system with feedback link in DrControl.

4.4 OpenModelica Notebook Commands

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

4.4.1 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.

4.4.2 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. It consists of two parts. The main cellcursor is basically just a thin black horizontal line below the cell with input focus. 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. The dynamic cellcursor is a short blinking horisontal line. To make this visible, you must click once more on the main cellcursor (the long horizontal line). NOTE: In order to paste cells at the cellcursor, the *dynamic cellcursor must be made active* by clicking on the main cellcursor (the horisontal line).

4.4.3 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 positioin 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 done 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.

4.4.4 File Menu

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

- *Create a new notebo*ok – 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 Crtl+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.

4.4.5 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. Pasteing 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.
- *Find* – Find text string in the current notebook, with the options match full word, match cell, search within closed cells. Short command Ctrl+F.
- *Replace* – Find and replace text string in the current notebook, with the options match full word, match cell, search+replace within closed cells. Short command Ctrl+H.
- *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.

4.4.6 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* – Spliting a cell is done with the menu item `Cell->Split cell` or the key combination Ctrl+Shift+P. The cell is splited 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.

4.4.7 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)

4.4.8 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.

4.4.9 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.

4.4.10 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.

4.4.11 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.

4.5 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, LITH-IDA-EX--06/057—SE, Dept. Computer and Information Science, Linköping University, Sweden, September 4, 2006.

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 3.0, Sept 2007. http://www.modelica.org.

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

Chapter 5

Interactive Simulation

In order to offer a user-interactive and time synchronous simulation, OpenModelica has an additional subsystem to fulfill general requirements on such simulations.

This module is part of the simulation runtime core and is called “OpenModelica Interactive” (OMI). OMI will result in an executable simulation application, such as the non interactive simulation. The executable file will be generated by the OMC, which contains the full Modelica model as C/C++ code with all required equations, conditions and different solvers to simulate a whole system or a single system component. This executable file offers a non-interactive and an interactive simulation runtime.

The following are some general functionalities of an interactive simulation runtime:

- The user will be able to stimulate the system during a running system simulation and to observe its’ reaction immediately.
- Simulation runtime behavior will be controllable and adaptable to offer an interaction with a user.
- A user will receive simulation results during a simulation synchronous to the real-time. Since network process time and some other factors like scheduling of processes from the operation system this is not given at any time.
- In order to offer a stable simulation, a runtime will inform a user interface of errors and consequential simulation aborts.
- Simulation results will not under-run or exceed a tolerance compared to a thoroughly reliable value, for a correct simulation.
- Communication between a simulation runtime and a user interface will use a well defined interface and be base on a common technology, in this case network communication.

Note that OMI is available in an easy-to-use way from OMEdit, see Section 2.7.

5.1 OpenModelica Interactive

5.1.1 Interactively Changeable Parameters

An important modification/addition to the semantics of the Modelica language during interactive simulation is the fact that parameters are changeable while simulating interactively using OMI. All properties using the prefix “parameter” can be changed during an interactive simulation. The fully qualified name is used as a unique identifier, so a parameter value can be found and changed regardless of its hierarchical position in the model.

5.1.2 OpenModelica Interactive Components description

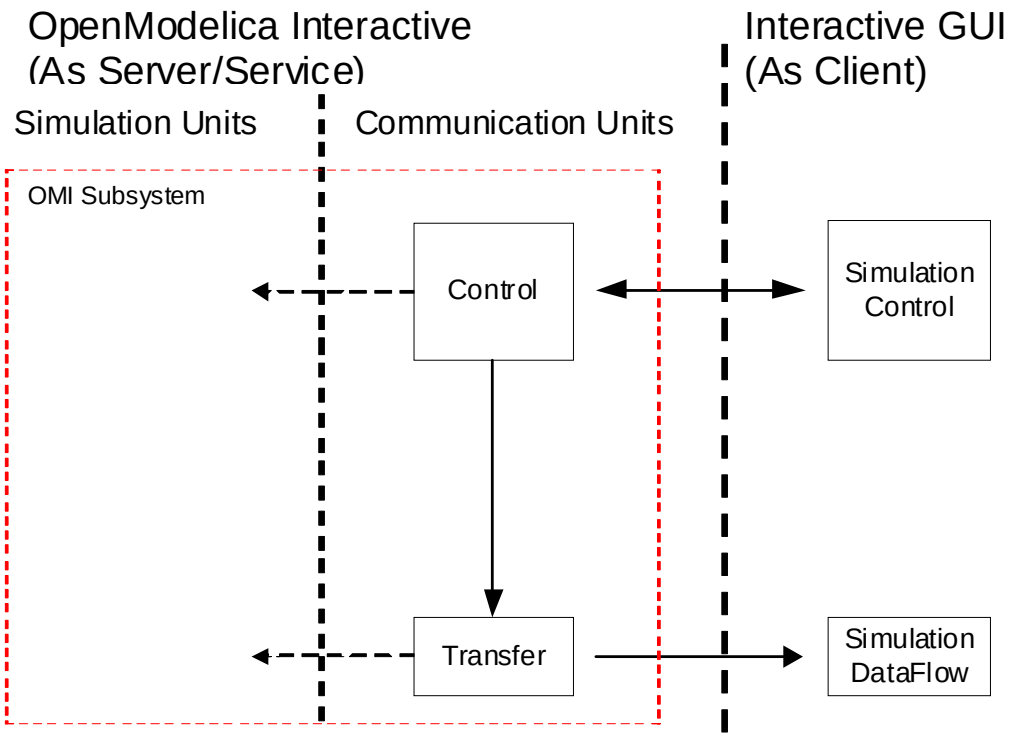


Figure 5-1. OpenModelica interactive communication architecture..

The OpenModelica Interactive subsystem is also separated into different modules, following are important for the user to communicate with:

- Control: The "Control" module is the interface between OMI and a UI. It is implemented as a single thread to support parallel tasks and independent reactivity. As the main controlling and communication instance at simulation initialization phase and while simulation is running it manages simulation properties and also behavior. A client can permanently send operations as messages to the "Control" unit, it can react at any time to feedback from the other internal OMI components and it also sends messages to a client, for example error or status messages.
- Transfer: Similar to a consumer, the "Transfer" thread tries to get simulation results from a result manager and sends them to the UI immediately after starting a simulation. If the communication takes longer than a calculation step, it is also possible to create more than one consumer. The "Transfer" uses a property filter mask containing all property names whose result values are important for the UI. The UI must set this mask using the "setfilter" operation from chapter 2.1.3.2, otherwise the transfer sends only the actual simulation time. This is very useful for increasing the communication speed while sending results to the UI.

5.1.3 Communication Interface

The network communication technology "TCP/IPv4" (later IPv6) will be used to send and receive messages. Each system has its own server and client implementations to receive and send messages respectively. The Control and Transfer are the OMI components which are designated for a communication over TCP/IP.

5.1.4 Network configuration Settings

Name	Description	URL
Control Server	Waits for requests from the UI	By Default, waits for connection on: 127.0.0.1:10501
Control Client	Replies to the UI and sends other synchronization messages to it	By Default, tries to connect on: 127.0.0.1:10500
Transfer Client	Sends simulation results to a UI	By Default, tries to connect on: 127.0.0.1:10502

OMI server and client components: Communication behaviour and configuration by default

Name	Description	URL
Control Client	Requests to the OMI Control Server	By Default, tries to connect on: 127.0.0.1:10501
Control Server	Waits for information from the OMI Control Client	By Default, waits for connection on: 127.0.0.1:10500
Transfer Server	Waits for simulation results from the OMI Transfer Client	By Default, waits for connection on: 127.0.0.1:10502

UI server and client components: Suggested configuration by default

5.1.4.1 Operation Messages

To use messages parsing there is a need to specify a communications protocol.

A string message begins with a specified prefix and ends with a specified suffix.

The prefix describes the request type, for example an operation. Depending on the request type, some additional information and parameters can append on it. The suffix is to check if the message has been received correctly and if the sender has created it correctly. All parts should be separated with "#".

A sequence number is helpful to manage operation request and reply, a UI has to send a sequence number combined with an operation.

The following are all available message strings between a UI and the OMI system:

Request from UI to Control

UI Request	Description	OMI::Control Reply
start#SEQ#end	Starts or continues the simulation	done#SEQ#end
pause#SEQ#end	Pauses the running simulation	done#SEQ#end
stop#SEQ#end	Stops the running simulation and resets all values to the beginning	done#SEQ#end
shutdown#SEQ#end	Shuts the simulation down	done#SEQ#end
setfilter#SEQ# var1:var2# par1:par2# end	Sets the filter for variables and parameters which should send from OMI to the client UI	done#SEQ#end
useindex#SEQ#end	Uses indexes as attribute names. The index will be used at transmitting results to a client. This will cause much less data to transmit. (??Not implemented yet)	done#SEQ#end
setcontrolclienturl#SEQ# ip#port# end	Changes the IP and port of the Control Server. Otherwise the default configuration will be used.	done#SEQ#end
settransferclienturl#SEQ# ip#port# end	Changes the IP and port of the Control Server. Otherwise the default configuration will be used.	done#SEQ#end

changetime#SEQ#Tn#end	Changes the simulation time and goes back to a specific time step	done#SEQ#end
changevalue#SEQ#Tn# par1=2.3:par2=33.3# end	Changes the value of the appended parameters and stets the simulation time back to the point where the user clicked in the UI	done#SEQ#end
error#TYPE#end	Error handling not implemented yet	Error: *

Table 5-1 Available messages from a UI to OMI (Request-Reply)

Messages from Control to UI

OMI::Control	Description	UI
Error: MESSAGE	If an error occurs the OMI::Control generates an error messages and sends the messages with the prefix "Error:" to the UI (not implemented yet)	Up to the UI developers

Table 5-2 Available messages from OMI::Control to UI

Messages from Transfer to UI

OMI::Transfer	Description	UI
result#ID#Tn# var1=Val:var2=Val# par1=Val:par2=Val# end	Sends the simulation result for a time step Tn to the client UI, using the property names as identifier. Maybe a result ID is important to identify the results which are obsolete (not implemented yet).	None
result#ID#Tn# 1=Val:2=Val# 1=Val:2=Val# end	Sends the simulation result for a time step Tn to the client UI, using an index as identifier. This requires a convention about the used index mask. Transfer optimization. NOTE: Operation from UI needed, Mask creation using the standard array index is recommended. Maybe a result ID is important to identify the results which are obsolete (not implemented yet).	None

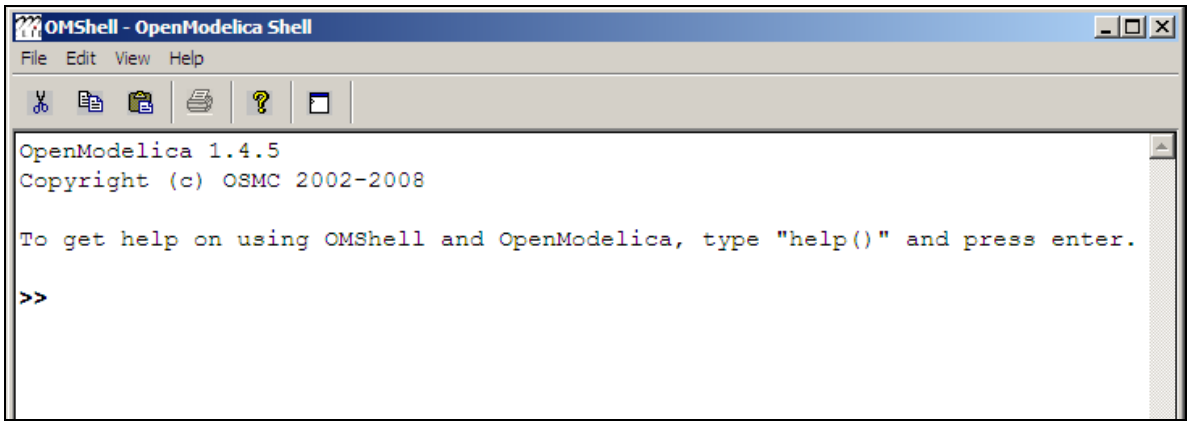
Table 5-3 Available messages from OMI::Transfer to UI

5.1.5 Interactive Simulation general Procedure

Note that OMI is available in an easy-to-use way from OMEdit, see Section 2.7, as an alternative to the procedure described below.

5.1.5.1 Initialize an Interactive Simulation Session

Start the OpenModelica Shell or OMNotebook which is available in the start menu as OpenModelica->OpenModelica Shell or OpenModelica->OMNotebook.



1. Load a model or file.
   Optional: You can check if your model or file has been loaded correctly with the operation “list()”
2. Build the model using the operation “buildModel(…)” with the following parameters:
   a) *Model main class name*: Name of the main class of your model.
   b) *numberOfIntervals*: Number of output values in an interval of one second. For Example: “numberOfIntervall=5” means that 5 results will be put out every one second (0s, 0.2s, 0.4s, 0.6s, 0.8s, 1.0s…).
   c) *Note*: You can use all parameters which are accepted from the operation “buildModel” except the parameters “Start” and “Stop”. These parameters are unnecessary because an interactive simulation always starts at the time “0s” and runs as long as it won’t be stopped or aborted.
3. Execute the created simulation runtime with the parameter “-interactive” and with a port for the control server optionally “-port xxxxx”. After starting the runtime it will wait until a client connects to its control server port. Now you can enter the operations mentioned above.

5.1.6 Interactive Simulation Example

In this chapter we will explain how to simulate a Modelica system interactively. This procedure should be a default step by step procedure for using OMI with a UI.

5.1.6.1 How to get an example Modelica Model

The application sample for Windows is present in C:\OpenModelica1.8.0\share\doc\omc\interactive-simulation. Also read C:\OpenModelica1.8.0\share\doc\omc\interactive-simulation\README.txt.

The source code for the client is in the Subversion repository: trunk/c_runtime/interactive.

An application test is in the Subversion repository here: trunk/testsuite/interactive-simulation.

See here how to get the code: https://www.openmodelica.org/index.php/developer/source-code

5.1.6.2 Create the simulation runtime

We will use an example system based on a demonstration model which is given in the Modelica book by Peter Fritzson [[2], Page 386].

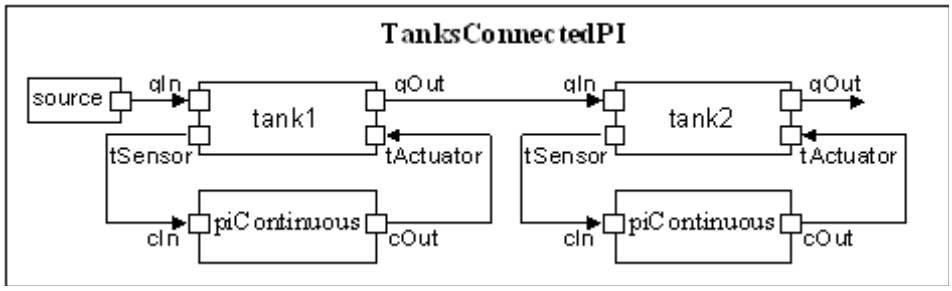
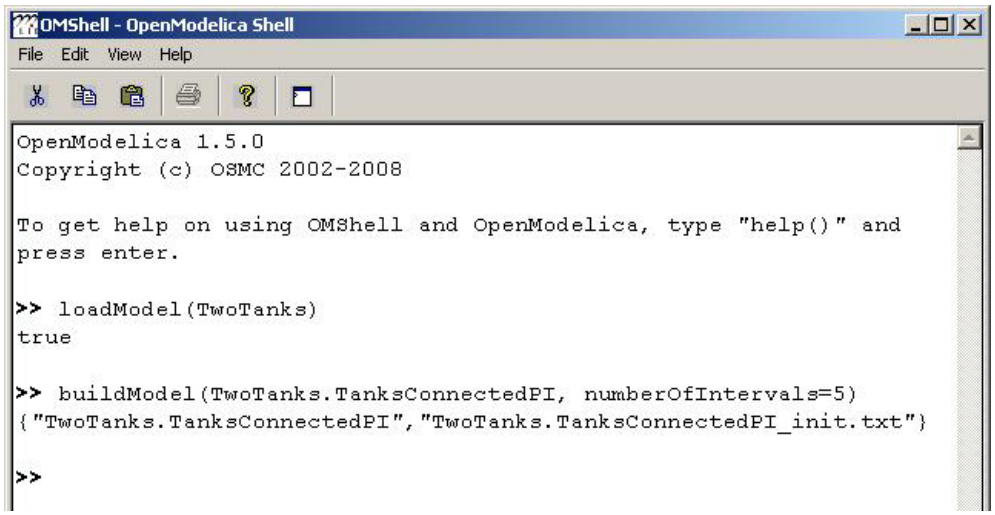


Figure 5-2. TanksConnectedPI structure diagram.

Please follow the steps to create an executable simulation runtime file.

1. Start OMShell "Start->OpenModelica->OpenModelica Shell"
2. Enter the operation **"loadModel(TwoTanks)"**
   NOTE: We assume that the TwoTanks model is in the ModelicaLibary OM installation folder **(…\OpenModelica1.8.0\ModelicaLibrary\TwoTanks)** otherwise please load the file from its location **(…\OpenModelica1.8.0\share\doc\omc\interactive-simulation*.zip).**
3. Use the "**buildModel**" operation with the following parameters to build the TwoTanks model: **buildModel(TwoTanks.TanksConnectedPI, numberOfIntervals=5)**

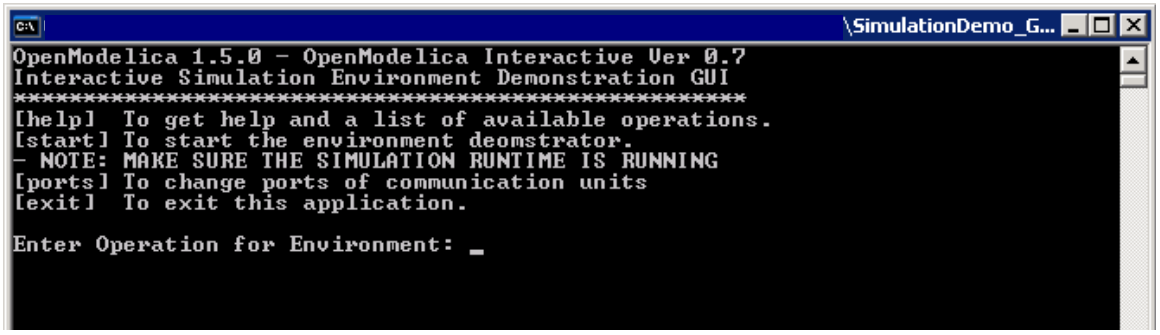


5.1.6.3 Start an interactive Simulation Session

Start the created simulation runtime it should be located in the "tmp" folder of the OM installation folder (…\OpenModelica1.8.0\tmp\TwoTanks.TanksConnectedPI.exe)

Use the b "`-interactive -port xxxxx`". **NOTE:** If the default port (10501) should be used ignore the parameter "–port". Now the simulation runtime will be waiting until a UI client has been connected on its port.

Start the client: "client.exe".



(Deprecated: Now enter "`start`" into the console and wait until the client is successfully connected.)

Enter following operation for the simulation runtime:

```
setcontrolclienturl#1#127.0.0.1#10500#end
settransferclienturl#2#127.0.0.1#10502#end
setfilter#3#tank1.h#source.flowLevel#end
```

Start the simulation with: `start#4#end`

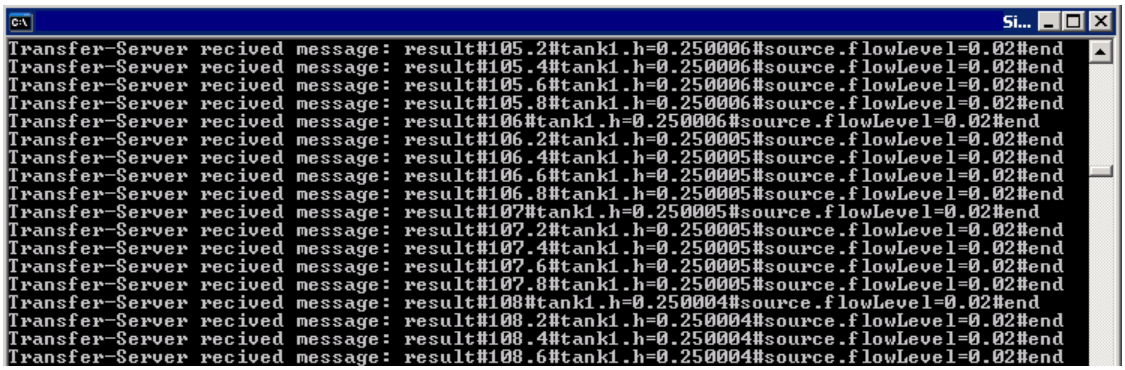
NOTE: After starting the simulation your keyboard entries and the results will be displayed in the same console and you can't see what you are typing. Please pause the simulation first than enter a longer operation string.

Pause the simulation with: `pause#5#end`

Change a Value with: changevalue#6#xx.x#source.flowLevel=0.04#end.

