The OpenModelica Modeling, Simulation, and Development Environment

Peter Fritzson, Peter Aronsson, Håkan Lundvall, Kaj Nyström, Adrian Pop, Levon Saldamli, David Broman
PELAB – Programming Environment Lab, Dept. Computer Science
Linköping University, S-581 83 Linköping, Sweden
{petfr, petar, haklu, kajny, adrpo, levsa, davbr}@ida.liu.se

Abstract

Modelica is a modern, strongly typed, declarative, and object-oriented language for modeling and simulation of complex systems. This paper gives a quick overview of some aspects of the OpenModelica environment – an open-source environment for modeling, simulation, and development of Modelica applications. An introduction of the objectives of the environment is given, an overview of the architecture is outlined and a number of examples are illustrated.

1 Introduction

The OpenModelica environment described in this paper has several goals, including, but not limited to the following:

- Providing an efficient interactive computational environment for the Modelica language.
- Development of a complete reference implementation of Modelica in a (currently) extended version of Modelica itself.
- Providing an environment for teaching modeling and simulation. 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.
- *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.

In this paper we briefly present a few of the subsystems, as well as some architectural aspects of the environment. Further, we will give examples of the usage of the interactive session handler, the DrModelica notebook, and the debugging support.

1.1 Environment Overview

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

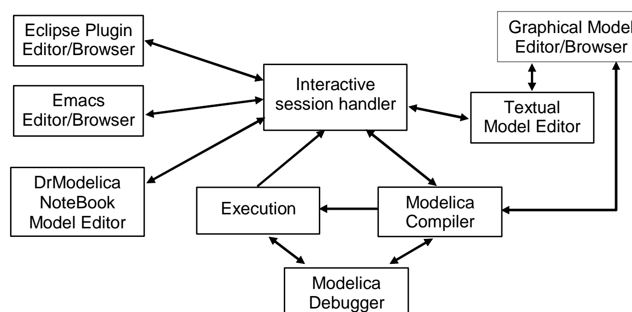

Figure 1. The architecture of the OpenModelica environment.

Arrows denote data and control flow. Several subsystems provide different forms of browsing and textual editing of Modelica code. The debugger currently provides debugging of an extended algorithmic subset of

Modelica. The graphical model editor is not really part of OpenModelica but integrated into the system and available from MathCore [6] without cost for academic usage. The following subsystems are currently integrated in the OpenModelica environment:

- *An interactive session handler*, that parses and interprets commands and Modelica expressions for evaluation, simulation, plotting, etc. The session handler also contains simple history facilities, and completion of file names and certain identifiers in commands.
- *A Modelica compiler subsystem*, translating Modelica to C code, with a symbol table containing definitions of classes, functions, and variables. Such definitions can be predefined, user-defined, or obtained from libraries. The compiler also includes a Modelica interpreter for interactive usage and constant expression evaluation. The subsystem also includes facilities for building simulation executables linked with selected numerical ODE or DAE solvers.
- *An execution and run-time module*. This module currently executes compiled binary code from translated expressions and functions, as well as simulation code from equation based models, linked with numerical solvers. Limited event handling facilities are included for the discrete and hybrid parts of the Modelica language.
- *Emacs textual model editor/browser*. In principle any text editor could be used. We have so far primarily employed Gnu Emacs, which has the advantage of being programmable for future extensions. A Gnu Emacs mode for Modelica has previously been developed. The Emacs mode hides Modelica graphical annotations during editing, which otherwise clutters the code and makes it hard to read. A speedbar browser menu allows to browse a Modelica file hierarchy, and among the class and type definitions in those files.
- *Eclipse plugin editor/browser/compilation manager*. The Eclipse plugin provides file and class hierarchy browsing and text editing capabilities, rather analogous to previously described Emacs editor/browser. Some syntax highlighting facilities are also included. The Eclipse framework has the advantage of making it easier to add future extensions such as refactoring and cross referencing support. A compilation manager is also included.
- *DrModelica notebook model editor*. This subsystem provides a lightweight notebook editor, compared to the more advanced Mathematica notebooks available in MathModelica. This basic functionality still allows essentially the whole DrModelica tutorial to be handled. Hierarchical text documents with chapters and sections can be represented and edited, including basic formatting. Cells can contain ordinary text or Modelica models and expressions, which can be evaluated and simulated. However, no mathematical typesetting or graphic plotting facilities are yet available in the cells of this notebook editor.
- *Graphical model editor/browser*. This is a graphical connection editor, for component based model design by connecting instances of Modelica classes, and browsing Modelica model libraries for reading and picking component models. The graphical model editor is not really part of OpenModelica but integrated into the system and provided by MathCore AB [6] without cost for academic usage. The graphical model editor also includes a textual editor for editing model class definitions, and a window for interactive Modelica command evaluation.
- *Modelica debugger*. The current implementation of the debugger [7] 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 [8] to Modelica. This is conventional full-feature debugger, using Emacs for displaying the source code during stepping, setting breakpoints, etc. Various back-trace and inspection commands are available. The debugger also includes a data-view browser for browsing hierarchical data such as tree- or list structures in extended Modelica.