For example if time is higher than 60 and lower than 200 enter →

```
changevalue#6#60.0#source.flowLevel=0.0004#end
```



Shutdown the simulation runtime and the environment with: `shutdown#7#end`

5.2 OPC and OPC UA Interfaces

In addition to OMI, OpenModelica can also be stimulated through the OPC interface. At the moment OPC DA and OPC UA interfaces are supported. As OPC UA is seen as the technology that will replace the regular OPC in the future, the OPC UA implementation is concentrated on more than OPC DA.

5.2.1 Introduction to the OPC Interfaces

In this chapter, the basics of OPC are explained. In addition, a literature survey [OPC_1] has been made for the project "OPC Interfaces in OpenModelica". In that survey, OPC is explained on a higher level generally as well as from the viewpoint of OpenModelica.

OPC is a set of specifications which defines a common interface for communication between software packages and hardware devices of different kinds. The most used of the OPC interfaces, OPC DA (3.00) specification [OPC_2], defines how to get and set data in real time devices using client-server communication paradigm and Ethernet based communication. OPC DA uses Microsoft COM (Component Object Model) technology.

The next generation of OPC specifications is OPC Unified Architecture. It combines the different OPC interfaces under one specification. The basic idea behind OPC UA is the same even though the technology behind is different. Unlike the regular OPC specification, OPC UA is platform independent and has new features, such as method calls, included. A binary transfer protocol is provided for a better performance, as well as the XML based one. The OPC UA specification is defined in a 13-part specification downloadable in OPC Foundation homepage [OPC_3].

Since OPC and OPC UA are rather large specifications, their usage cannot be discussed here in detail. For a deeper understanding of the technology behind OPC and OPC UA the reader should get familiar with the interface specifications above. However, for both of the interfaces there exist test clients which can be used to utilize the interfaces in a quick and easy manner. A couple of test clients are introduced in section 5.2.3.

5.2.2 Implemented Features

Both the OPC UA and the OPC DA interface (combined with the Simulation Control (SC) interface) provide a very similar functionality. Thus only OPC UA is discussed extensively here. In subsection 5.2.2.2 the most noteworthy differences in features between OPC UA and OPC are discussed.

To utilize either of the OPC interfaces, the OPC branch needs to be merged to c_runtime. UA Server is linked to the simulation executable by adding -lOPCregistry -lua_server at the end of the compiling script. OPC DA server is linked to the simulation executable by adding -lOPCregistry -lOPCKit at the end of the compiling script. At the moment there is no option in the OMC to do this automatically.

5.2.2.1 OPC UA

After an OPC UA client has established a connection to an OpenModelica simulation, the following features can be utilized. In Figure 5-3 UA Expert connected to an OpenModelica simulation ('dcmotor') with the OPC UA server included, a view of the UA Expert test client is shown. The client is connected to an example OpenModelica simulation, namely the 'dcmotor' [OPC_4]. In the following, all the examples use Figure 5-3 UA Expert connected to an OpenModelica simulation ('dcmotor') with the OPC UA server included to explain the concepts.

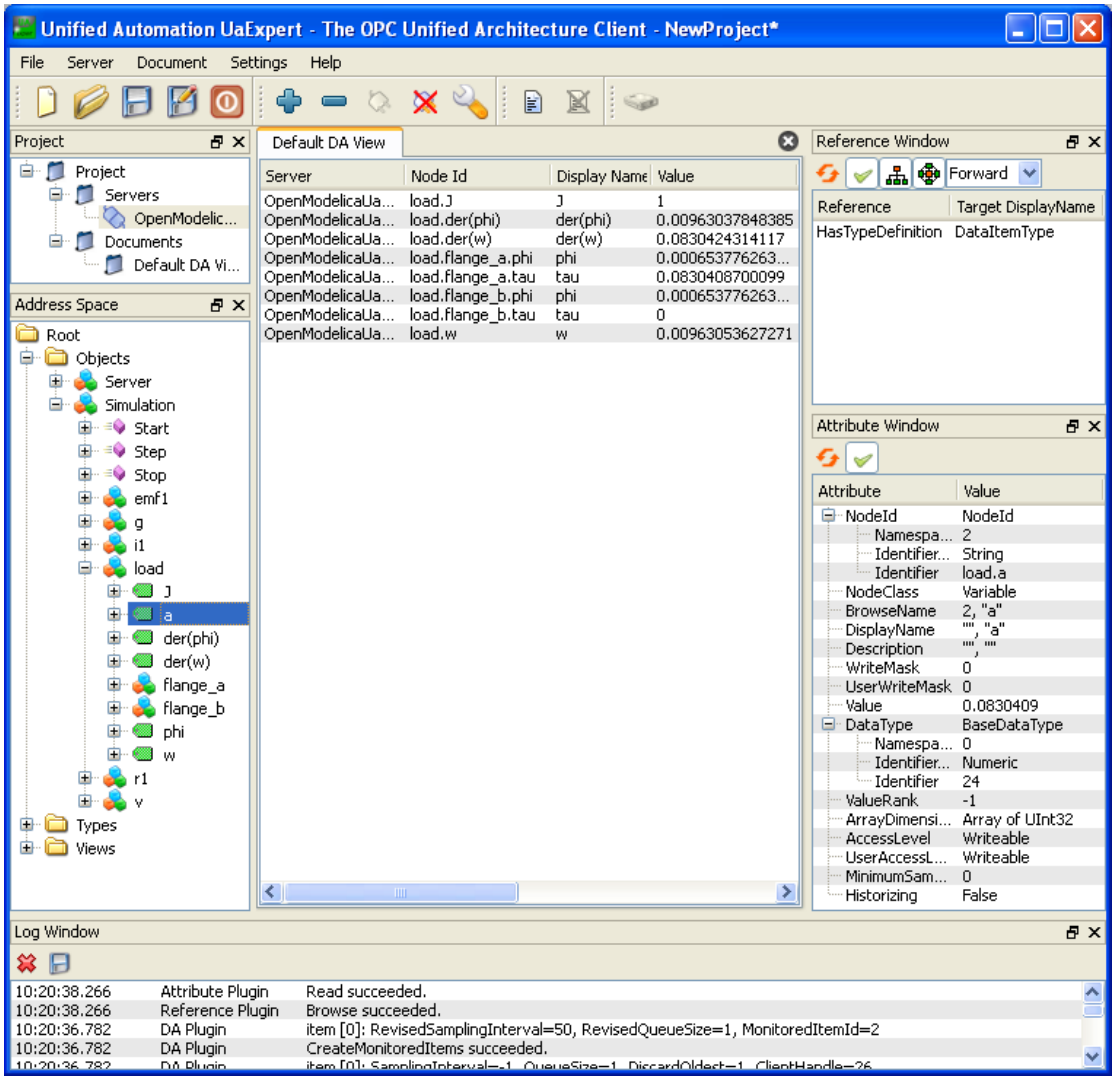


Figure 5-3 UA Expert connected to an OpenModelica simulation (‘dcmotor’) with the OPC UA server included

Browse

The data structure (in OPC UA terminology: address space) of the OpenModelica simulation can be browsed. The address space consists of nodes and references between them. The part of the address space in which all the elements of the simulation are is a tree under the ‘Simulation’ node. This is shown in the left plane in Figure 5-3. In addition, the three methods are there as nodes as well; these methods are discussed later.

Read

Values of variables and parameters can be read through OPC UA after each simulation step. In addition to that, OPC UA offers some metadata, the majority of which is not utilized, though. The value and the metadata can be found on the right plane in Figure 5-3.

Write

Values of attributes and parameters can also be altered during the simulation between the simulation steps. When a value has been changed, the simulation is initialized with the new values and continued. This is needed if variables are wished to be changed, however a parameter change would not require this. Hence the implementation should be fixed on this matter.

Subscribe

Variable (and parameter) values can be subscribed. There are two alternative types of subscription: The first option is that the OPC UA server sends the value of the subscribed variable to the client each time after a defined time interval has passed (real time). The other one is that the value is sent to the client after each simulation step. In UA Expert, variables can be subscribed by dragging them to the middle plane.

Start, Stop, Step

The simulation can be controlled by the three methods: start(), stop(), and step(). With the OPC UA server included, the OpenModelica simulation starts in a stopped state. With start() and stop() methods this state can be changed. The simulation can also be run one step at a time with step().

5.2.2.2 OPC DA and Simulation Control (SC)

The OPC DA server offers roughly the same functionality than the OPC UA server. The biggest conceptual difference between the two specifications is groups: Before variables and parameters can be used in any way they have to be grouped. A group is an entity consisting of items. A group can contain any variables and parameters as items. The other major differences between the two interfaces are described in the following.

Browse

The data structure of OPC DA is a tree consisting of branches and leaves. The leaves correspond to variables and parameters whereas the branches form the tree-like structure (e.g. ‘load’ and ‘flange_a’ are shown as branches in the dcmotor example).

Read

Reading values doesn’t differ much from OPC UA, except that the items read have to be grouped. There is also almost no metadata available through the OPC DA.

Write

There are no major differences.

Subscribe

The biggest difference in OPC DA is that single items cannot be subscribed. Instead, a subscription can be made for a group.

Start, Stop, Step

OPC DA interface doesn’t enable methods as such. Thus a proprietary interface, Simulation Control (SC), is used. In practice this means that in addition to the OPC DA client an SC client must be run alongside.

5.2.3 Test clients

There are free test clients available in the Internet to test the OPC and OPC UA interfaces. One test client for each interface is shortly presented below. In addition, a test client for the SC interface is published as part of OpenModelica.

UA Expert by Unified Automation is an OPC UA test client with a GUI. It supports all the functionalities provided by the OPC UA in OpenModelica. The latest version of UA Expert can be downloaded for free from the Unified Automation homepage [OPC_5].

For OPC DA there are numerous test clients available. The one which was most used for testing the OPC server is MatrikonOPC Explorer. It supports all the OPC DA functionalities needed to access data in

OpenModelica. As well as UA Expert, this test client has a graphical user interface. The latest version of MatrikonOPC Explorer can be downloaded for free from the MatrikonOPC homepage [OPC_6].

As the SC interface is a non-standard interface, there are no clients for it in the Internet. SimpleOPCClient is a small test client for OPC and SC interfaces published with OpenModelica. Besides some basic OPC features, this test client allows the simulation to be controlled, i.e. start(), stop(), and step() functions can be used with it. Unlike the commercial products above, SimpleOPCClient is published in source code format.

5.2.4 References

[OPC_1] OPC Interfaces in OpenModelica – Technical Specification (Task 5.3); Online: https://openmodelica.org/svn/OpenModelica/trunk/doc/opc/OPC_Interfaces_in_OpenModelica.pdf (Accessed 10 June 2011).

[OPC_2] OPC DA 3.00 Specification; Online: http://opcfoundation.org/DownloadFile.aspx?CM=3&RI=67&CN=KEY&CI=283&CU=6 (Accessed 9 June 2011).

[OPC_3] The OPC Foundation – The Interoperability Standard for a Connected World; Online: http://opcfoundation.org/ (Accessed 8 June 2011).

[OPC_4] OpenModelica: DC Motor model; Online: https://openmodelica.ida.liu.se/svn/OpenModelica/tags/OPENMODELICA_1_5_0/Examples/dcmotor.mo (Accessed 10 June 2011).

[OPC_5] Unified Automation GmbH | OPC UA Clients; Online: http://www.unified-automation.com/opc-ua-clients/ (Accessed 9 June 2011).

[OPC_6] MatrikonOPC: Free OPC Downloads; Online: http://www.matrikonopc.com/downloads/ (Accessed 9 June 2011).

Chapter 6

Model Import and Export with FMI 1.0

The new standard for model exchange with Functional Mockup Interface (FMI) 1.0 allows export of pre-compiled models, i.e., C-code or binary code, from a tool for import in another tool, and vice versa. The FMI model exchange standard is Modelica independent. Import and export works both between different Modelica tools, or between certain non-Modelica tools and Modelica tools. OpenModelica supports FMI 1.0 import as well as FMI 1.0 export.

6.1 FMI Import

To import the FMU package use the OpenModelica command `importFMU("fmufile", "outputdirectory")` from command line interface, OMShell, OMNotebook or MDT. The `importFMU` command is also integrated with OMEdit. Select `FMI > Import FMI` the FMU package is extracted in the directory specified by `outputdirectory`, since the `outputdirectory` parameter is optional so if its not specified then the current directory of omc is used. You can use the `cd()` command to see the current location.

The FMI Import is currently a prototype. It was implemented on the Windows 7 x64 platform under the MinGW environment. The prototype has been tested in OpenModelica 1.8.0 (revision-10129) with several examples. A more fullfleged version for FMI Import will be released in the near future. The present version can be used as a stand-alone FMU import for Modelica simulators.

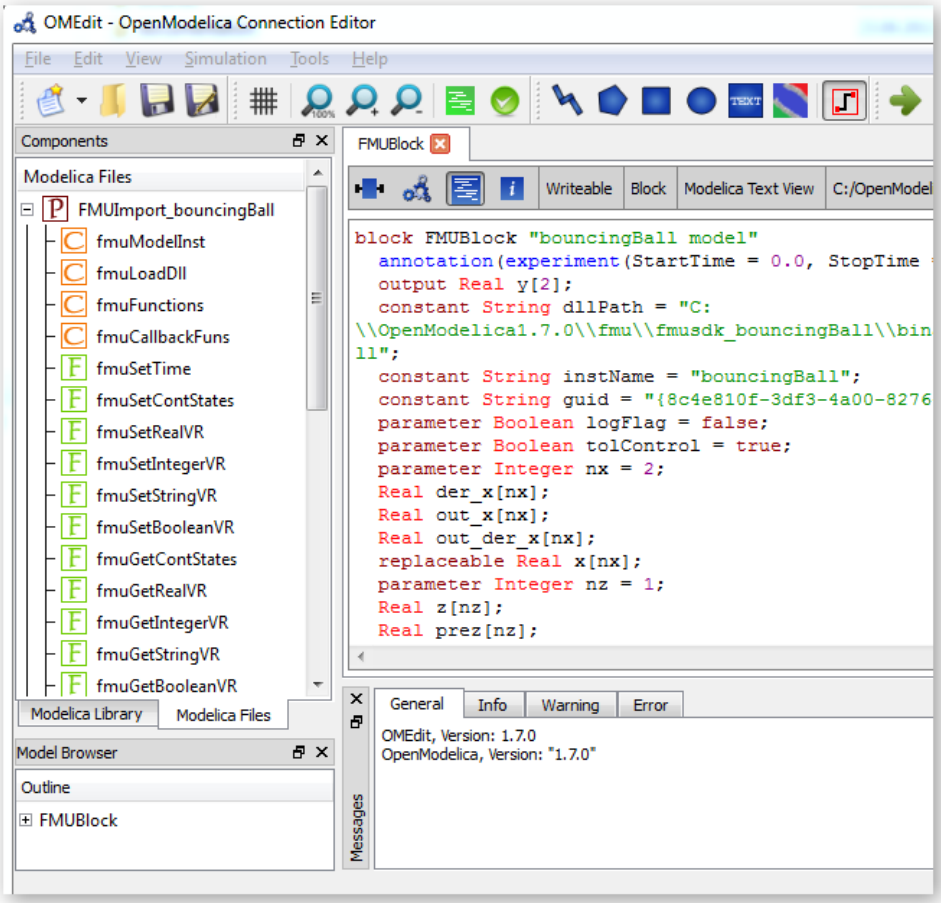


Figure 6-1: Example of FMU Import in OpenModelica where a bouncing ball model is imported.

6.2 FMI Export

To export the FMU use the OpenModelica command `translateModelFMU(ModelName)` from command line interface, OMShell, OMNotebook or MDT. The export FMU command is also integrated with OMEdit. Select `FMI > Export FMI` the FMU package is generated in the current directory of omc. You can use the `cd()` command to see the current location.

After the command execution is complete you will see that a file `ModelName.fmu` has been created. As depicted in Figure 6-2, we first changed the current directory to `C:/OpenModelica1.7.0/bin` , then we loaded a Modelica file with `BouncingBall` example model and finally created an FMU for it using the `translateModelFMU` call.

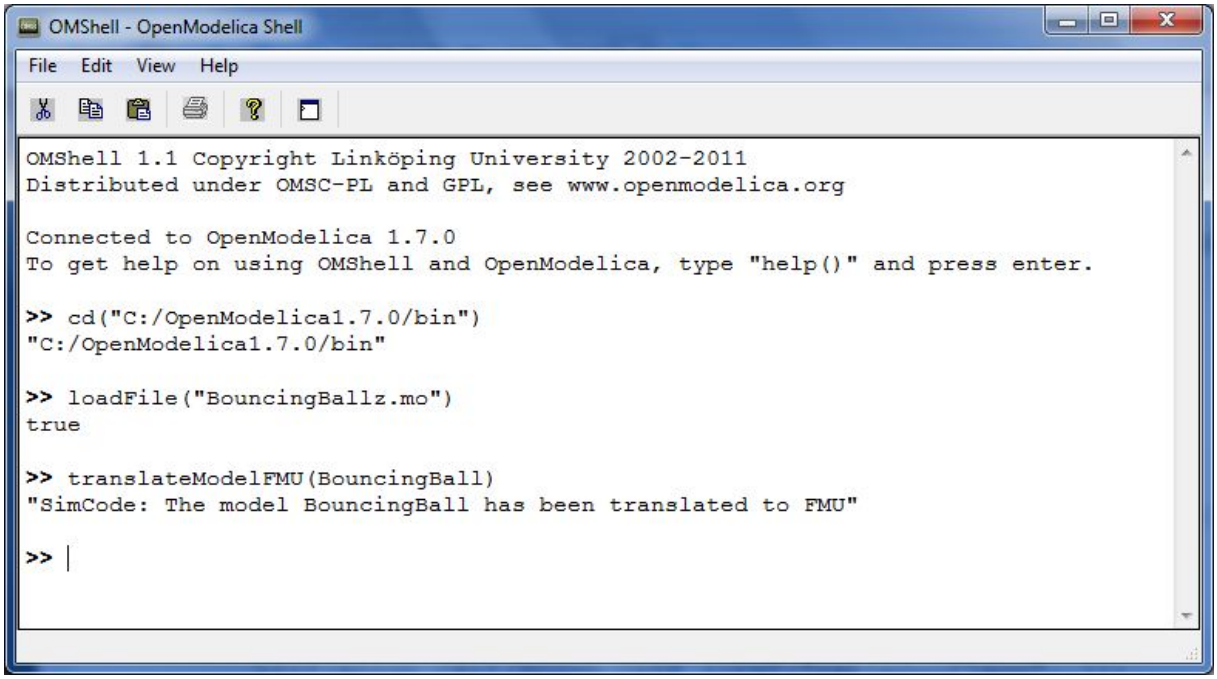


Figure 6-2: OMShell screenshot for creating an FMU

A log file for FMU creation is also generated named `ModelName_FMU.log`. If there are some errors while creating FMU they will be shown in the command line window and logged in this log file as well.

Chapter 7

OMOptim – Optimization with OpenModelica

7.1 Introduction

OMOptim is a tool dedicated to optimization of Modelica models. By optimization, one should understand a procedure which minimizes/maximizes one or more objectives by adjusting one or more parameters.

OMOptim 0.9 contains meta-heuristic optimization algorithms which allow optimizing all sorts of models with following functionalities:

- One or several objectives optimized simultaneously
- One or several parameters (integer or real variables)

However, the user must be aware of the large number of simulations an optimization might require.

7.2 Preparing the Model

Before launching OMOptim, one must prepare the model in order to optimize it.

7.2.1 Parameters

An optimization parameter is picked up from all model variables. The choice of parameters can be done using the OMOptim interface.

For all intended parameters, please note that:

- The corresponding variable is ***constant*** during all simulations. The OMOptim optimization in version 0.9 only concerns static parameters' optimization *i.e.* values found for these parameters will be constant during all simulation time.
- The corresponding variable should play an ***input*** role in the model *i.e.* its modification influences model simulation results.

7.2.2 Constraints

If some constraints should be respected during optimization, they must be defined in the Modelica model itself.

For instance, if mechanical stress must be less than 5 $N.m^{-2}$, one should write in the model:

```
assert( mechanicalStress < 5, "Mechanical stress too high");
```

If during simulation, the variable *mechanicalStress* exceeds 5 $N.m^{-2}$, the simulation will stop and be considered as a failure.

7.2.3 Objectives

As parameters, objectives are picked up from model variables. Objectives' values are considered by the optimizer at the final time.

7.3 Set problem in OMOptim

7.3.1 Launch OMOptim

OMOptim can be launched using the executable placed in `OpenModelicaInstallationDirectory/bin/ OMOptim/OMOptim.exe`. Alternately, choose `OpenModelica > OMOptim` from the start menu.

7.3.2 Create a new project

To create a new project, click on menu `File -> New project`

Then set a name to the project and save it in a dedicated folder. The created file created has a `.min` extension. It will contain information regarding model, problems, and results loaded.

7.3.3 Load models

First, you need to load the model(s) you want to optimize. To do so, click on *Add .mo* button on main window or select menu *Model -> Load Mo file…*

When selecting a model, the file will be loaded in OpenModelica which runs in the background.

While OpenModelica is loading the model, you could have a frozen interface. This is due to multi-threading limitation but the delay should be short (few seconds).

You can load as many models as you want.

If an error occurs (indicated in log window), this might be because:

- Dependencies have not been loaded before (e.g. modelica library)
- Model use syntax incompatible with OpenModelica.

7.3.3.1 Dependencies

OMOptim should detect dependencies and load corresponding files. However, it some errors occur, please load by yourself dependencies. You can also load Modelica library using `Model->Load Modelica library.`

When the model correctly loaded, you should see a window similar to Figure 7-1.

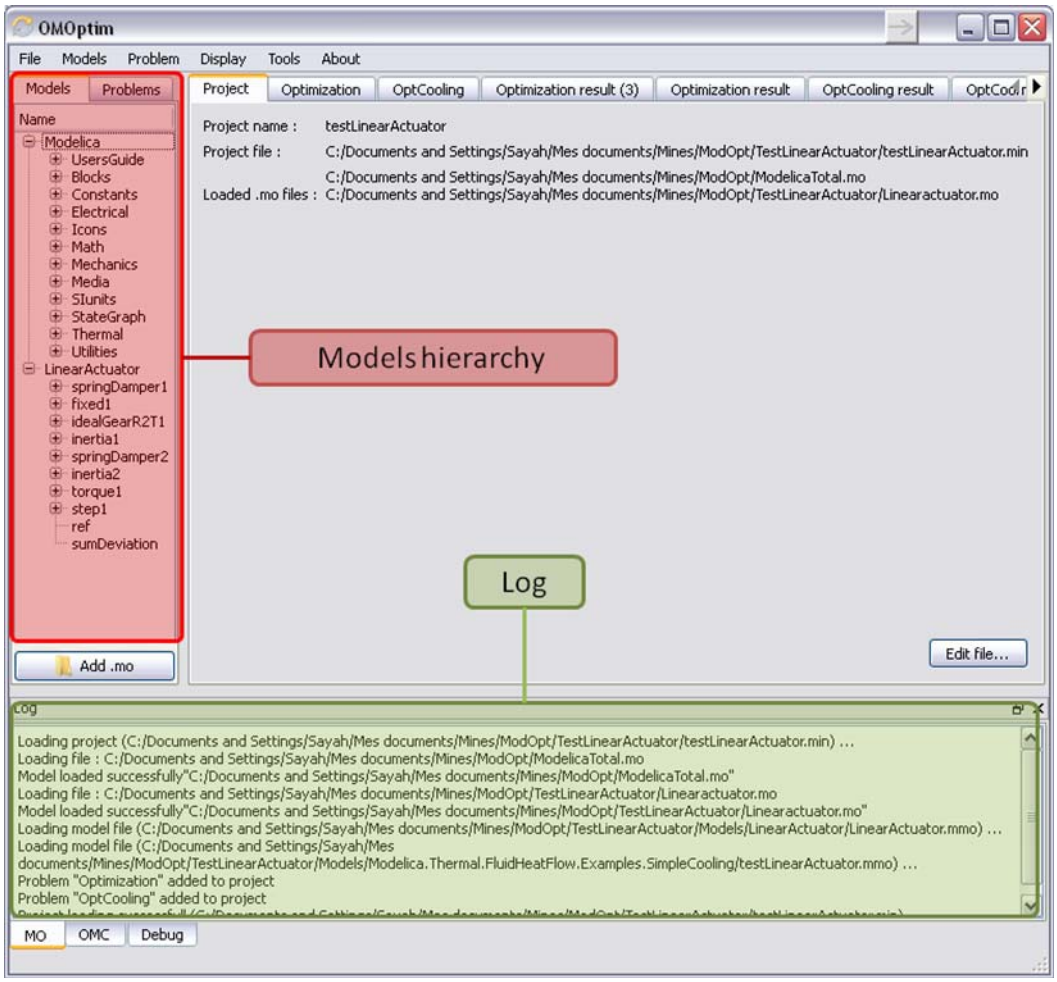


Figure 7-1. OMOptim window after having loaded model.

7.3.4 Create a new optimization problem

`Problem->Add Problem->Optimization`

A dialog should appear. Select the model you want to optimize. Only `Model` can be selected (no `Package`, `Component`, `Block`…).

A new form will be displayed. This form has two tabs. One is called `Variables`, the other is called `Optimization`.

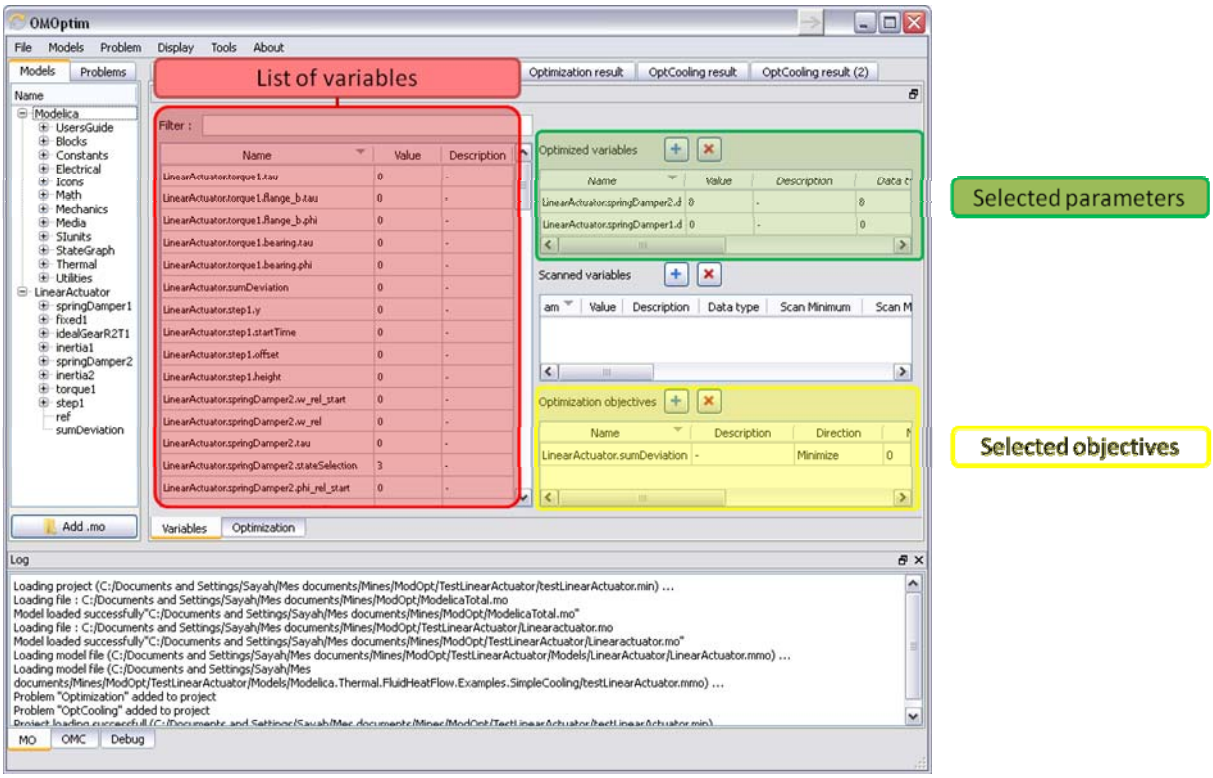


Figure 7-2. Forms for defining a new optimization problem.

7.3.4.1 List of Variables is Empty

If variables are not displayed, right click on model name in model hierarchy, and select *Read variables*.

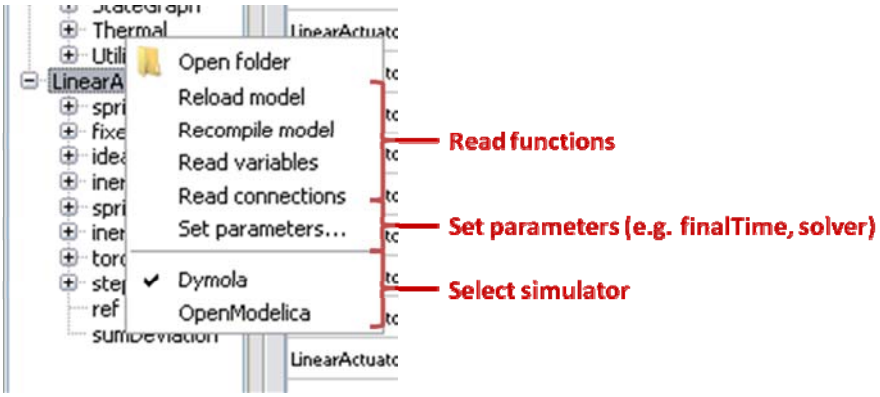


Figure 7-3. Selecting read variables, set parameters, and selecting simulator.

7.3.5 Select Optimized Variables

To set optimization, we first have to define the variables the optimizer will consider as free *i.e.* those that it should find best values of. To do this, select in the left list, the variables concerned. Then, add them to *Optimized variables* by clicking on corresponding button (✚).

For each variable, you must set minimum and maximum values it can take. This can be done in the *Optimized variables* table.

7.3.6 Select objectives

Objectives correspond to the final values of chosen variables. To select these last, select in left list variables concerned and click ✚ button of *Optimization objectives* table.

For each objective, you must:

- Set minimum and maximum values it can take. If a configuration does not respect these values, this configuration won't be considered. You also can set minimum and maximum equals to "-" : it will then
- Define whether objective should be minimized or maximized.

This can be done in the *Optimized variables* table.

7.3.7 Select and configure algorithm

After having selected variables and objectives, you should now select and configure optimization algorithm. To do this, click on *Optimization* tab.

Here, you can select optimization algorithm you want to use. In version 0.9, OMOptim offers three different genetic algorithms. Let's for example choose `SPEA2Adapt` which is an auto-adaptative genetic algorithm.

By clicking on *parameters*… button, a dialog is opened allowing defining parameters. These are:

- Population size: this is the number of configurations kept after a generation. If it is set to 50, your final result can't contain more than 50 different points.
- Off spring rate: this is the number of children per adult obtained after combination process. If it is set to 3, each generation will contain 150 individual (considering population size is 50).
- Max generations: this number defines the number of generations after which optimization should stop. In our case, each generation corresponds to 150 simulations. Note that you can still stop optimization while it is running by clicking on *stop* button (which will appear once optimization is launched). Therefore, you can set a really high number and still stop optimization when you want without losing results obtained until there.
- Save frequency: during optimization, best configurations can be regularly saved. It allows to analyze evolution of best configurations but also to restart an optimization from previously obtained results. A Save Frequency parameter set to 3 means that after three generations, a file is automatically created containing best configurations. These files are named iteraion1.sav, iteration2.sav and are store in *Temp* directory, and moved to *SolvedProblems* directory when optimization is finished.
- ReinitStdDev: this is a specific parameter of EAAdapt1. It defines whether standard deviation of variables should be reinitialized. It is used only if you start optimization from previously obtained configurations (using *Use start file* option). Setting it to yes (1) will, in most of cases, lead to a spread research of optimized configurations, forgetting parameters' variations' reduction obtained in previous optimization.

7.3.7.1 Use start file

As indicated before, it is possible to pursue an optimization finished or stopped. To do this, you must enable *Use start file* option and select file from which optimization should be started. This file is an *iteration_.sav* file created in previous optimization. It is stored in corresponding *SolvedProblems* folder (*iteration10.sav* corresponds to the tenth generation of previous optimization).

Note that this functionality can only work with same variables and objectives. However, minimum, maximum of variables and objectives can be changed before pursuing an optimization.

7.3.8 Launch

You can now launch Optimization by clicking *Launch* button.

7.3.9 Stopping Optimization

Optimization will be stopped when the generation counter will reach the generation number defined in parameters. However, you can still stop the optimization while it is running without loosing obtained results. To do this, click on *Stop* button. Note that this will not immediately stop optimization: it will first finish the current generation.

This stop function is especially useful when optimum points do not vary any more between generations. This can be easily observed since at each generation, the optimum objectives values and corresponding parameters are displayed in log window.

7.4 Results

The result tab appear when the optimization is finished. It consists of two parts: a table where variables are displayed and a plot region.

7.4.1 Obtaining all Variable Values

During optimization, the values of optimized variables and objectives are memorized. The others are not. To get these last, you must recomputed corresponding points. To achieve this, select one or several points in point's list region and click on *recompute*.

For each point, it will simulate model setting input parameters to point corresponding values. All values of this point (including those which are not optimization parameters neither objectives).

7.5 Window Regions in OMOptim GUI

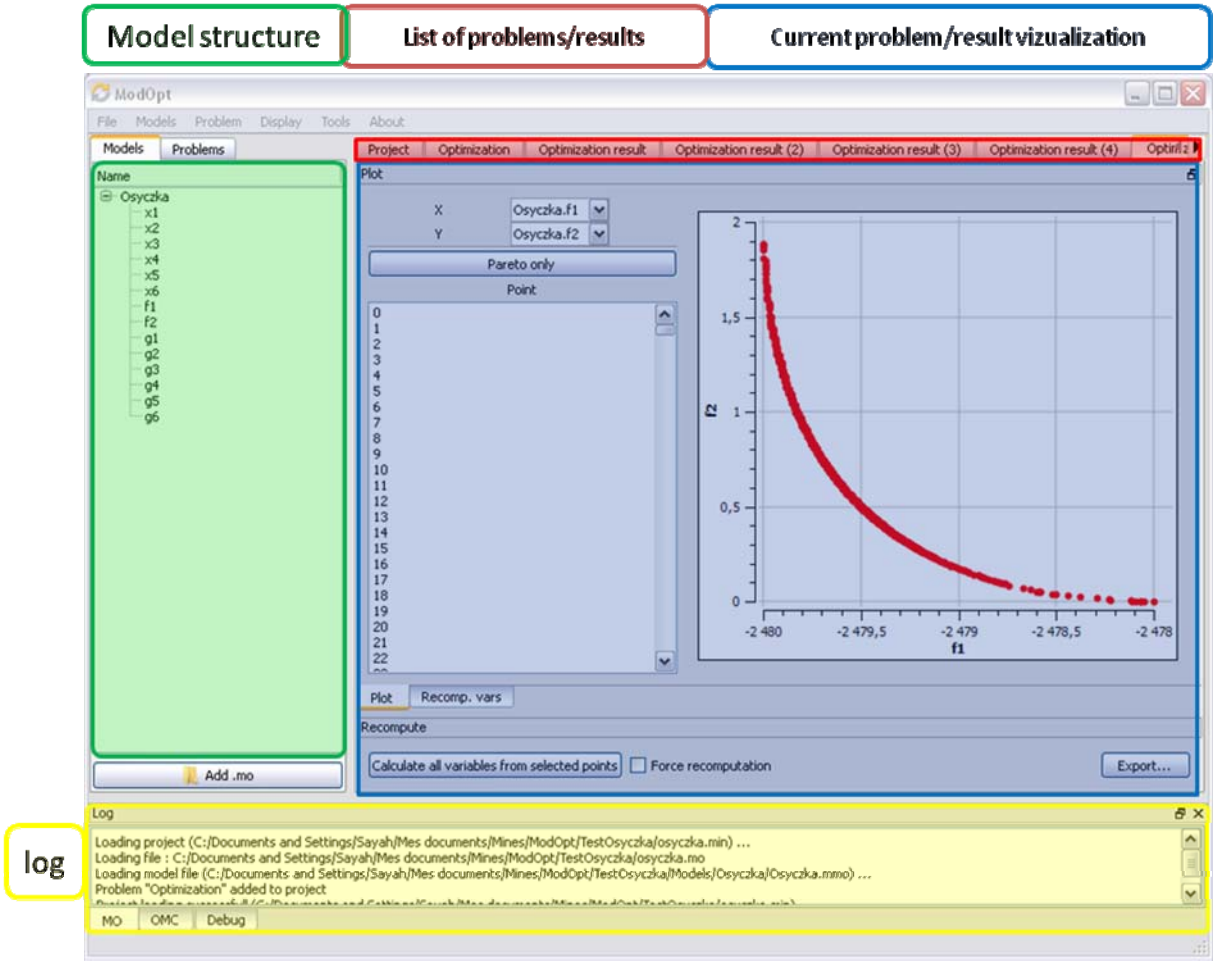


Figure 7-4. Window regions in OMOptim GUI.

Chapter 8

MDT – The OpenModelica Development Tooling Eclipse Plugin

8.1 Introduction

The Modelica Development Tooling (MDT) Eclipse Plug-In as part of OMDev – The OpenModelica Development Environment 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 or other libraries
- Code completion for class names and function argument lists.
- Goto definition for classes, types, and functions.
- Displaying type information when hovering the mouse over an identifier.

8.2 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

8.3 Getting Started

8.3.1 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.

8.3.2 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`..

8.3.3 Selecting a Workspace Folder

Eclipse stores your projects in a folder called a workspace. You need to choose a workspace folder for this session, see Figure 5-8-1

Figure 5-8-1. Eclipse Setup – Switching Workspace.

8.3.4 Creating one or more Modelica Projects

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`.

Figure 5-8-2. Eclipse Setup – creating a Modelica project in the workspace.

You need to disable automatic build for the project(s) (Figure 5-8-3).

Figure 5-8-3. Eclipse Setup – disable automatic build for the projects.

Repeat the procedure for all the projects you need, e.g. for the exercises described in the MetaModelica users guide: 01_experiment, 02a_exp1, 02b_exp2, 03_assignment, 04a_assigntwotype, etc.

NOTE: Leave open only the projects you are working on! Close all the others!

8.3.5 Building and Running a Project

After having created a project, you eventually need to build the project (Figure 8-4).

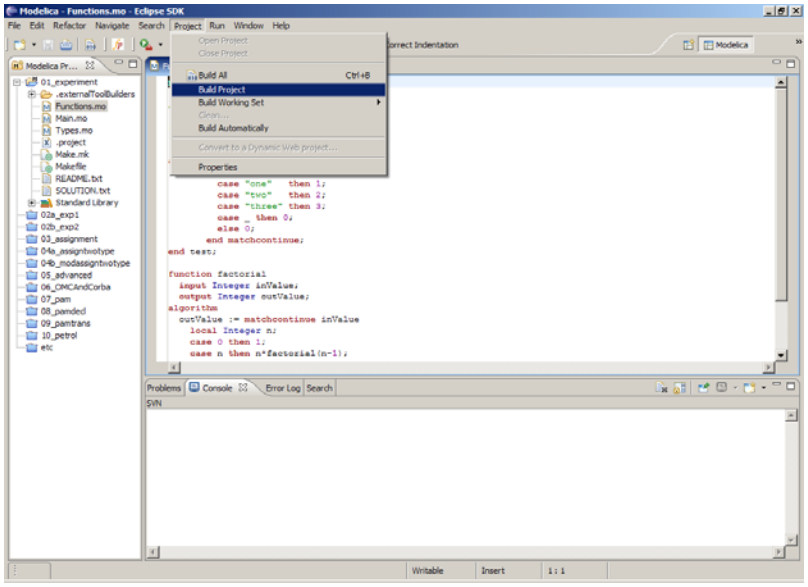


Figure 8-4. Eclipse MDT – Building a project.

There are several options: building, building from scratch (clean), running, see Figure 8-5.

??missing figure

Figure 8-5. Eclipse – building and running a project.

You may also open additional views, e.g as in Figure 8-6.

??missing figure

Figure 8-6. Eclipse – Opening views.

8.3.6 Switching to Another Perspective

If you need, you can (temporarily) switch to another perspective, e.g. to the Java perspective for working with an OpenModelica Java client as in Figure 8-7.

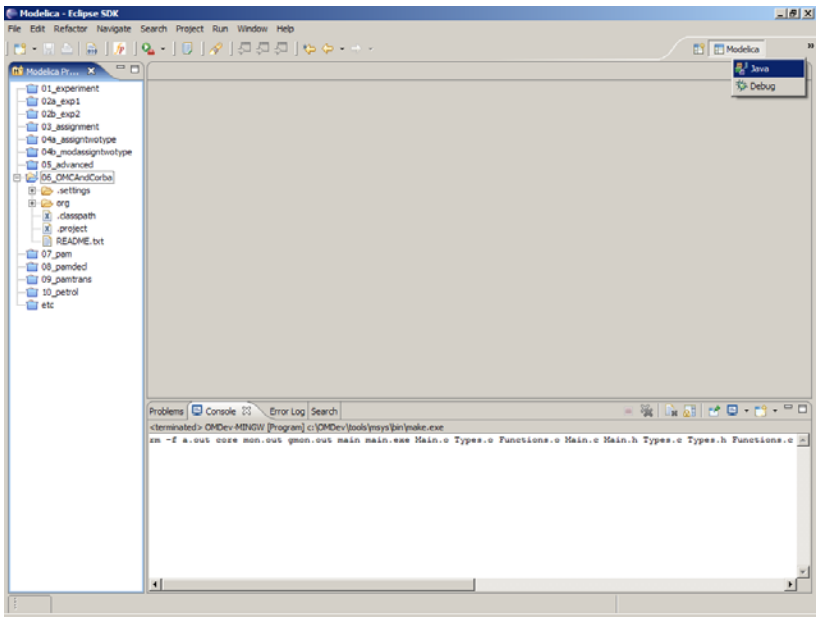


Figure 8-7. Eclipse – Switching to another perspective – e.g. the Java Perspective.