1.2 Implementation Status

The current version of the OpenModelica environment (Sept 2005) allows most of the expression, algorithm, and function parts of Modelica to be executed interactively, as well as equation models and Modelica functions to be compiled into efficient C code. The generated C code is combined with a library of utility functions, a run-time library, and a numerical DAE solver. An external function library interfacing a LAPACK subset and other basic algorithms is under development.

Not all subsystems are yet integrated as well as is indicated in Figure 1. Currently there are two versions of the Modelica compiler, one which supports most of standard Modelica including simulation, and is connected to the interactive session handler, the notebook editor, and the graphic model editor, and another meta-programming Modelica compiler version which is integrated with the debugger and Emacs, supports meta-programming Modelica extensions [8], but does not allow equation-based modeling and simulation. Those

two versions are currently being merged into a single Modelica compiler version.

2 The OpenModelica Client-Server Architecture

The OpenModelica client-server architecture is schematically depicted in Figure 2, showing two typical clients: a graphic model editor and an interactive session handler for command interpretation.

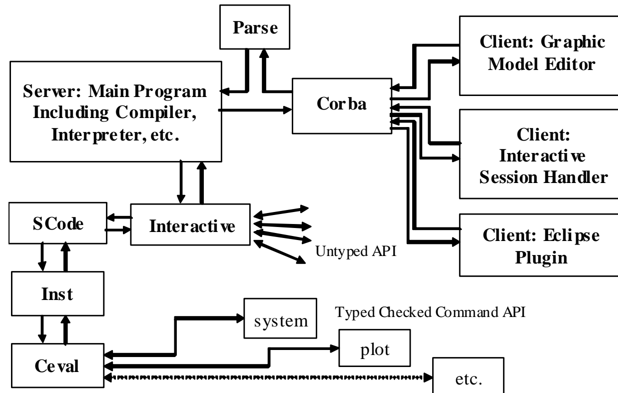

Figure 2. Client-Server interconnection structure of the compiler/interpreter main program and interactive tool interfaces.

Commands or Modelica expressions are sent as text from the clients via the CORBA interface, parsed, and divided into two groups by the main program:

- All kinds of declarations of classes, types, functions, constants, etc., as well as equations and assignment statements. Moreover, function calls to the untyped API also belong to this group – a function name is checked if it belongs to the API names. The Interactive module handles this group of declarations and untyped API commands.
- Expressions and type checked API commands, which are handled by the Ceval module.

The reason the untyped API calls are not passed via SCode (a module generating an intermediate form of the abstract syntax tree) and Inst (which performs symbolic instantiation of components) to Ceval is that Ceval can only handle typed calls – the type is always computed and checked, whereas the untyped API prioritizes performance and typing flexibility. The Main module checks the name of a called function name to determine if it belongs to the untyped API, and should be routed to Interactive.

Moreover, the Interactive module maintains an environment of all interactively given declarations and assignments at the top-level, which is the reason such items need to be handled by the Interactive module.

3 Simplified Overall Structure of the Compiler

The OpenModelica compiler is separated into a number of modules, to separate different stages of the translation, and to make it more manageable. The top level function is called `main`, and appears as follows in simplified form that emits flat Modelica (leaving out the code generation and symbolic equation manipulation):

```
function main
  input String f  "file name";
protected
  Absyn  ast;
  SCode  scode1;
  SCode  scode2;
algorithm
  ast := Parser.parse(f);
  scode1 := SCode.elaborate(ast);
  scode2 := Inst.elaborate(scode1);
  DAE.dump(scode2);
end main;
```

The simplified overall structure of the OpenModelica compiler is depicted in Figure 3, showing the most important modules, some of which can be recognized from the above `main` function. The total system contains approximately 40 modules.

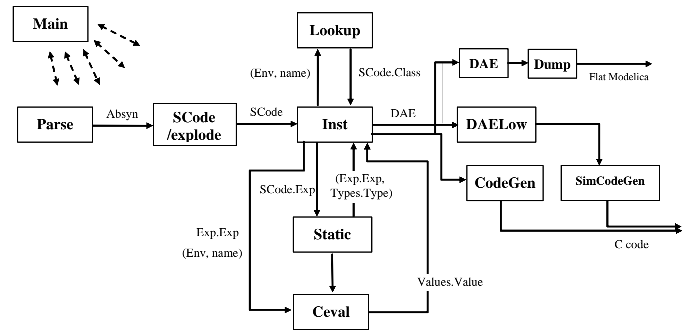

Figure 3. The Modelica compiler decomposed into modules and data flow connections.