8.3.7 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. Note: for the exercises we already have existing packages.

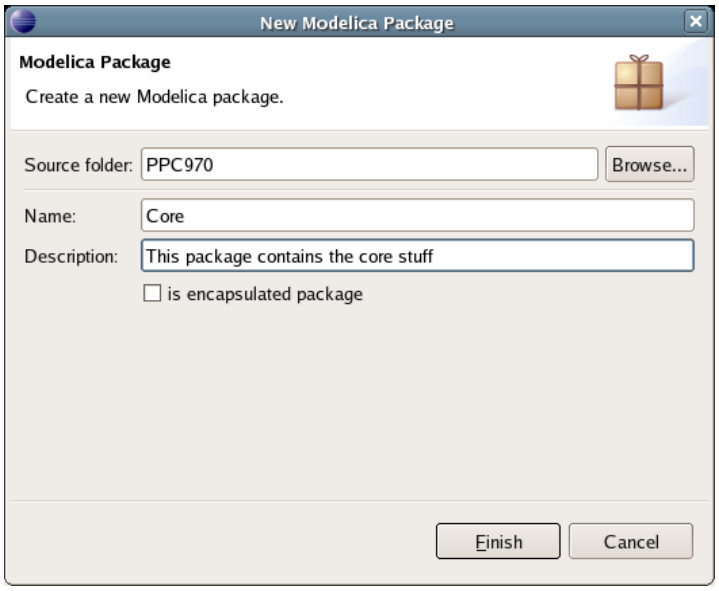


Figure 8-8. Creating a new Modelica package.

8.3.8 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.

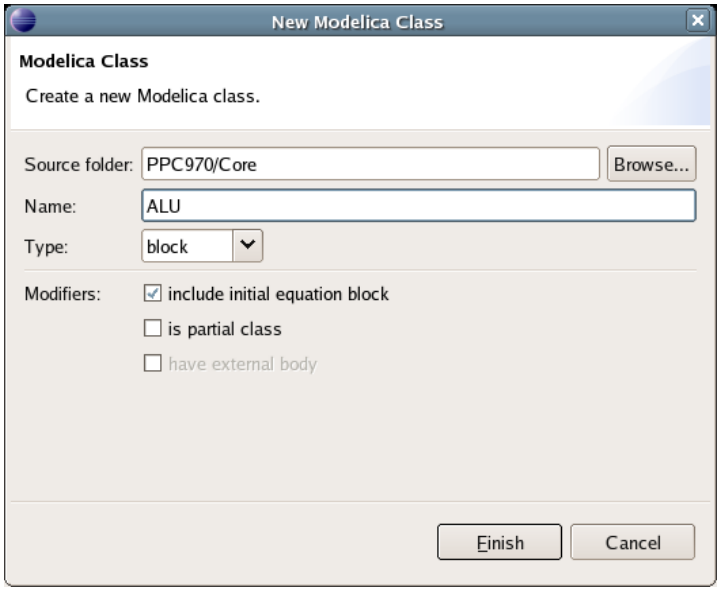


Figure 8-9. Creating a new Modelica class.

8.3.9 Syntax Checking

Whenever a build command is given to the MDT environment, modified and saved Modelica (`.mo`) files are 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.

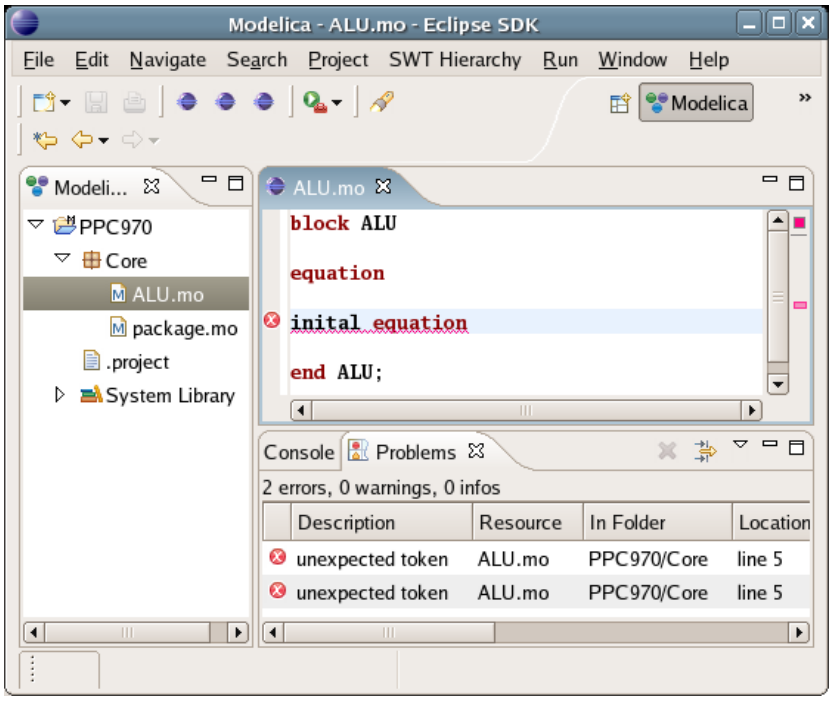


Figure 8-10. Syntax checking.

8.3.10 Automatic 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.

8.3.11 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. Besides the alternatives, Modelica documentation from comments is shown if is available. This makes the selection easyer.

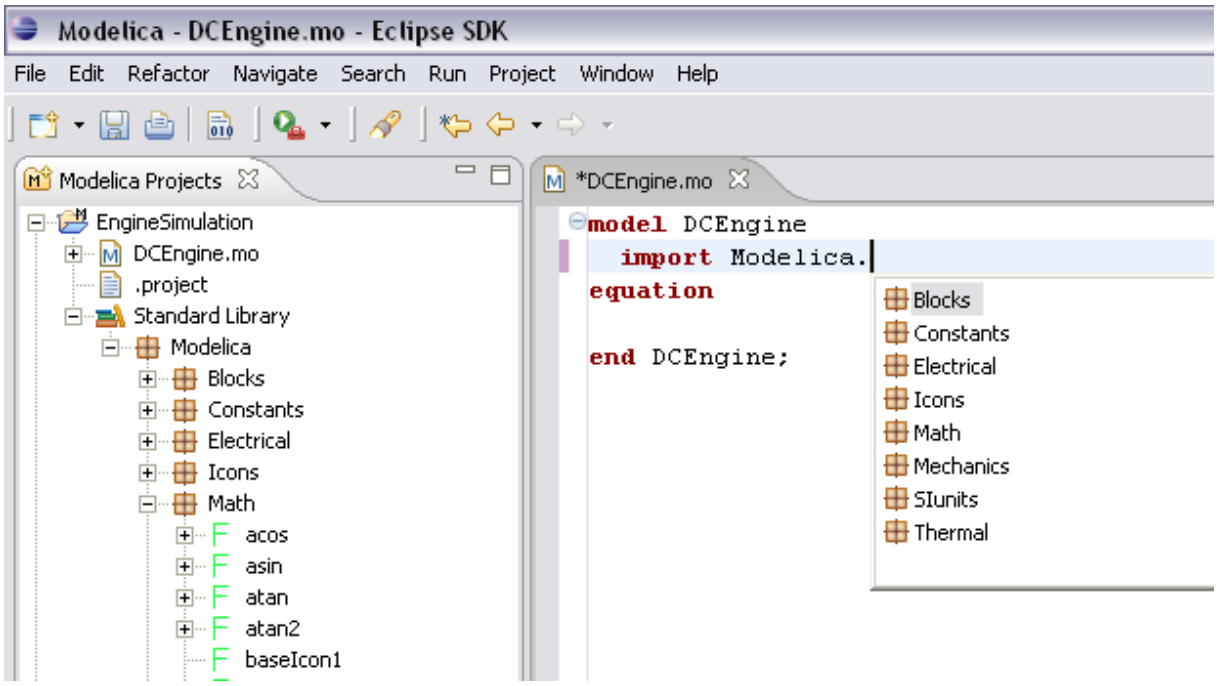


Figure 8-11. Code completion when typing a dot.

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:

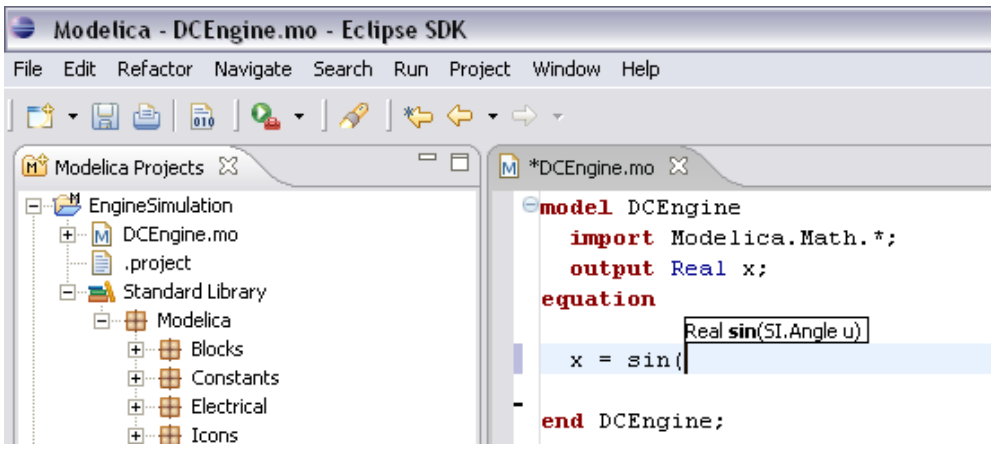


Figure 8-12. Code completion at a function call when typing left parenthesis.

8.3.12 Code Assistance on Identifiers when Hovering

When hovering with the mouse over an identifier a popup with information about the identifier is displayed. If the text is too long, the user can press `F2` to focus the popup dialog and scroll up and down to examine all the text. As one can see the information in the popup dialog is syntax-highlighted.

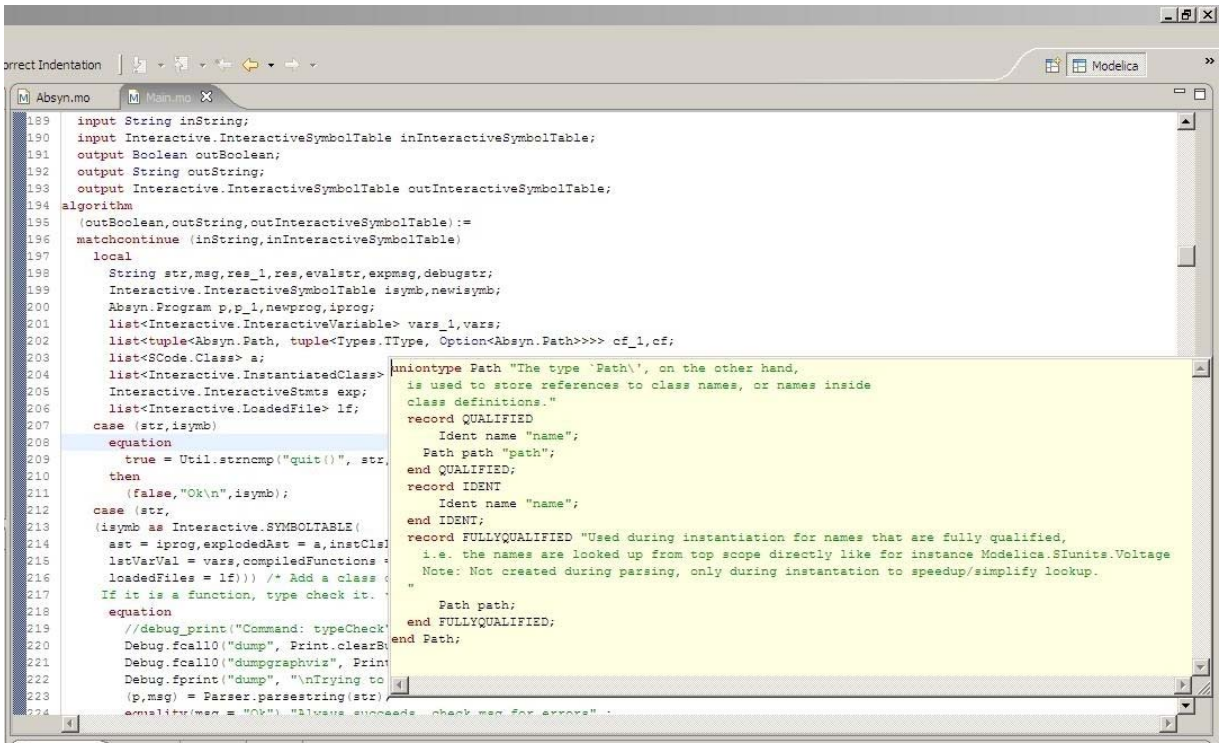


Figure 8-13. Displaying information for identifiers on hovering

8.3.13 Go to Definition Support

Besides hovering information the user can press `CTRL+click` to go to the definition of the identifier. When pressing `CTRL` the identifier will be presented as a link and when pressing mouse click the editor will go to the definition of the identifier.

8.3.14 Code Assistance on Writing Records

When writing records, the same functionality as for function calls is used. This is useful especially in MetaModelica when writing cases in match constructs.

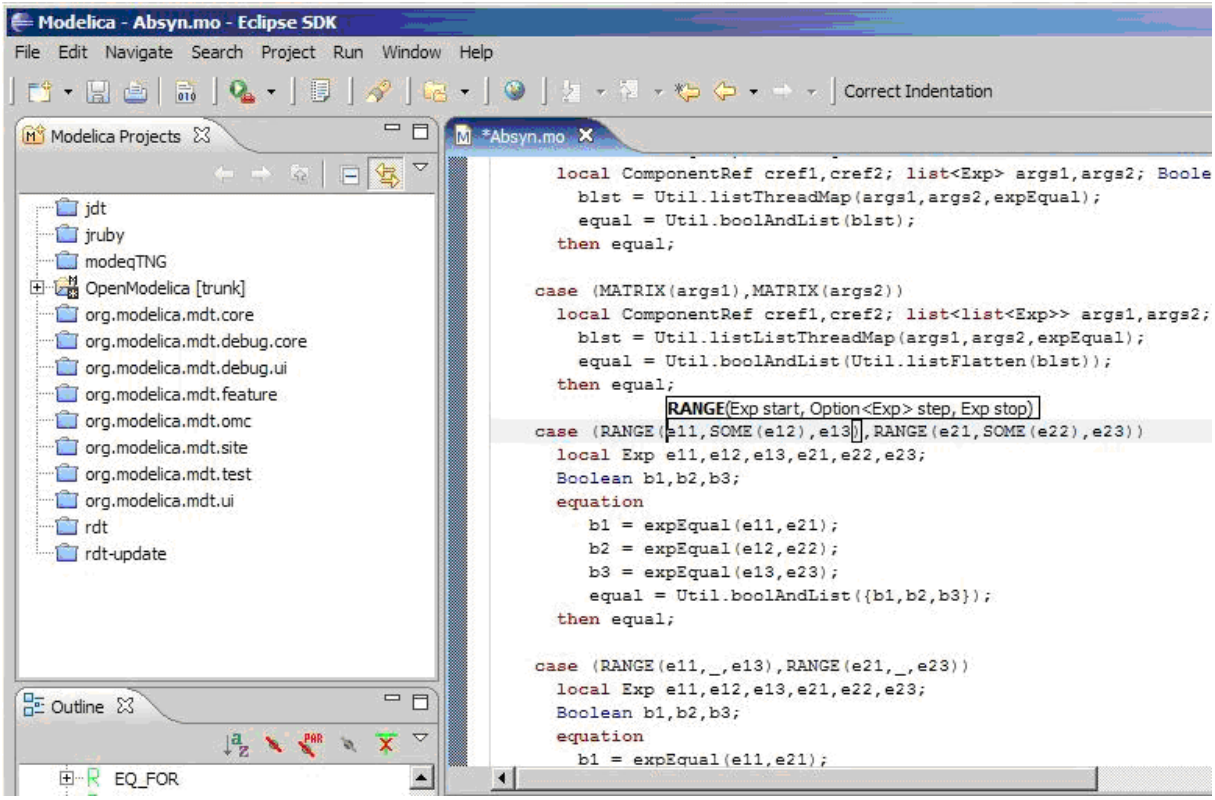


Figure 8-14. Code assistance when writing cases with records in MetaModelica.

8.3.15 Using the MDT Console for Plotting

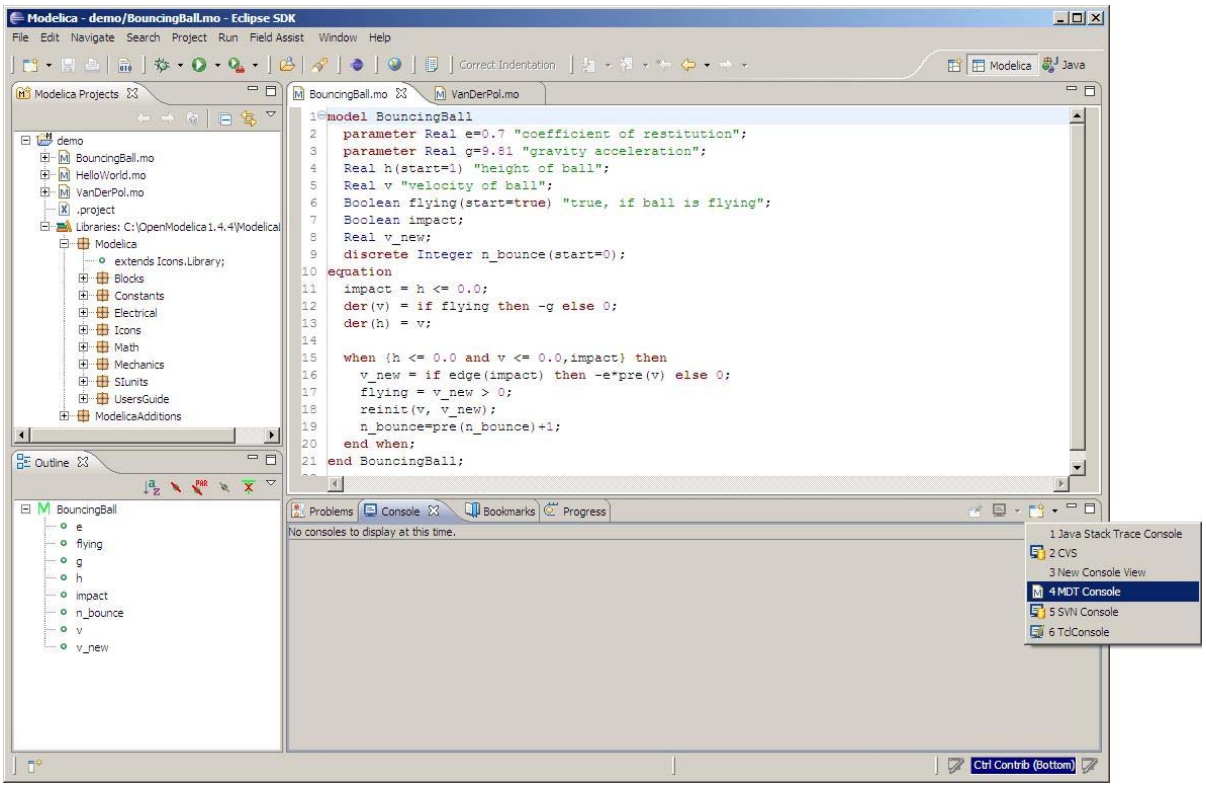


Figure 8-15. Activate the MDT Console

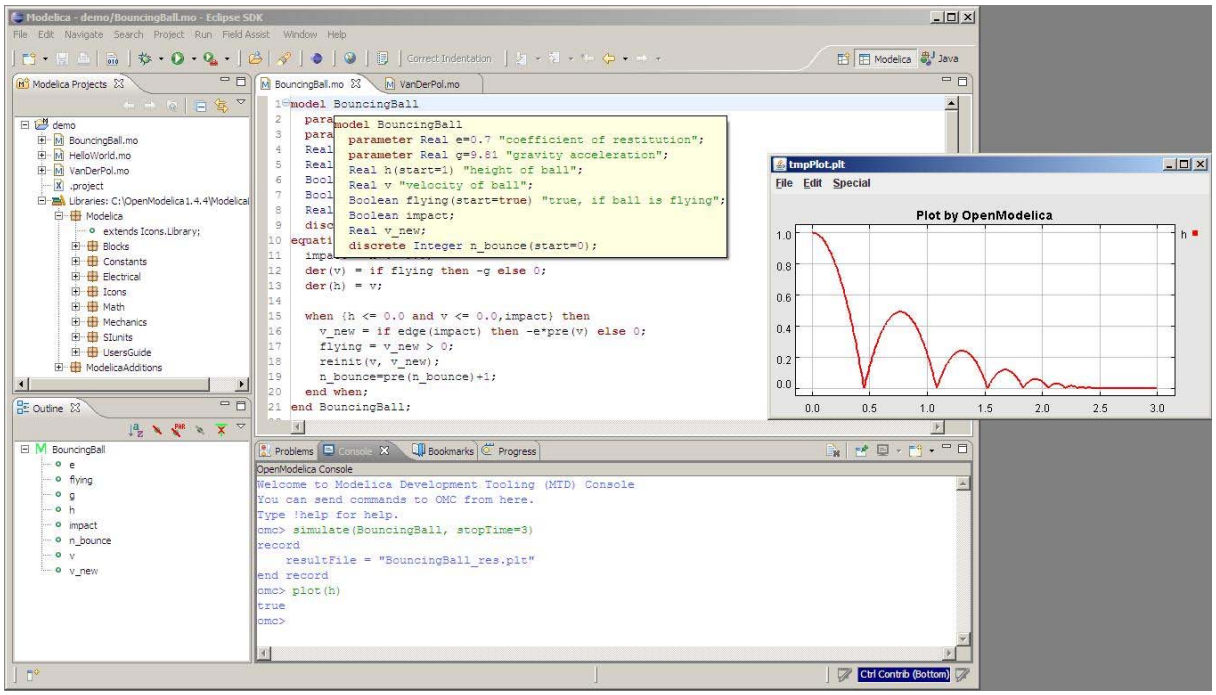


Figure 8-16. Simulation from MDT Console

Chapter 9

Modelica Performance Analyzer

A common problem when simulating models in an equation-based language like Modelica is that the model may contain non-linear equation systems. These are solved in each time-step by extrapolating an initial guess and running a non-linear system solver. If the simulation takes too long to simulate, it is useful to run the performance analysis tool. The tool has around 5~25% overhead, which is very low compared to instruction-level profilers (30x-100x overhead). Due to being based on a single simulation run, the report may contain spikes in the charts.

When running a simulation for performance analysis, execution times of user-defined functions as well as linear, non-linear and mixed equation systems are recorded.

To start a simulation in this mode, just use the `measureTime` flag of the simulate command.

```
simulate(modelname, measureTime = true)
```

The generated report is in HTML format (with images in the SVG format), stored in a file `modelname_prof.html`, but the XML database and measured times that generated the report and graphs are also available if you want to customize the report for comparison with other tools.

Below we use the performance profiler on the simple model A:

```
model A
   function f
     input Real r;
     output Real o := sin(r);
   end f;
   String s = "abc";
   Real x = f(x) "This is x";
   Real y(start=1);
   Real z1 = cos(z2);
   Real z2 = sin(z1);
equation
   der(y) = time;
end A;
```

We simulate as usual, but set `measureTime=true` to activate the profiling:

```
simulate(A, measureTime = true)
```

```
  // // record SimulationResult
  //  resultFile = "A_res.mat",
  //  messages = "Time measurements are stored in A_prof.html (human-readable)
and A_prof.xml (for XSL transforms or more details)"
  // end SimulationResult;
```

9.1 Example Report Generated for the A Model

9.1.1 Information

All times are measured using a real-time wall clock. This means context switching produces bad worst-case execution times (max times) for blocks. If you want better results, use a CPU-time clock or run the command using real-time priviliges (avoiding context switches).

Note that for blocks where the individual execution time is close to the accuracy of the real-time clock, the maximum measured time may deviate a lot from the average.

For more details, see the generated file A_prof.xml, shown in Section 9.1.7 below.

9.1.2 Settings

The settings for the simulation are summarized in the table below:

Name	Value
Integration method	euler
Output format	mat
Output name	A_res.mat
Output size	24.0 kB
Profiling data	A_prof.data
Profiling size	27.3 kB

9.1.3 Summary

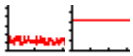
Execution times for different activities:

Task	Time	Fraction
Pre-Initialization	0.000401	19.17%
Initialization	0.000046	2.20%
Event-handling	0.000036	1.72%
Creating output file	0.000264	12.62%
Linearization	0.000000	0.00%
Time steps	0.001067	51.00%
Overhead	0.000273	13.05%
Unknown	0.000406	0.24%
Total simulation time	0.002092	100.00%

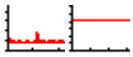
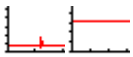
9.1.4 Global Steps

	Steps	Total Time	Fraction	Average Time	Max Time	Deviation
	499	0.001067	51.00%	2.13827655310621e-06	0.000006611	2.09x

9.1.5 Measured Function Calls

	Name	Calls	Time	Fraction	Max Time	Deviation
	A.f	1506	0.000092990	4.45%	0.000000448	6.26x

9.1.6 Measured Blocks

	Name	Calls	Time	Fraction	Max Time	Deviation
	residualFunc3	2018	0.000521137	24.91%	0.000035456	136.30x
	residualFunc1	1506	0.000393709	18.82%	0.000002735	9.46x

9.1.6.1 Equations

Name	Variables
SES_ALGORITHM 0	
SES_SIMPLE_ASSIGN 1	der(y)
residualFunc3	z2, z1
residualFunc1	x

9.1.6.2 Variables

Name	Comment
y	
der(y)	
x	This is x
z1	
z2	

S	

9.1.7 Genenerated XML for the Example

```
<!DOCTYPE doc (View Source for full doctype...)>
- <simulation>
- <modelinfo>
  <name>A</name>
  <prefix>A</prefix>
  <date>2011-03-07 12:55:53</date>
  <method>euler</method>
  <outputFormat>mat</outputFormat>
  <outputFilename>A_res.mat</outputFilename>
  <outputFilesize>24617</outputFilesize>
  <overheadTime>0.000273</overheadTime>
  <preinitTime>0.000401</preinitTime>
  <initTime>0.000046</initTime>
  <eventTime>0.000036</eventTime>
  <outputTime>0.000264</outputTime>
  <linearizeTime>0.000000</linearizeTime>
  <totalTime>0.002092</totalTime>
  <totalStepsTime>0.001067</totalStepsTime>
  <numStep>499</numStep>
  <maxTime>0.000006611</maxTime>
  </modelinfo>
- <profilingdataheader>
  <filename>A_prof.data</filename>
  <filesize>28000</filesize>
- <format>
  <uint32>step</uint32>
  <double>time</double>
  <double>cpu time</double>
  <uint32>A.f (calls)</uint32>
  <uint32>residualFunc3 (calls)</uint32>
  <uint32>residualFunc1 (calls)</uint32>
  <double>A.f (cpu time)</double>
  <double>residualFunc3 (cpu time)</double>
  <double>residualFunc1 (cpu time)</double>
  </format>
  </profilingdataheader>
- <variables>
- <variable id="1000" name="y" comment="">
  <info filename="a.mo" startline="8" startcol="3" endline="8" endcol="18"
readonly="writable" />
  </variable>
- <variable id="1001" name="der(y)" comment="">
  <info filename="a.mo" startline="8" startcol="3" endline="8" endcol="18"
readonly="writable" />
  </variable>
- <variable id="1002" name="x" comment="This is x">
  <info filename="a.mo" startline="7" startcol="3" endline="7" endcol="28"
readonly="writable" />
  </variable>
- <variable id="1003" name="z1" comment="">
  <info filename="a.mo" startline="9" startcol="3" endline="9" endcol="20"
readonly="writable" />
  </variable>
- <variable id="1004" name="z2" comment="">
  <info filename="a.mo" startline="10" startcol="3" endline="10" endcol="20"
readonly="writable" />
  </variable>
```

```
- <variable id="1005" name="s" comment="">
  <info filename="a.mo" startline="6" startcol="3" endline="6" endcol="19"
readonly="writable" />
  </variable>
  </variables>
- <functions>
- <function id="1006">
  <name>A.f</name>
  <ncall>1506</ncall>
  <time>0.000092990</time>
  <maxTime>0.000000448</maxTime>
  <info filename="a.mo" startline="2" startcol="3" endline="5" endcol="8"
readonly="writable" />
  </function>
  </functions>
- <equations>
- <equation id="1007" name="SES_ALGORITHM 0">
  <refs />
  </equation>
- <equation id="1008" name="SES_SIMPLE_ASSIGN 1">
- <refs>
  <ref refid="1001" />
  </refs>
  </equation>
- <equation id="1009" name="residualFunc3">
- <refs>
  <ref refid="1004" />
  <ref refid="1003" />
  </refs>
  </equation>
- <equation id="1010" name="residualFunc1">
- <refs>
  <ref refid="1002" />
  </refs>
  </equation>
  </equations>
- <profileblocks>
- <profileblock>
  <ref refid="1009" />
  <ncall>2018</ncall>
  <time>0.000521137</time>
  <maxTime>0.000035456</maxTime>
  </profileblock>
- <profileblock>
  <ref refid="1010" />
  <ncall>1506</ncall>
  <time>0.000393709</time>
  <maxTime>0.000002735</maxTime>
  </profileblock>
  </profileblocks>
  </simulation>
```

Chapter 10

Modelica Algorithmic Subset Debugger

This chapter presents a comprehensive Modelica debugger for an extended algorithmic subset of the Modelica language called MetaModelica. 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, stepping, inspecting variables, etc. The debugger is integrated with Eclipse.

10.1 The Eclipse-based Debugging Environment

The debugging framework is based on the Eclipse environment and is implememented as a set of plugins which are available from Modelica Development Tooling (MDT) environment. Some of the debugger functionality is presented below. In the right part a variable value is explored. In the top-left part the stack trace is presented. In the middle-left part the execution point is presented.

The debugger provides the following general functionalities:

- Adding/Removing breakpoints.
- Step Over – moves to the next line, skipping the function calls.
- Step In – takes the user into the function call.
- Step Return – complete the execution of the function and takes the user back to the point from where the function is called.
- Suspend – interrupts the running program.

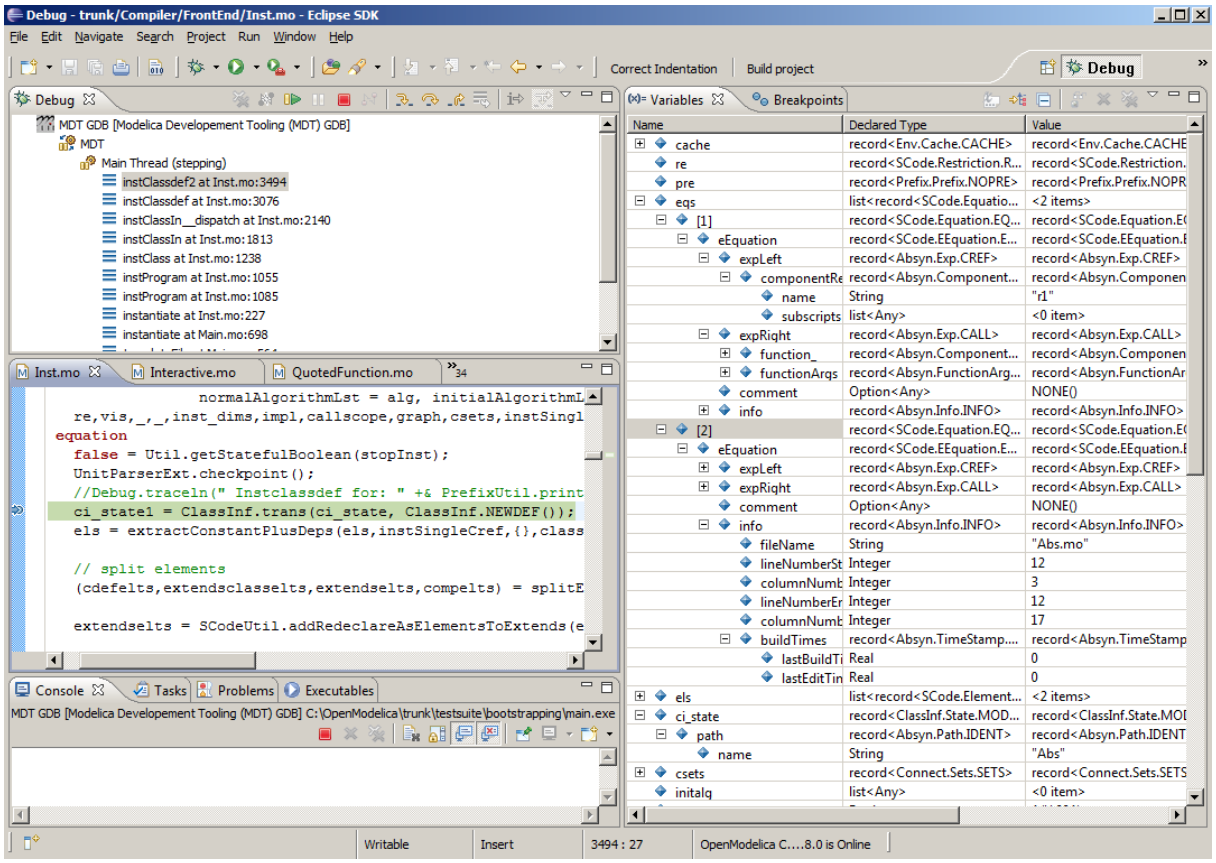


Figure 10-1. Debugging functionality.

10.2 Starting the Modelica Debugging Perspective

To be able to run in debug mode, one has to go through the following steps:

- create a mos file
- setting the debug configuration
- setting breakpoints
- running the debug configuration

All these steps are presented below using images.

10.2.1 Create mos file

In order to debug Modelica code we need to load the Modelica files into the OpenModelica Compiler. For this we can write a small script file like this,

```
setCommandLineOptions({"+d=rml,noevalfunc","+g=MetaModelica"});
setEnvironmentVar("MODELICAUSERCFLAGS","-g");
loadFile("HelloWorld.mo");
getErrorString();
HelloWorld(120.0);
getErrorString();
// Result:
// {true,true}
// true
// true
```

```
// ""
// endResult
```

So lets say that we want to debug `HelloWorld.mo`. For that we must load it into the compiler using the script file. Put all the Modelica files there in the script file to be loaded. We should also initiate the debugger by calling the starting function, in the above code `HelloWorld(120.0);`

10.2.2 Setting the debug configuration

While the Modelica perspective is activated the user should click on the bug icon on the toolbar and select Debug in order to access the dialog for building debug configurations.

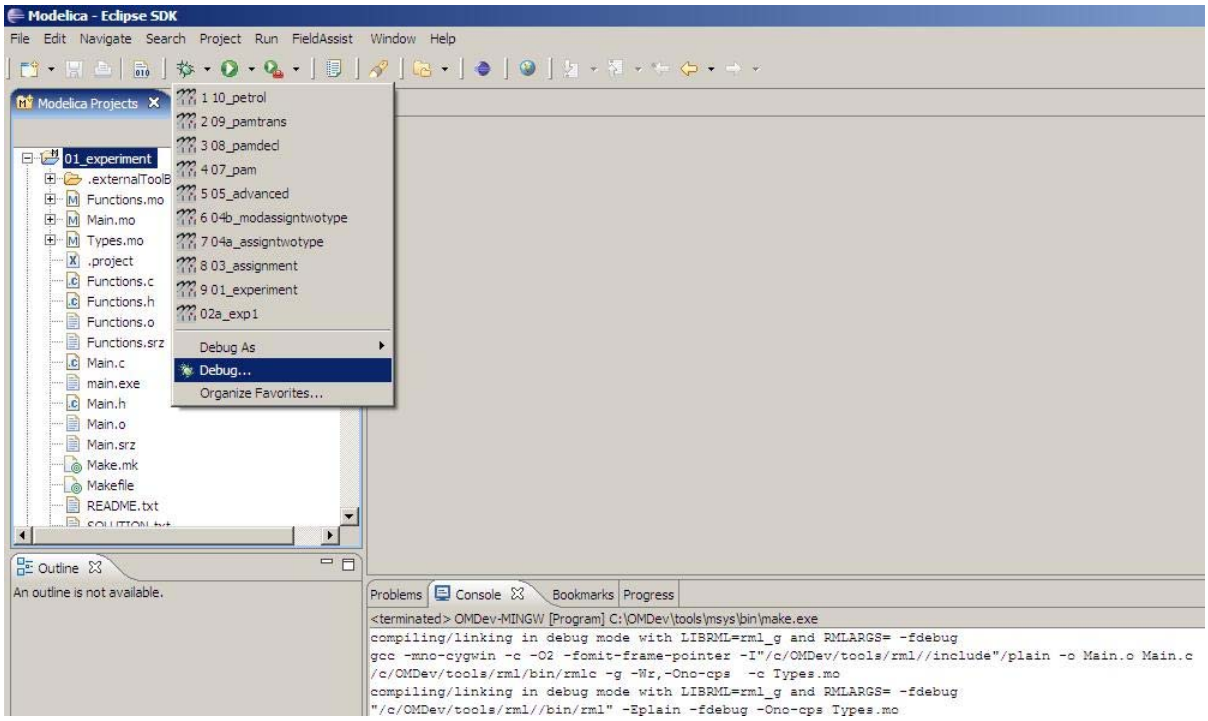


Figure 10-2. Accessing the debug configuration dialog.

To create the debug configuration, right click on the classification `Modelica Development Tooling (MDT) GDB` and select `New` as in figure below. Then give a name to the configuration, select the debugging executable to be executed and give it command line parameters. There are several tabs in which the user can select additional debug configuration settings like the environment in which the executable should be run.

Note that we require `Gnu Debugger (GDB)` for debugging session. We must specify the `GDB` location, also we must pass our script file as an argument to `OMC.`

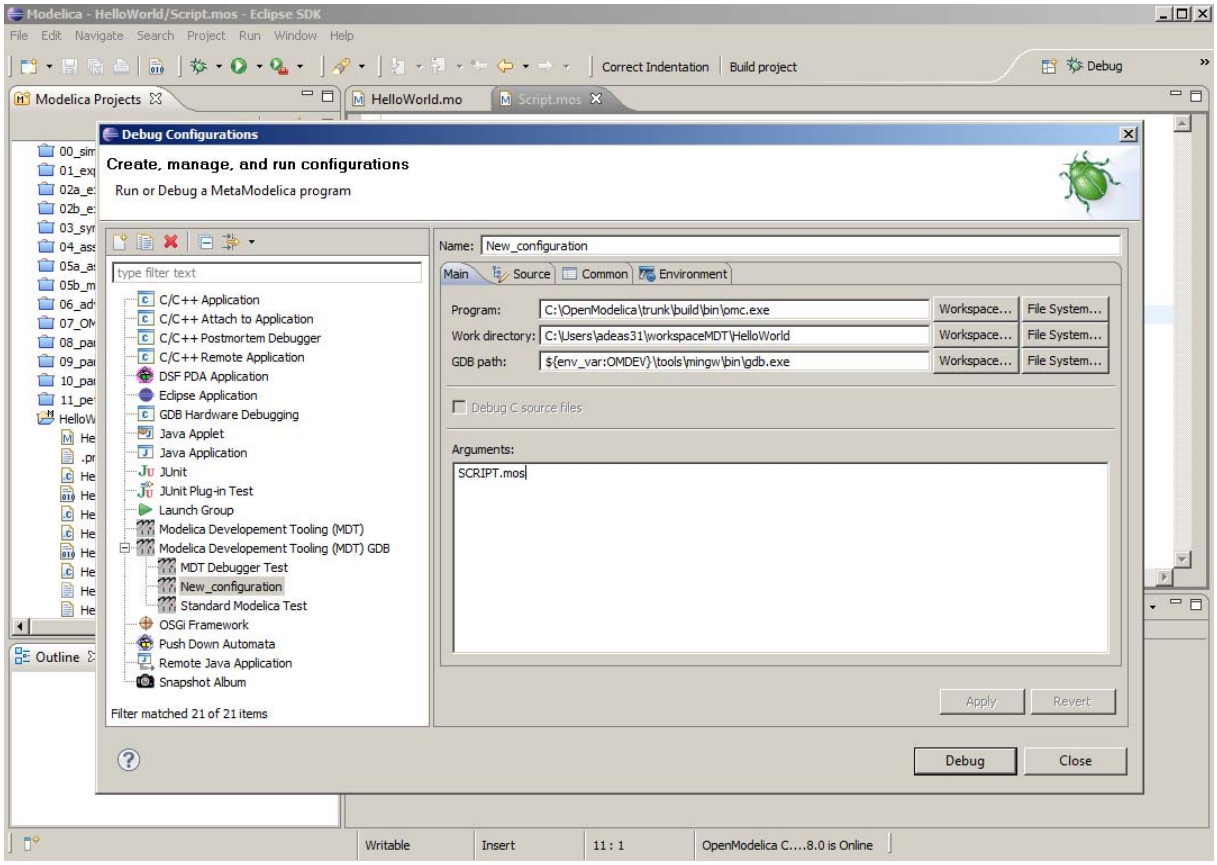


Figure 10-3. Creating the Debug Configuration.

10.2.3 Setting/Deleting Breakpoints

The Eclipse interface allows to add/remove breakpoints. At the moment only line number based breakpoints are supported. Other alternative to set the breakpoints is; function breakpoints.