The parser generates abstract syntax (Absyn) which is converted to the simplified (SCode) intermediate form. The code instantiation module (Inst) calls Lookup to find a name in an environment. It also generates the DAE equation representation which is simplified by DAELow. The Ceval module performs compile-time or interactive expression evaluation and returns values. The Static module performs static semantics and type checking. The DAELow module performs BLT sorting and index reduction (see Chapter 18 in [2]). The DAE module internally uses Exp.Exp, Types.Type and Algorithm.Algorithm; the SCode module internally uses Absyn.

4 Interactive Session with Examples

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

The Windows version which at installation is made available in the start menu as `OpenModelica->OpenModelica Shell` responds with an interaction window shown in Figure 4.

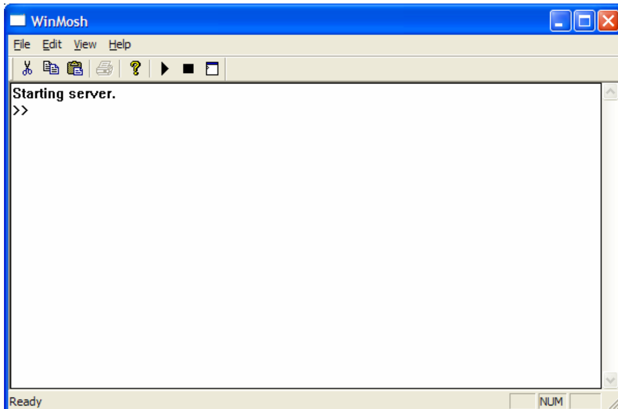

Figure 4. Initial screen of the interactive session handler.

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}
```

Look at the type of `x`:

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

The function `bubblesort` is called to sort this vector 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, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1}
```

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.

```
>> 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;
```

It is also possible to enter a function directly into the session handler.

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

And then call the function:

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

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

```
>> cd("..")
"/home/petfr/modelica"
```

We load a model, here the whole Modelica standard library:

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

We also load a file containing the dcmotor model:

```
>> loadFile("M:/modeq/VC7/Setup/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.ConstantVo
ltage 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 plot part of the simulated result:

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

The output is shown in Figure 5.

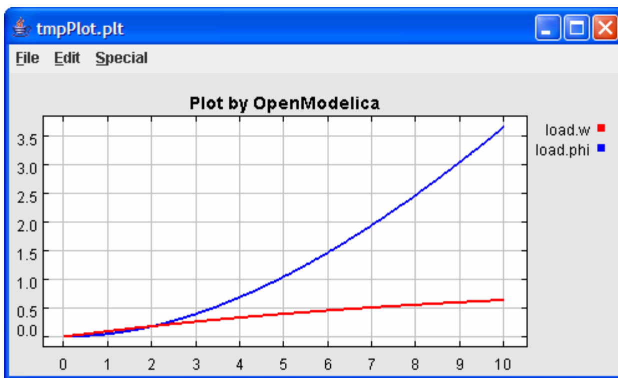

Figure 5. Plot of the simulated DCmotor model**.**

Clear all loaded libraries and models:

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

List the loaded models – but nothing left:

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

We load another model, the `Influenza` model:

```
>> loadFile("M:/modeq/VC7/Setup/testmodels/
Influenza.mo")
true
```

It is simulated:

```
>> simulate(Influenza,startTime=0.0,
stopTime=3.0)
record
    resultFile = "Influenza_res.plt"
end record
```

The simulated population is plotted, which is shown in Figure 6.