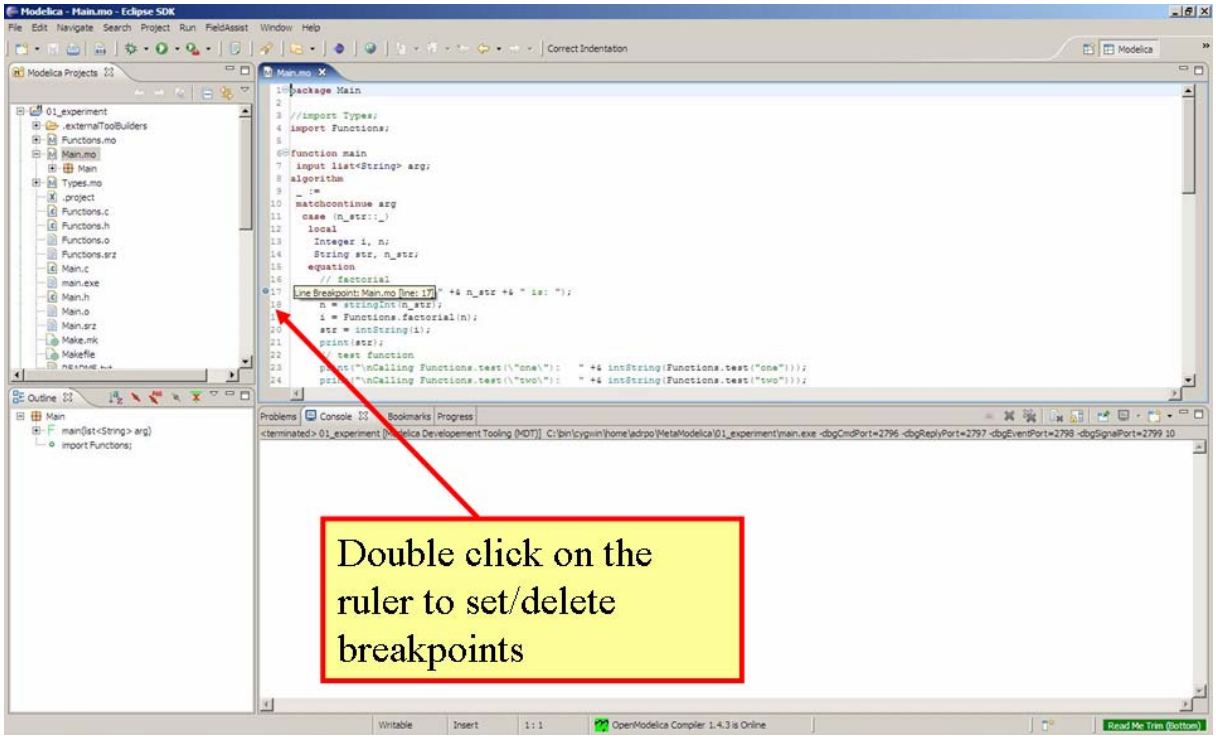


Figure 10-4. Setting/deleting breakpoints.

10.2.4 Starting the debugging session and enabling the debug perspective

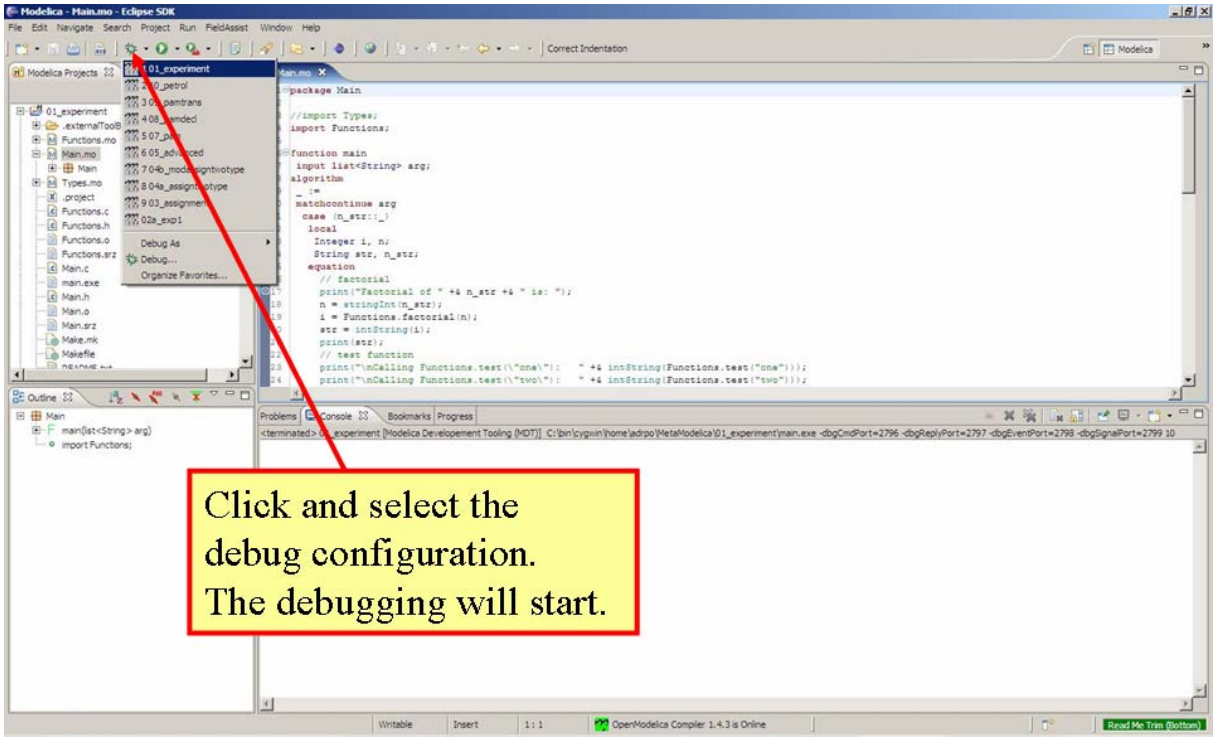


Figure 10-5. Starting the debugging session.

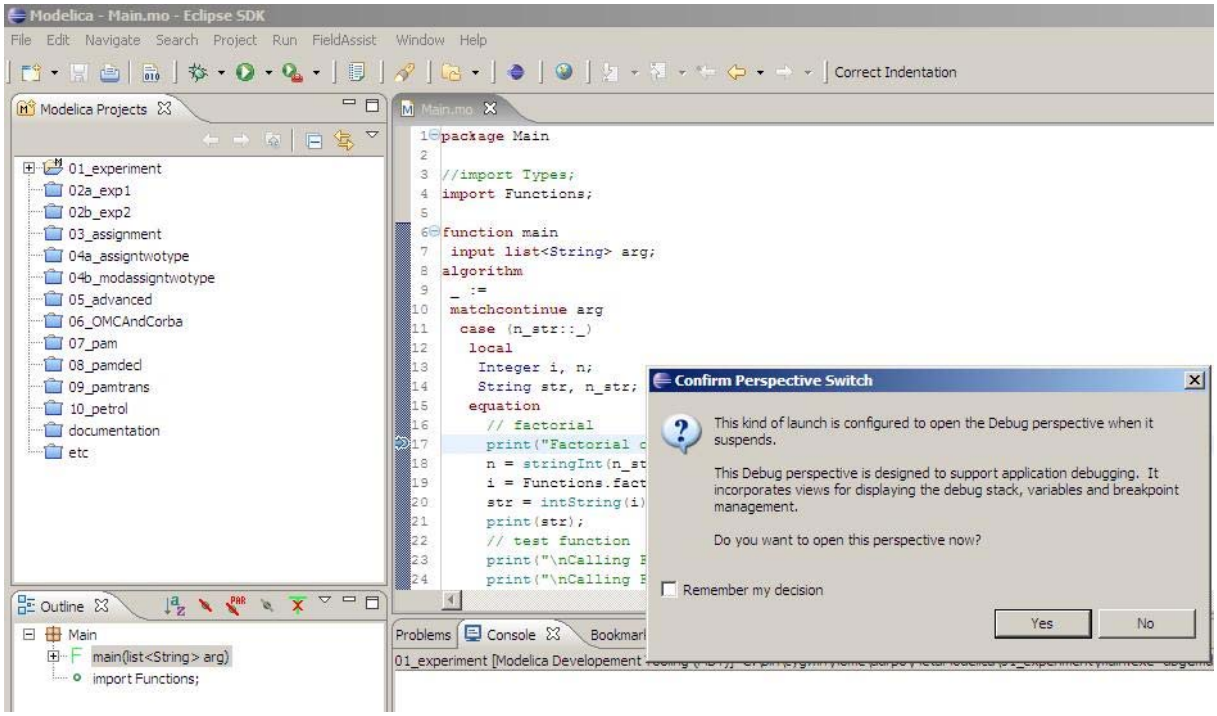


Figure 10-6. Eclipse will ask if the user wants to switch to the debugging perspective.

10.3 The Debugging Perspective

The debug view primarily consists of two main views:

- Stack Frames View
- Variables View

The stack frame view, shown in the figure below, shows a list of frames that indicates how the flow had moved from one function to another or from one file to another. This allows backtracing of the code. It is very much possible to select the previous frame in the stack and inspect the values of the variables in that frame. However, it is not possible to select any of the previous frame and start debugging from there. Each frame is shown as `<function_name at file_name:line_number>`.

The Variables view shows the list of variables at a certain point in the program, containing four colums:

- Name – the variable name.
- Declared Type – the Modelica type of the variable.
- Value – the variable value.
- Actual Type – the mapped C type.

By preserving the stack frames and variables it is possible to keep track of the variables values. If the value of any variable is changed while stepping then that variable will be highlighted yellow (the standard Eclipse way of showing the change).

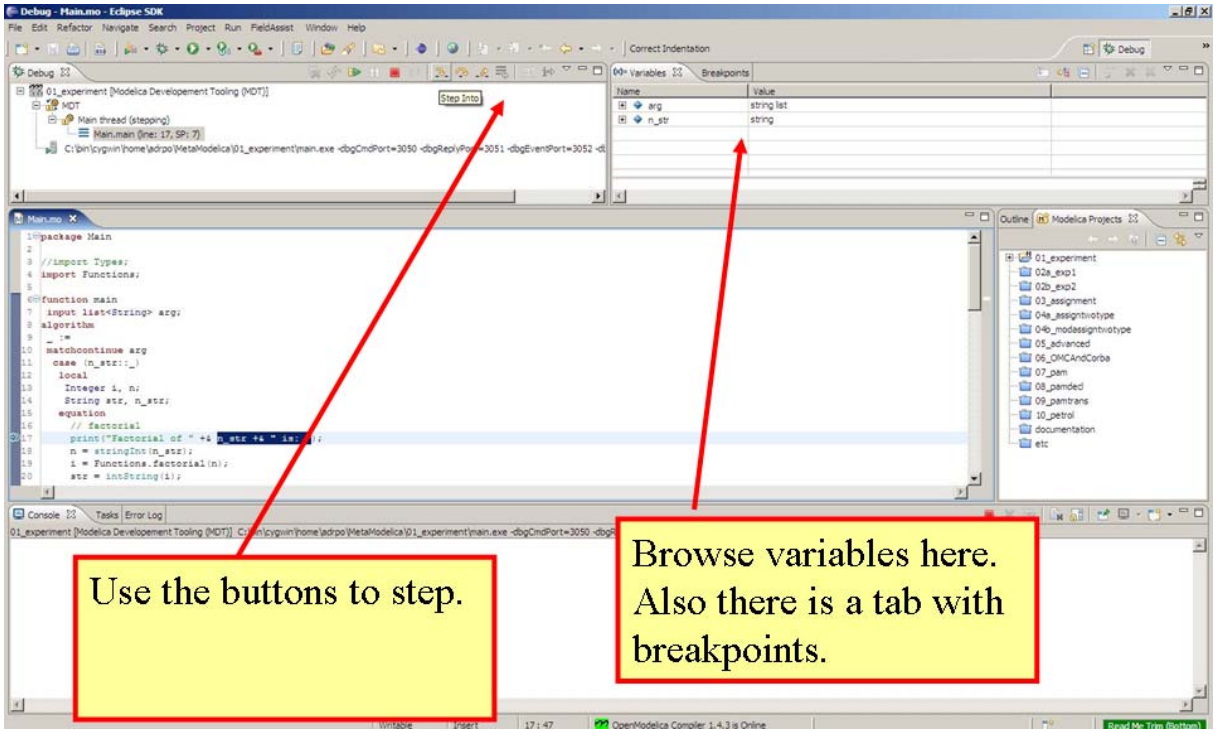


Figure 10-7. The debugging perspective.

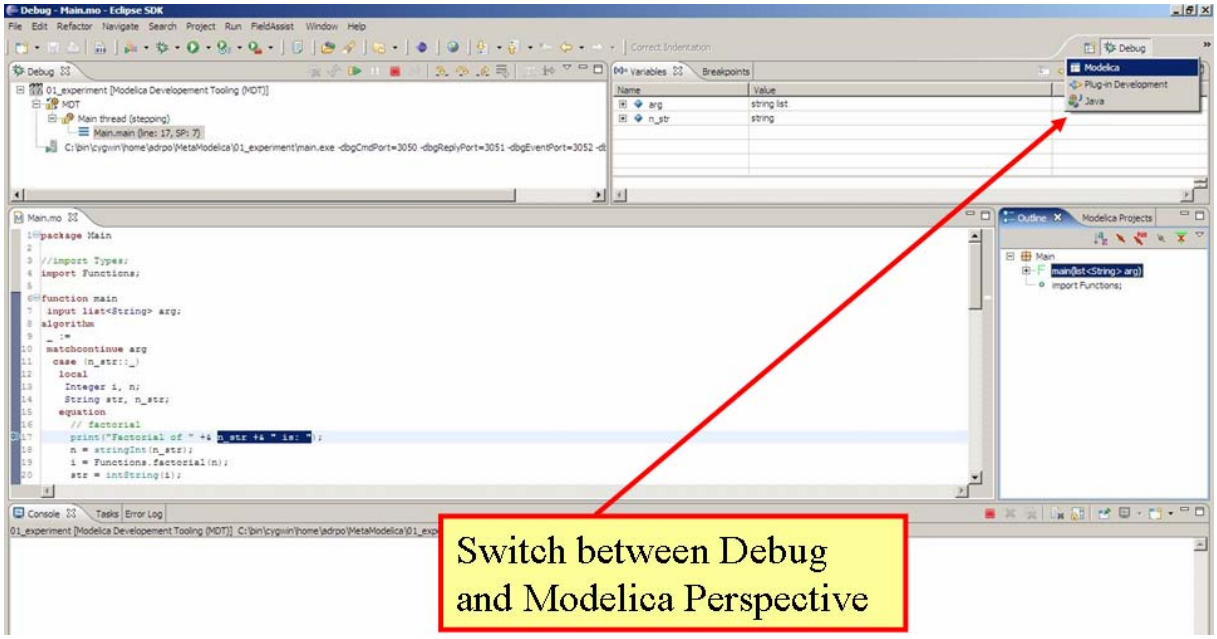


Figure 10-8. Switching between perspectives.

Chapter 11

Interoperability – C, Java, and Python

Below is information and examples about the OpenModelica external C and Java interfaces, as well as examples of Python interoperability.

11.1 Calling External C functions

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

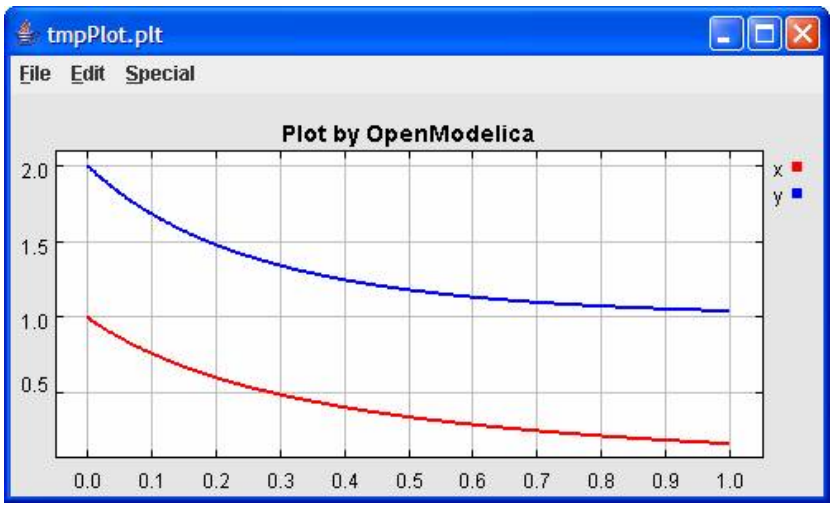
```
loadFile("ExternalLibraries.mo");
system("gcc -c -o libExternalFunc1_ext.o ExternalFunc1.c");
system("gcc -c -o libExternalFunc2.a ExternalFunc2.c");
simulate(ExternalLibraries);
```

We run the script:

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

and plot the results:

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



11.2 Calling External Java Functions

There exists a bidirectional OpenModelica-Java CORBA interface, which is capable of passing both standard Modelica data types, as well as abstract syntax trees and list structures to and from Java and process them in either Java or the OpenModelica Compiler.

The following is a small example (`ExternalJavaLib.mo`) to show the use of external Java function calls in Modelica, i.e., only the case calling Java from Modelica:

```
model ExternalJavaLib
  Real x(start=1.0);
equation
  der(x)=- ExternalJavaLog(x);
end ExternalJavaLib;
```

```
function ExternalJavaLog
  input Real x;
  output Real y;
external "Java" y='java.lang.Math.log'(x) annotation(JavaMapping = "simple");
end ExternalJavaLog;
```

The datatypes are mapped according to the tables below. There is one mapping for interacting with existing Java code (simple), and a default mapping that handles all OpenModelica datatypes. The definitions of the default datatypes exist in the Java package org.openmodelica (see $OPENMODELICA-HOME/share/java/modelica_java.jar).

For more complete examples on how to use the Java interface, download the OpenModelica source code and view the examples in testsuite/java.

Modelica	**Default Mapping**	**JavaMapping = "simple"**
Real	ModelicaReal	double
Integer	ModelicaInteger	int
Boolean	ModelicaBoolean	bool
String	ModelicaString	String
Record	ModelicaRecord	
T[:]	ModelicaArray<T>	

MetaModelica	**Default Mapping**
list<T>	ModelicaArray<T>
tuple<T1, ... , Tn>	ModelicaTuple
Option<T>	ModelicaOption<T>
Uniontype	IModelicaRecord

11.3 Python Interoperability

The interaction with Python can be perfomed in four different ways whereas one is illustrated below. Assume that we have the following Modelica code (`CalledbyPython.mo`):

```
model CalledbyPython
  Real x(start=1.0),y(start=2.0);
  parameter Real b = 2.0;
equation
  der(x) = -b*y;
  der(y) = x;
end CalledbyPython;
```

In the following Python (.py) files the above Modelica model is simulated via the OpenModelica scripting interface.

```
# file: PythonCaller.py
#!/usr/bin/python
import sys,os
global newb = 0.5
os.chdir(r'C:\Users\Documents\python')
execfile('CreateMosFile.py')
os.popen(r"C:\OpenModelica1.4.5\bin\omc.exe CalledbyPython.mos").read()
```

```
execfile('RetrResult.py')

# file: CreateMosFile.py
#!/usr/bin/python
mos_file = open('CalledbyPython.mos','w',1)
mos_file.write("loadFile(\"CalledbyPython.mo\");\n")
mos_file.write("setComponentModifierValue(CalledbyPython,b,Code(="+str(newb)+")
             );\n")
mos_file.write("simulate(CalledbyPython,stopTime=10);\n")
mos_file.close()

# file: RetrResult.py
#!/usr/bin/python
def zeros(n): #
  vec = [0.0]
  for i in range(int(n)-1): vec = vec + [0.0]
  return vec
res_file = open("CalledbyPython_res.plt",'r',1)
line = res_file.readline()
size = int(res_file.readline().split('=')[1])
time = zeros(size)
y    = zeros(size)
while line != ['DataSet: time\n']: line = res_file.readline().split(',')[0:1]
for j in range(int(size)): time[j]=float(res_file.readline().split(',')[0])
while line != ['DataSet: y\n']: line=res_file.readline().split(',')[0:1]
for j in range(int(size)):  y[j]=float(res_file.readline().split(',')[1])
res_file.close()
```

A second option of simulating the above Modelica model is to use the command `buildModel` instead of the `simulate` command and setting the parameter value in the initial parameter file, `CalledbyPython_init.txt` instead of using the command `setComponentModifierValue`. Then the file `CalledbyPython.exe` is just executed.

The third option is to use the Corba interface for invoking the compiler and then just use the scripting interface to send commands to the compiler via this interface.

The fourth variant is to use external function calls to directly communicate with the executing simulation process.

Chapter 12

Frequently Asked Questions (FAQ)

Below are some frequently asked questions in three areas, with associated answers.

12.1 OpenModelica General

- Q: Why are not the Media and Fluid libraries included in the OpenModelica distribution.
- A: These libraries need special features in the Modelica language which are not yet implemented in OpenModelica. We are working on it, but it will take some time.
- Q: OpenModelica does not read the MODELICAPATH environment variable, even though this is part of the Modelica Language Specification.
- A: Use the OPENMODELICALIBRARY environment variable instead. We have temporarily switched to this variable, in order not to interfere with other Modelica tools which might be installed on the same system. In the future, we might switch to a solution with a settings file, that also allows the user to turn on the MODELICAPATH functionality if desired.
- Q: How do I enter multi-line models into OMShell since it evaluates when typing the Enter/Return key?
- A: There are basically three methods: 1) load the model from a file using the pull-down menu or the loadModel command. 2) Enter the model/function as one (possibly long) line. 3) Type in the model in another editor, where using multiple lines is no problem, and copy/paste the model into OMShell as one operation, then push Enter. Another option is to use OMNotebook instead to enter and evaluate models.

12.2 OMNotebook

- Q: OMNotebook hangs, what to do?
- A: It is probably waiting for the omc.exe (compiler) process. (Under windows): Kill the processes omc.exe, g++.exe (C-compiler), as.exe (assembler), if present. If OMNotebook then asks whether to restart OMC, answer yes. If not, kill the process OMNotebook.exe and restart manually.
- Q: I always get simulation failed, and plotting does not work..
- A: This is cause by problems compiling and linking the generated simulation code with the MINGW (Gnu) C compiler under Windows. You probably have some Logitech software installed

that prevents Corba communication to start the compilation. There is a known bug/incompatibility in Logitech products. For example, if **lvprcsrv**.**exe** is running, **kill it** and/or prevent it to start again at reboot; it does not do anything really useful, not needed for operation of web cameras or mice.

-
- Q: After a previous session, when starting OMNotebook again, I get a strange message.
- A: You probably quit the previous OpenModelica session in the wrong way, which left the process omc.exe running. Kill that process, and try starting OMNotebook again.
- Q: I copy and paste a graphic figure from Word or some other application into OMNotebook, but the graphic does not appear. What is wrong?
- A: OMNotebook supports the graphic picture formats supported by Qt 4, including the .png, .bmp (bitmap) formats, but not for example the gif format. Try to convert your picture into one of the supported formats, (e.g. in Word, first do paste as bitmap format), and then copy the converted version into a text cell in OMNotebook.
- Q: Plotting does not work in OMNotebook.
- A: You probably have an old version of Java installed. Update your installation, and try again. (Another known problem, soon to be fixed, is that plotting of parameters and constants does not yet work).
- Q: I select a cell, copy it (e.g. Ctrl-C), and try to paste it at another place in the notebook. However, this does not work. Instead some other text that I earlier put on the clipboard is pasted into the nearest text cell.
- A: The problem is wrong choice of cursor mode, which can be text insertion or cell insertion. If you click inside a cell, the cursor become vertical, and OMNotebook expects you to paste text inside the cell. To paste a cell, you must be in cell insertion mode, i.e., click between two cells (or after a cell), you will get a vertical line. Place the cursor carefully on that vertical line until you see a small horizontal cursor. Then you should past the cell.
- Q: I am trying to click in cells to place the vertical character cursor, but it does not seem to react.
- A: This seems to be a Qt feature. You have probably made a selection (e.g. for copying) in the output section of an evaluation cell. This seems to block cursor position. Click again in the output section to disable the selection. After that it will work normally.
- Q: I have copied a text cell and start writing at the beginning of the cell. Strangely enough, the font becomes much smaller than it should be.
- A: This seems to be a Qt feature. Keep some of the old text and start writing the new stuff inside the text, i.e., at least one character position to the right. Afterwards, delete the old text at the beginning of the cell.

12.3 OMDev - OpenModelica Development Environment

- Q: I get problems compiling and linking some files when using OMDev with the MINGW (Gnu) C compiler under Windows.
- A: You probably have some Logitech software installed. There is a known bug/incompatibility in Logitech products. For example, if lvprcsrv.exe is running, kill it and/or prevent it to start again at reboot; it does not do anything really useful, not needed for operation of web cameras or mice.

Appendix A

Major OpenModelica Releases

This Appendix lists the most important OpenModelica releases and a brief description of their contents. Right now the versions from 1.3.1 to 1.8 are described.

A.1 OpenModelica 1.8, November 2011

The OpenModelica 1.8 release contains OMC flattening improvements for the Media library – it now flattens the whole library and simulates about 20% of its example models. Moreover, about half of the Fluid library models also flatten. This release also includes two new tool functionalities – the FMI for model exchange import and export, and a new efficient Eclipse-based debugger for Modelica/MetaModelica algorithmic code.

A.1.1 OpenModelica Compiler (OMC)

This release includes bug fixes and improvements of the flattening frontend part of the OpenModelica Compiler (OMC) and several improvements of the backend, including, but not restricted to:

- A faster and more stable OMC model compiler. The 1.8.0 version flattens and simulates more models than the previous 1.7.0 version.
- Flattening of the whole Media library, and about half of the Fluid library. Simulation of approximately 20% of the Media library example models.
- Functional Mockup Interface FMI 1.0 for model exchange, export and import.
- Bug fixes in the OpenModelica graphical model connection editor OMEdit, supporting easy-to-use graphical drag-and-drop modeling and MSL 3.1.
- Bug fixes in the OMOptim optimization subsystem.
- Beta version of compiler support for a new Eclipse-based very efficient algorithmic code debugger for functions in MetaModelica/Modelica, available in the development environment when using the bootstrapped OpenModelica compiler.
- Improvements in initialization of simulations.
- Improved index reduction with dynamic state selection, which improves simulation.
- Better error messages from several parts of the compiler, including a new API call for giving better error messages.
- Automatic partitioning of equation systems and multi-core parallel simulation of independent parts based on the shared-memory OpenMP model. This version is a preliminary experimental version without load balancing.

A.1.2 OpenModelica Notebook (OMNotebook)

No changes.

A.1.3 OpenModelica Shell (OMShell)

Small performance improvements.

A.1.4 OpenModelica Eclipse Plug-in (MDT)

Small fixes and improvements. MDT now also includes a beta version of a new Eclipse-based very efficient algorithmic code debugger for functions in MetaModelica/Modelica.

A.1.5 OpenModelica Development Environment (OMDev)

Third party binaries, including Qt libraries and executable Qt clients, are now part of the OMDev package. Also, now uses GCC 4.4.0 instead of the earlier GCC 3.4.5.

A.1.6 Graphic Editor OMEdit

Bug fixes. Access to FMI Import/Export through a pull-down menu. Improved configuration of library loading. A function to go to a specific line number. A button to cancel an on-going simulation. Support for some updated OMC API calls.

A.1.7 New OMOptim Optimization Subsystem

Bug fixes, especially in the Linux version.

A.1.8 FMI Support

The Functional Mockup Interface FMI 1.0 for model exchange import and export is supported by this release. The functionality is accessible via API calls as well as via pull-down menu commands in OMEdit.

A.2 OpenModelica 1.7, April 2011

The OpenModelica 1.7 release contains OMC flattening improvements for the Media library, better and faster event handling and simulation, and fast MetaModelica support in the compiler, enabling it to compiler itself. This release also includes two interesting new tools – the OMOpttim optimization subsystem, and a new performance profiler for equation-based Modelica models.

A.2.1 OpenModelica Compiler (OMC)

This release includes bug fixes and performance improvements of the flattening frontend part of the OpenModelica Compiler (OMC) and several improvements of the backend, including, but not restricted to:

- Flattening of the whole Modelica Standard Library 3.1 (MSL 3.1), except Media and Fluid.
- Progress in supporting the Media library, some models now flatten.
- Much faster simulation of many models through more efficient handling of alias variables, binary output format, and faster event handling.
- Faster and more stable simulation through new improved event handling, which is now default.
- Simulation result storage in binary .mat files, and plotting from such files.
- Support for Unicode characters in quoted Modelica identifiers, including Japanese and Chinese.
- Preliminary MetaModelica 2.0 support. (use setCommandLineOptions({"+g=MetaModelica"})). Execution is as fast as MetaModelica 1.0, except for garbage collection.
- Preliminary bootstrapped OpenModelica compiler: OMC now compiles itself, and the bootstrapped compiler passes the test suite. A garbage collector is still missing.
- Many bug fixes.

A.2.2 OpenModelica Notebook (OMNotebook)

Improved much faster and more stable 2D plotting through the new OMPlot module. Plotting from binary .mat files. Better integration between OMEdit and OMNotebook, copy/paste between them.

A.2.3 OpenModelica Shell (OMShell)

Same as previously, except the improved 2D plotting through OMPlot.

A.2.4 OpenModelica Eclipse Plug-in (MDT)

Same as previously.

A.2.5 OpenModelica Development Environment (OMDev)

No changes.

A.2.6 Graphic Editor OMEdit

Several enhancements of OMEdit are included in this release. Support for Icon editing is now available. There is also an improved much faster 2D plotting through the new OMPlot module. Better integration between OMEdit and OMNotebook, with copy/paste between them. Interactive on-line simulation is available in an easy-to-use way.

A.2.7 New OMOptim Optimization Subsystem

A new optimization subsystem called OMOptim has been added to OpenModelica. Currently, parameter optimization using genetic algorithms is supported in this version 0.9. Pareto front optimization is also supported.

A.2.8 New Performance Profiler

A new, low overhead, performance profiler for Modelica models has been developed.

A.3 OpenModelica 1.6, November 2010

The OpenModelica 1.6 release primarily contains flattening, simulation, and performance improvements regarding Modelica Standard Library 3.1 support, but also has an interesting new tool – the OMEdit graphic connection editor, and a new educational material called DrControl, and an improved ModelicaML UML/Modelica profile with better support for modeling and requirement handling.

A.3.1 OpenModelica Compiler (OMC)

This release includes bug fix and performance improvemetns of the flattening frontend part of the OpenModelica Compiler (OMC) and some improvements of the backend, including, but not restricted to:

- Flattening of the whole Modelica Standard Library 3.1 (MSL 3.1), except Media and Fluid.
- Improved flattening speed of a factor of 5-20 compared to OpenModelica 1.5 for a number of models, especially in the MultiBody library.
- Reduced memory consumption by the OpenModelica compiler frontend, for certain large models a reduction of a factor 50.
- Reorganized, more modular OpenModelica compiler backend, can now handle approximately 30 000 equations, compared to previously approximately 10 000 equations.
- Better error messages from the compiler, especially regarding functions.
- Improved simulation coverage of MSL 3.1. Many models that did not simulate before are now simulating. However, there are still many models in certain sublibraries that do not simulate.
- Progress in supporting the Media library, but simulation is not yet possible.
- Improved support for enumerations, both in the frontend and the backend.
- Implementation of stream connectors.
- Support for linearization through symbolic Jacobians.

- Many bug fixes.

A.3.2 OpenModelica Notebook (OMNotebook)

A new DrControl electronic notebook for teaching control and modeling with Modelica.

A.3.3 OpenModelica Shell (OMShell)

Same as previously.

A.3.4 OpenModelica Eclipse Plug-in (MDT)

Same as previously.

A.3.5 OpenModelica Development Environment (OMDev)

Several enhancements. Support for match-expressions in addition to matchcontinue. Support for real if-then-else. Support for if-then without else-branches. Modelica Development Tooling 0.7.7 with small improvements such as more settings, improved error detection in console, etc.

A.3.6 New Graphic Editor OMEdit

A new improved open source graphic model connection editor called OMEdit, supporting 3.1 graphical annotations, which makes it possible to move models back and forth to other tools without problems. The editor has been implemented by students at Linköping University and is based on the C++ Qt library.

A.4 OpenModelica 1.5, July 2010

This OpenModelica 1.5 release has major improvements in the OpenModelica compiler frontend and some in the backend. A major improvement of this release is full flattening support for the MultiBody library as well as limited simulation support for MultiBody. Interesting new facilities are the interactive simulation and the integrated UML-Modelica modeling with ModelicaML. Approximately 4 person-years of additional effort have been invested in the compiler compared to the 1.4.5 version, e.g., in order to have a more complete coverage of Modelica 3.0, mainly focusing on improved flattening in the compiler frontend.

A.4.1 OpenModelica Compiler (OMC)

This release includes major improvements of the flattening frontend part of the OpenModelica Compiler (OMC) and some improvements of the backend, including, but not restricted to:

- Improved flattening speed of at least a factor of 10 or more compared to the 1.4.5 release, primarily for larger models with inner-outer, but also speedup for other models, e.g. the robot model flattens in approximately 2 seconds.
- Flattening of all MultiBody models, including all elementary models, breaking connection graphs, world object, etc. Moreover, simulation is now possible for at least five MultiBody models: Pendulum, DoublePendulum, InitSpringConstant, World, PointGravityWithPointMasses.
- Progress in supporting the Media library, but simulation is not yet possible.
- Support for enumerations, both in the frontend and the backend.
- Support for expandable connectors.
- Support for the inline and late inline annotations in functions.
- Complete support for record constructors, also for records containing other records.
- Full support for iterators, including nested ones.
- Support for inferred iterator and for-loop ranges.
- Support for the function derivative annotation.

- Prototype of interactive simulation.
- Prototype of integrated UML-Modelica modeling and simulation with ModelicaML.
- A new bidirectional external Java interface for calling external Java functions, or for calling Modelica functions from Java.
- Complete implementation of replaceable model extends.
- Fixed problems involving arrays of unknown dimensions.
- Limited support for tearing.
- Improved error handling at division by zero.
- Support for Modelica 3.1 annotations.
- Support for all MetaModelica language constructs inside OpenModelica.
- OpenModelica works also under 64-bit Linux and Mac 64-bit OSX.
- Parallel builds and running test suites in parallel on multi-core platforms.
- New OpenModelica text template language for easier implementation of code generators, XML generators, etc.
- New OpenModelica code generators to C and C# using the text template language.
- Faster simulation result data file output optionally as comma-separated values.
- Many bug fixes.

It is now possible to graphically edit models using parts from the Modelica Standard Library 3.1, since the simForge graphical editor (from Politecnico di Milano) that is used together with OpenModelica has been updated to version 0.9.0 with a important new functionality, including support for Modelica 3.1 and 3.0 annotations. The 1.6 and 2.2.1 Modelica graphical annotation versions are still supported.

A.4.2 OpenModelica Notebook (OMNotebook)

Improvements in platform availability.

- Support for 64-bit Linux.
- Support for Windows 7.
- Better support for MacOS, including 64-bit OSX.

A.4.3 OpenModelica Shell (OMShell)

Same as previously.

A.4.4 OpenModelica Eclipse Plug-in (MDT)

Minor bug fixes.

A.4.5 OpenModelica Development Environment (OMDev)

Minor bug fixes.

A.5 OpenModelica 1.4.5, January 2009

This release has several improvements, especially platform availability, less compiler memory usage, and supporting more aspects of Modelica 3.0.

A.5.1 OpenModelica Compiler (OMC)

This release includes small improvements and some bugfixes of the OpenModelica Compiler (OMC):

- Less memory consumption and better memory management over time. This also includes a better API supporting automatic memory management when calling C functions from within the compiler.

- Modelica 3.0 parsing support.
- Export of DAE to XML and MATLAB.
- Support for several platforms Linux, MacOS, Windows (2000, Xp, Vista).
- Support for record and strings as function arguments.
- Many bug fixes.
- (Not part of OMC): Additional free graphic editor SimForge can be used with OpenModelica.

A.5.2 OpenModelica Notebook (OMNotebook)

A number of improvements, primarily in the plotting functionality and platform availability.

- A number of improvements in the plotting functionality: scalable plots, zooming, logarithmic plots, grids, etc.
- Programmable plotting accessible through a Modelica API.
- Simple 3D visualization.
- Support for several platforms Linux, MacOS, Windows (2000, Xp, Vista).

A.5.3 OpenModelica Shell (OMShell)

Same as previously.

A.5.4 OpenModelica Eclipse Plug-in (MDT)

Minor bug fixes.

A.5.5 OpenModelica Development Environment (OMDev)

Same as previously.

A.1 OpenModelica 1.4.4, Feb 2008

This release is primarily a bug fix release, except for a preliminary version of new plotting functionality available both from the OMNotebook and separately through a Modelica API. This is also the first release under the open source license OSMC-PL (Open Source Modelica Consortium Public License), with support from the recently created Open Source Modelica Consortium. An integrated version handler, bug-, and issue tracker has also been added.

A.5.6 OpenModelica Compiler (OMC)

This release includes small improvements and some bugfixes of the OpenModelica Compiler (OMC):

- Better support for if-equations, also inside when.
- Better support for calling functions in parameter expressions and interactively through dynamic loading of functions.
- Less memory consumtion during compilation and interactive evaluation.
- A number of bug-fixes.

A.5.7 OpenModelica Notebook (OMNotebook)

Test release of improvements, primarily in the plotting functionality and platform availability.

- Preliminary version of improvements in the plotting functionality: scalable plots, zooming, logarithmic plots, grids, etc., currently available in a preliminary version through the plot2 function.
- Programmable plotting accessible through a Modelica API.

A.5.8 OpenModelica Shell (OMShell)

Same as previously.

A.5.9 OpenModelica Eclipse Plug-in (MDT)

This release includes minor bugfixes of MDT and the associated MetaModelica debugger:

A.5.10 OpenModelica Development Environment (OMDev)

Extended test suite with a better structure. Version handling, bug tracking, issue tracking, etc. now available under the integrated Codebeamer

A.6 OpenModelica 1.4.3, June 2007

This release has a number of significant improvements of the OMC compiler, OMNotebook, the MDT plugin and the OMDev. Increased platform availability now also for Linux and Macintosh, in addition to Windows. OMShell is the same as previously, but now ported to Linux and Mac.

A.6.1 OpenModelica Compiler (OMC)

This release includes a number of improvements of the OpenModelica Compiler (OMC):

- Significantly increased compilation speed, especially with large models and many packages.
- Now available also for Linux and Macintosh platforms.
- Support for when-equations in algorithm sections, including elsewhen.
- Support for inner/outer prefixes of components (but without type error checking).
- Improved solution of nonlinear systems.
- Added ability to compile generated simulation code using Visual Studio compiler.
- Added "smart setting of fixed attribute to false. If initial equations, OMC instead has fixed=true as default for states due to allowing overdetermined initial equation systems.
- Better state select heuristics.
- New function getIncidenceMatrix(ClassName) for dumping the incidence matrix.
- Builtin functions String(), product(), ndims(), implemented.
- Support for terminate() and assert() in equations.
- In emitted flat form: protected variables are now prefixed with protected when printing flat class.
- Some support for tables, using omcTableTimeIni instead of dymTableTimeIni2.
- Better support for empty arrays, and support for matrix operations like a*[1,2;3,4].
- Improved val() function can now evaluate array elements and record fields, e.g. val(x[n]), val(x.y) .
- Support for reinit in algorithm sections.
- String support in external functions.
- Double precision floating point precision now also for interpreted expressions
- Better simulation error messages.
- Support for der(expressions).
- Support for iterator expressions such as {3*i for i in 1..10}.
- More test cases in the test suite.
- A number of bug fixes, including sample and event handling bugs.

A.6.2 OpenModelica Notebook (OMNotebook)

A number of improvements, primarily in the platform availability.

- Available on the Linux and Macintosh platforms, in addition to Windows.
- Fixed cell copying bugs, plotting of derivatives now works, etc.

A.6.3 OpenModelica Shell (OMShell)

Now available also on the Macintosh platform.

A.6.4 OpenModelica Eclipse Plug-in (MDT)

This release includes major improvements of MDT and the associated MetaModelica debugger:

- Greatly improved browsing and code completion works both for standard Modelica and for MetaModelica.
- Hovering over identifiers displays type information.
- A new and greatly improved implementation of the debugger for MetaModelica algorithmic code, operational in Eclipse. Greatly improved performance – only approx 10% speed reduction even for 100 000 line programs. Greatly improved single stepping, step over, data structure browsing, etc.
- Many bug fixes.

A.6.5 OpenModelica Development Environment (OMDev)

Increased compilation speed for MetaModelica. Better if-expression support in MetaModelica.

A.7 OpenModelica 1.4.2, October 2006

This release has improvements and bug fixes of the OMC compiler, OMNotebook, the MDT plugin and the OMDev. OMShell is the same as previously.

A.7.1 OpenModelica Compiler (OMC)

This release includes further improvements of the OpenModelica Compiler (OMC):

- Improved initialization and index reduction.
- Support for integer arrays is now largely implemented.
- The val(variable,time) scripting function for accessing the value of a simulation result variable at a certain point in the simulated time.
- Interactive evalution of for-loops, while-loops, if-statements, if-expressions, in the interactive scripting mode.
- Improved documentation and examples of calling the Model Query and Manipulation API.
- Many bug fixes.

A.7.2 OpenModelica Notebook (OMNotebook)

Search and replace functions have been added. The DrModelica tutorial (all files) has been updated, obsolete sections removed, and models which are not supported by the current implementation marked clearly. Automatic recognition of the .onb suffix (e.g. when double-clicking) in Windows makes it even more convenient to use.

A.7.3 OpenModelica Eclipse Plug-in (MDT)

Two major improvements are added in this release:

- Browsing and code completion works both for standard Modelica and for MetaModelica.
- The debugger for algorithmic code is now available and operational in Eclipse for debugging of MetaModelica programs.

A.7.4 OpenModelica Development Environment (OMDev)

Mostly the same as previously.

A.8 OpenModelica 1.4.1, June 2006

This release has only improvements and bug fixes of the OMC compiler, the MDT plugin and the OMDev components. The OMShell and OMNotebook are the same.

A.8.1 OpenModelica Compiler (OMC)

This release includes further improvements of the OpenModelica Compiler (OMC):

- Support for external objects.
- OMC now reports the version number (via command line switches or CORBA API getVersion()).
- Implemented caching for faster instantiation of large models.
- Many bug fixes.

A.8.2 OpenModelica Eclipse Plug-in (MDT)

Improvements of the error reporting when building the OMC compiler. The errors are now added to the problems view. The latest MDT release is version 0.6.6 (2006-06-06).

A.8.3 OpenModelica Development Environment (OMDev)

Small fixes in the MetaModelica compiler. MetaModelica Users Guide is now part of the OMDev release. The latest OMDev was release in 2006-06-06.

A.9 OpenModelica 1.4.0, May 2006

This release has a number of improvements described below. The most significant change is probably that OMC has now been translated to an extended subset of Modelica (MetaModelica), and that all development of the compiler is now done in this version..

A.9.1 OpenModelica Compiler (OMC)

This release includes further improvements of the OpenModelica Compiler (OMC):

- Partial support for mixed system of equations.
- New initialization routine, based on optimization (minimizing residuals of initial equations).
- Symbolic simplification of builtin operators for vectors and matrices.
- Improved code generation in simulation code to support e.g. Modelica functions.
- Support for classes extending basic types, e.g. connectors (support for MSL 2.2 block connectors).
- Support for parametric plotting via the plotParametric command.
- Many bug fixes.

A.9.2 OpenModelica Shell (OMShell)

Essentially the same OMShell as in 1.3.1. One difference is that now all error messages are sent to the command window instead of to a separate log window.

A.9.3 OpenModelica Notebook (OMNotebook)

Many significant improvements and bug fixes. This version supports graphic plots within the cells in the notebook. Improved cell handling and Modelica code syntax highlighting. Command completion of the most common OMC commands is now supported. The notebook has been used in several courses.

A.9.4 OpenModelica Eclipse Plug-in (MDT)

This is the first really useful version of MDT. Full browsing of Modelica code, e.g. the MSL 2.2, is now supported. (MetaModelica browsing is not yet fully supported). Full support for automatic indentation of Modelica code, including the MetaModelica extensions. Many bug fixes. The Eclipse plug-in is now in use for OpenModelica development at PELAB and MathCore Engineering AB since approximately one month.

A.9.5 OpenModelica Development Environment (OMDev)

The following mechanisms have been put in place to support OpenModelica development.

- A separate web page for OMDev (OpenModelica Development Environment).
- A pre-packaged OMDev zip-file with precompiled binaries for development under Windows using the mingw Gnu compiler from the Eclipse MDT plug-in. (Development is also possible using Visual Studio).
- All source code of the OpenModelica compiler has recently been translated to an extended subset of Modelica, currently called MetaModelica. The current size of OMC is approximately 100 000 lines All development is now done in this version.
- A new tutorial and users guide for development in MetaModelica.
- Successful builds and tests of OMC under Linux and Solaris.

A.10 OpenModelica 1.3.1, November 2005

This release has several important highlights.

This is also the *first* release for which the New BSD (Berkeley) open-source license applies to the source code, including the whole compiler and run-time system. This makes is possible to use OpenModelica for both academic and commercial purposes without restrictions.

A.10.1 OpenModelica Compiler (OMC)

This release includes a significantly improved OpenModelica Compiler (OMC):

- Support for hybrid and discrete-event simulation (if-equations, if-expressions, when-equations; not yet if-statements and when-statements).
- Parsing of full Modelica 2.2
- Improved support for external functions.
- Vectorization of function arguments; each-modifiers, better implementation of replaceable, better handling of structural parameters, better support for vector and array operations, and many other improvements.
- Flattening of the Modelica Block library version 1.5 (except a few models), and simulation of most of these.
- Automatic index reduction (present also in previous release).
- Updated User's Guide including examples of hybrid simulation and external functions.

A.10.2 OpenModelica Shell (OMShell)

An improved window-based interactive command shell, now including command completion and better editing and font size support.

A.10.3 OpenModelica Notebook (OMNotebook)

A free implementation of an OpenModelica notebook (OMNOtebook), for electronic books with course material, including the DrModelica interactive course material. It is possible to simulate and plot from this notebook.

A.10.4 OpenModelica Eclipse Plug-in (MDT)

An early alpha version of the first Eclipse plug-in (called MDT for Modelica Development Tooling) for Modelica Development. This version gives compilation support and partial support for browsing Modelica package hierarchies and classes.

A.10.5 OpenModelica Development Environment (OMDev)

The following mechanisms have been put in place to support OpenModelica development.

- Bugzilla support for OpenModelica bug tracking, accessible to anybody.
- A system for automatic regression testing of the compiler and simulator, (+ other system parts) usually run at check in time.
- Version handling is done using SVN, which is better than the previously used CVS system. For example, name change of modules is now possible within the version handling system.

Appendix B

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, promotion, 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.

B.1 OpenModelica Contributors 2011

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

Adrian Pop, PELAB, Linköping University, Linköping, Sweden.
Willi Braun, Fachhochschule Bielefeld, Bielefeld, Germany.
Jens Frenkel, TU Dresden, Dresden, Germany.
Martin Sjölund, PELAB, Linköping University, Linköping, Sweden.
Per Östlund, PELAB, Linköping University, Linköping, Sweden.

Peter Aronsson, MathCore Engineering AB, Linköping, Sweden.
Adeel Asghar, PELAB, Linköping University, Linköping, Sweden.
David Akhvlediani, PELAB, Linköping University, Linköping, Sweden.
Mikael Axin, IEI, Linköping University, Linköping, Sweden.
Bernhard Bachmann, Fachhochschule Bielefeld, Bielefeld, Germany.
Vasile Baluta, PELAB, Linköping University, Linköping, Sweden.
Robert Braun, IEI, Linköping University, Linköping, Sweden.
David Broman, PELAB, Linköping University, Linköping, Sweden.
Stefan Brus, PELAB, Linköping University, Linköping, Sweden.
Francesco Casella, Politecnico di Milano, Milan, Italy.
Filippo Donida, Politecnico di Milano, Milan, Italy.
Mahder Gebremedhin, PELAB, Linköping University, Linköping, Sweden.
Pavel Grozman, Equa AB, Stockholm, Sweden.
Michael Hanke, NADA, KTH, Stockholm.
Daniel Hedberg, MathCore Engineering AB, Linköping, Sweden.
Zoheb Hossain, PELAB, Linköping University, Linköping, Sweden.
Alf Isaksson, ABB Corporate Research, Västerås, Sweden.
Kim Jansson, PELAB, Linköping University, Linköping, Sweden.
Daniel Kanth, Bosch-Rexroth, Lohr am Main, Germany.
Tommi Karhela, VTT, Espoo, Finland.
Joel Klinghed, PELAB, Linköping University, Linköping, Sweden.
Petter Krus, IEI, Linköping University, Linköping, Sweden.
Juha Kortelainen, VTT, Espoo, Finland.
Abhinn Kothari, PELAB, Linköping University, Linköping, Sweden.
Alexey Lebedev, Equa Simulation AB, Stockholm, Sweden.
Oliver Lenord, Siemens PLM, California, USA.
Ariel Liebman, Energy Users Association of Australia, Victoria, Australia.
Rickard Lindberg, PELAB, Linköping University, Linköping, Sweden

Håkan Lundvall, PELAB, Linköping University, Linköping, Sweden.
Henrik Magnusson, Linköping, Sweden.
Abhi Raj Metkar, CDAC, Trivandrum, Kerala, India.
Eric Meyers, Pratt & Whitney Rocketdyne, Palm City, Florida, USA.
Tuomas Miettinen, VTT, Espoo, Finland.
Afshin Moghadam, PELAB, Linköping University, Linköping, Sweden.
Maroun Nemer, CEP Paristech, Ecole des Mines, Paris, France.
Hannu Niemistö, VTT, Espoo, Finland.
Peter Nordin, IEI, Linköping University, Linköping, Sweden.
Kristoffer Norling, PELAB, Linköping University, Linköping, Sweden.
Lennart Ochel, Fachhochschule Bielefeld, Bielefeld, Germany.
Karl Pettersson, IEI, Linköping University, Linköping, Sweden.
Pavol Privitzer, Institute of Pathological Physiology, Praha, Czech Republic.
Reino Ruusu, VTT, Espoo, Finland.
Per Sahlin, Equa Simulation AB, Stockholm, Sweden.
Wladimir Schamai, EADS, Hamburg, Germany.
Gerhard Schmitz, University of Hamburg, Hamburg, Germany.
Klas Sjöholm, PELAB, Linköping University, Linköping, Sweden.
Anton Sodja, University of Ljubljana, Ljubljana, Slovenia
Ingo Staack, IEI, Linköping University, Linköping, Sweden.
Kristian Stavåker, PELAB, Linköping University, Linköping, Sweden.
Sonia Tariq, PELAB, Linköping University, Linköping, Sweden.
Hubert Thierot, CEP Paristech, Ecole des Mines, Paris, France.
Mohsen Torabzadeh-Tari, PELAB, Linköping University, Linköping, Sweden.
Parham Vasaiely, EADS, Hamburg, Germany.
Niklas Worschech, Bosch-Rexroth, Lohr am Main, Germany.
Robert Wotzlaw, Goettingen, Germany.
Björn Zachrisson, MathCore Engineering AB, Linköping, Sweden.
Azam Zia, PELAB, Linköping University, Linköping, Sweden.

B.2 OpenModelica Contributors 2010

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

Adrian Pop, PELAB, Linköping University, Linköping, Sweden.
Martin Sjölund, PELAB, Linköping University, Linköping, Sweden.
Per Östlund, PELAB, Linköping University, Linköping, Sweden.

Peter Aronsson, MathCore Engineering AB, Linköping, Sweden.
Adeel Asghar, PELAB, Linköping University, Linköping, Sweden.
David Akhvlediani, PELAB, Linköping University, Linköping, Sweden.
Bernhard Bachmann, Fachhochschule Bielefeld, Bielefeld, Germany.
Vasile Baluta, PELAB, Linköping University, Linköping, Sweden.
Simon Björklén, PELAB, Linköping University, Linköping, Sweden.
Mikael Blom, PELAB, Linköping University, Linköping, Sweden.
Robert Braun, IEI, Linköping University, Linköping, Sweden.
Willi Braun, Fachhochschule Bielefeld, Bielefeld, Germany.
David Broman, PELAB, Linköping University, Linköping, Sweden.
Stefan Brus, PELAB, Linköping University, Linköping, Sweden.
Francesco Casella, Politecnico di Milano, Milan, Italy.
Filippo Donida, Politecnico di Milano, Milan, Italy.
Henrik Eriksson, PELAB, Linköping University, Linköping, Sweden.
Anders Fernström, PELAB, Linköping University, Linköping, Sweden.

Jens Frenkel, TU Dresden, Dresden, Germany.
Pavel Grozman, Equa AB, Stockholm, Sweden.
Michael Hanke, NADA, KTH, Stockholm.
Daniel Hedberg, MathCore Engineering AB, Linköping, Sweden.
Alf Isaksson, ABB Corporate Research, Västerås, Sweden.
Kim Jansson, PELAB, Linköping University, Linköping, Sweden.
Daniel Kanth, Bosch-Rexroth, Lohr am Main, Germany.
Tommi Karhela, VTT, Espoo, Finland.
Joel Klinghed, PELAB, Linköping University, Linköping, Sweden.
Petter Krus, IEI, Linköping University, Linköping, Sweden.
Juha Kortelainen, VTT, Espoo, Finland.
Alexey Lebedev, Equa Simulation AB, Stockholm, Sweden.
Magnus Leksell, Linköping, Sweden.
Oliver Lenord, Bosch-Rexroth, Lohr am Main, Germany.
Ariel Liebman, Energy Users Association of Australia, Victoria, Australia.
Rickard Lindberg, PELAB, Linköping University, Linköping, Sweden
Håkan Lundvall, PELAB, Linköping University, Linköping, Sweden.
Henrik Magnusson, Linköping, Sweden.
Eric Meyers, Pratt & Whitney Rocketdyne, Palm City, Florida, USA.
Hannu Niemistö, VTT, Espoo, Finland.
Peter Nordin, IEI, Linköping University, Linköping, Sweden.
Kristoffer Norling, PELAB, Linköping University, Linköping, Sweden.
Lennart Ochel, Fachhochschule Bielefeld, Bielefeld, Germany.
Atanas Pavlov, Munich, Germany.
Karl Pettersson, IEI, Linköping University, Linköping, Sweden.
Pavol Privitzer, Institute of Pathological Physiology, Praha, Czech Republic.
Reino Ruusu, VTT, Espoo, Finland.
Per Sahlin, Equa Simulation AB, Stockholm, Sweden.
Wladimir Schamai, EADS, Hamburg, Germany.
Gerhard Schmitz, University of Hamburg, Hamburg, Germany.
Klas Sjöholm, PELAB, Linköping University, Linköping, Sweden.
Anton Sodja, University of Ljubljana, Ljubljana, Slovenia
Ingo Staack, IEI, Linköping University, Linköping, Sweden.
Kristian Stavåker, PELAB, Linköping University, Linköping, Sweden.
Sonia Tariq, PELAB, Linköping University, Linköping, Sweden.
Mohsen Torabzadeh-Tari, PELAB, Linköping University, Linköping, Sweden.
Niklas Worschech, Bosch-Rexroth, Lohr am Main, Germany.
Robert Wotzlaw, Goettingen, Germany.
Björn Zachrisson, MathCore Engineering AB, Linköping, Sweden.

B.3 OpenModelica Contributors 2009

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

Adrian Pop, PELAB, Linköping University, Linköping, Sweden.
Peter Aronsson, MathCore Engineering AB, Linköping, Sweden.

David Akhvlediani, PELAB, Linköping University, Linköping, Sweden.
Bernhard Bachmann, Fachhochschule Bielefeld, Bielefeld, Germany.
Vasile Baluta, PELAB, Linköping University, Linköping, Sweden.
Constantin Belyaev, Bashpromavtomatika Ltd., Ufa, Russia
Simon Björklén, PELAB, Linköping University, Linköping, Sweden.
Mikael Blom, PELAB, Linköping University, Linköping, Sweden.

Willi Braun, Fachhochschule Bielefeld, Bielefeld, Germany.
David Broman, PELAB, Linköping University, Linköping, Sweden.
Stefan Brus, PELAB, Linköping University, Linköping, Sweden.
Francesco Casella, Politecnico di Milano, Milan, Italy
Filippo Donida, Politecnico di Milano, Milan, Italy
Henrik Eriksson, PELAB, Linköping University, Linköping, Sweden.
Anders Fernström, PELAB, Linköping University, Linköping, Sweden.
Jens Frenkel, TU Dresden, Dresden, Germany.
Pavel Grozman, Equa AB, Stockholm, Sweden.
Michael Hanke, NADA, KTH, Stockholm
Daniel Hedberg, MathCore Engineering AB, Linköping, Sweden.
Alf Isaksson, ABB Corporate Research, Västerås, Sweden
Kim Jansson, PELAB, Linköping University, Linköping, Sweden.
Daniel Kanth, Bosch-Rexroth, Lohr am Main, Germany
Tommi Karhela, VTT, Espoo, Finland.
Joel Klinghed, PELAB, Linköping University, Linköping, Sweden.
Juha Kortelainen, VTT, Espoo, Finland
Alexey Lebedev, Equa Simulation AB, Stockholm, Sweden
Magnus Leksell, Linköping, Sweden
Oliver Lenord, Bosch-Rexroth, Lohr am Main, Germany
Håkan Lundvall, PELAB, Linköping University, Linköping, Sweden.
Henrik Magnusson, Linköping, Sweden
Eric Meyers, Pratt & Whitney Rocketdyne, Palm City, Florida, USA.
Hannu Niemistö, VTT, Espoo, Finland
Kristoffer Norling, PELAB, Linköping University, Linköping, Sweden.
Atanas Pavlov, Munich, Germany.
Pavol Privitzer, Institute of Pathological Physiology, Praha, Czech Republic.
Per Sahlin, Equa Simulation AB, Stockholm, Sweden.
Gerhard Schmitz, University of Hamburg, Hamburg, Germany
Klas Sjöholm, PELAB, Linköping University, Linköping, Sweden.
Martin Sjölund, PELAB, Linköping University, Linköping, Sweden.
Kristian Stavåker, PELAB, Linköping University, Linköping, Sweden.
Mohsen Torabzadeh-Tari, PELAB, Linköping University, Linköping, Sweden.
Niklas Worschech, Bosch-Rexroth, Lohr am Main, Germany
Robert Wotzlaw, Goettingen, Germany
Björn Zachrisson, MathCore Engineering AB, Linköping, Sweden

B.4 OpenModelica Contributors 2008

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

Adrian Pop, PELAB, Linköping University, Linköping, Sweden.
Peter Aronsson, MathCore Engineering AB, Linköping, Sweden.

David Akhvlediani, PELAB, Linköping University, Linköping, Sweden.
Bernhard Bachmann, Fachhochschule Bielefeld, Bielefeld, Germany.
Vasile Baluta, PELAB, Linköping University, Linköping, Sweden.
Mikael Blom, PELAB, Linköping University, Linköping, Sweden.
David Broman, PELAB, Linköping University, Linköping, Sweden.
Henrik Eriksson, PELAB, Linköping University, Linköping, Sweden.
Anders Fernström, PELAB, Linköping University, Linköping, Sweden.
Pavel Grozman, Equa AB, Stockholm, Sweden.

Daniel Hedberg, MathCore Engineering AB, Linköping, Sweden.
Kim Jansson, PELAB, Linköping University, Linköping, Sweden.
Joel Klinghed, PELAB, Linköping University, Linköping, Sweden.
Håkan Lundvall, PELAB, Linköping University, Linköping, Sweden.
Eric Meyers, Pratt & Whitney Rocketdyne, Palm City, Florida, USA.
Kristoffer Norling, PELAB, Linköping University, Linköping, Sweden.
Anders Sandholm, PELAB, Linköping University, Linköping, Sweden.
Klas Sjöholm, PELAB, Linköping University, Linköping, Sweden.
Kristian Stavåker, PELAB, Linköping University, Linköping, Sweden.
Simon Bjorklén, PELAB, Linköping University, Linköping, Sweden.
Constantin Belyaev, Bashpromavtomatika Ltd., Ufa, Russia

B.5 OpenModelica Contributors 2007

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

Adrian Pop, PELAB, Linköping University, Linköping, Sweden.
Peter Aronsson, MathCore Engineering AB, Linköping, Sweden.

David Akhvlediani, PELAB, Linköping University, Linköping, Sweden.
Bernhard Bachmann, Fachhochschule Bielefeld, Bielefeld, Germany.
David Broman, PELAB, Linköping University, Linköping, Sweden.
Henrik Eriksson, PELAB, Linköping University, Linköping, Sweden.
Anders Fernström, PELAB, Linköping University, Linköping, Sweden.
Pavel Grozman, Equa AB, Stockholm, Sweden.
Daniel Hedberg, MathCore Engineering AB, Linköping, Sweden.
Ola Leifler, IDA, Linköping University, Linköping, Sweden.
Håkan Lundvall, PELAB, Linköping University, Linköping, Sweden.
Eric Meyers, Pratt & Whitney Rocketdyne, Palm City, Florida, USA.
Kristoffer Norling, PELAB, Linköping University, Linköping, Sweden.
Anders Sandholm, PELAB, Linköping University, Linköping, Sweden.
Klas Sjöholm, PELAB, Linköping University, Linköping, Sweden.
William Spinelli, Politecnico di Milano, Milano, Italy
Kristian Stavåker, PELAB, Linköping University, Linköping, Sweden.
Stefan Vorkoetter, MapleSoft, Waterloo, Canada.
Björn Zachrisson, MathCore Engineering AB, Linköping, Sweden.
Constantin Belyaev, Bashpromavtomatika Ltd., Ufa, Russia

B.6 OpenModelica Contributors 2006

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

Peter Aronsson, MathCore Engineering AB, Linköping, Sweden.
Adrian Pop, PELAB, Linköping University, Linköping, Sweden.

David Akhvlediani, 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.

B.7 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.

B.8 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.

B.9 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.

B.10 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

B.11 OpenModelica Contributors 2001

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

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

Peter Aronsson, Linköping University, Linköping, Sweden.

B.12 OpenModelica Contributors 2000

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

B.13 OpenModelica Contributors 1999

Peter Fritzson, PELAB, Linköping University, Linköping, Sweden

–

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

B.14 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