```
>> plot({Infected_Popul.p})
true
```

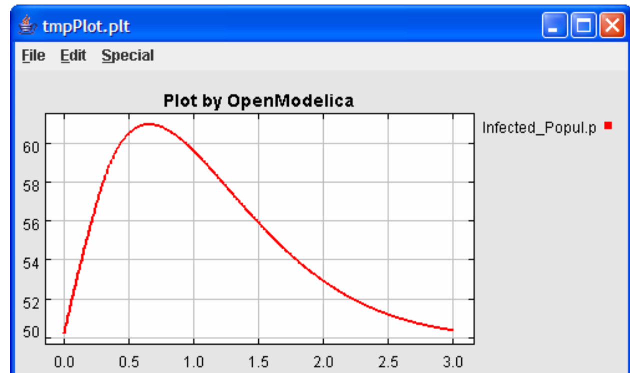

Figure 6. Plot of the Influenza model.

4.1 Commands for the Interactive Session Handler

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

- `instantiateModel(`*modelname*`)`
  Perform code instantiation of a model/class and return a string containing the flat class definition.
- `simulate(`*modelname*`)`
  Translate a model named *modelname* and simulate it.
- `simulate(`*modelname*`[,startTime=<`*Real*`>] [,stopTime=<`*Real*`>] [,numberOfIntervals =<`*Integer*`>])`
  Translate and simulate a model, with optional start time, stop time, and optional number of simulation intervals or steps for which the simulation results will be computed. Many steps will give higher time resolution, but occupy more space and take longer to compute. The default number of intervals is 500.
- `plot(`*vars*`)`
  Plot the variables given as a vector, e.g. `plot({x1,x2})`.
- `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.
- `typeOf(`*variable*`)`
  Return the type of the *variable* as a string.

- `clear()`
  Clear all loaded definitions.
- `clearVariables()`
  Clear all defined variables.
- `timing(`*expr*`)`
  Evaluate expression *expr* and return the number of seconds (elapsed time) the evaluation took.
- `cd()`
  Return the current directory.
- `cd(`*dir*`)`
  Change directory to the directory given as string.
- `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.
- `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*.
- `loadModel(`*classname*`)`
  Load model or package of name *classname* from `MODELICAPATH`.
- `loadFile(`*str*`)`
  Load Modelica file (`.mo`) with a name given as string argument *str*.
- `saveModel(`*str*`,`*modelname*`)`
  Save the model/class with name *modelname* in the file given by the string argument *str*.
- `help()`
  Print this helptext (returned as a string).
- `quit()`
  Leave and quit the OpenModelica environment

5 DrModelica Notebook and Textual Model Editor

The OpenModelica electronic notebook and model editor subsystem can be used as a textual modeling interface for Modelica, or as a Modelica tutoring system, i.e., a simplified version of the DrModelica tutoring system for teaching Modelica.

However, the OpenModelica notebook facility is work in progress, which currently is only partially completed. The simplified OpenModelica electronic notebooks are however still able to handle the full DrModelica tutorial material. It is advanced enough to represent hierarchical documents, simple type setting, text editing, etc.

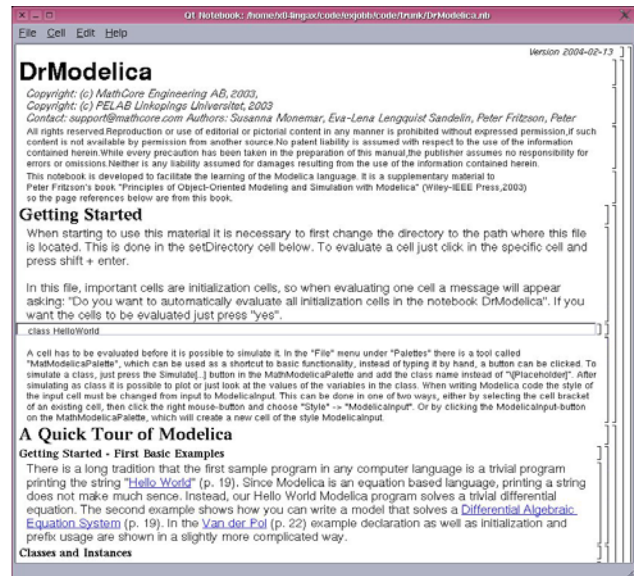

Figure 7. The start page (main page) of DrModelica in the OpenModelica notebook system.

This is exemplified by Figure 7, showing the DrModelica main page (start page) in the teaching material.

5.1 OpenModelica Notebook Commands

The current prototype of OpenModelica notebooks support the following operations:

- Opening and closing groups of cells by double clicking the hierarchical tree view (to the right).
- Evaluation of Modelica code, commands, and expressions in input cells by typing `SHIFT+RETURN`. The evaluation results are shown in a created output cell.
- Opening and loading notebook files stored in XML-format Command: `CTRL+O`).
- Opening and loading notebook files stored in Full-Form Mathematica notebook format.
- Saving notebook files in XML format.
- Terminating the notebook subsystem (`ALT+Q` or `ALT+F4`).
- Select a cell, by a single click on the cell in the tree view to the right.
- Possibility to edit the style template to change the appearance of different cell types.
- Move cursor, by `CTRL + UP ARROW` or `CTRL + DOWN ARROW`.
- Close current document (`CTRL+W`).
- Select and copy text inside a cell.

6 Modelica Algorithmic Subset Debugger

This section presents a comprehensive Modelica debugger [7] for an extended algorithmic subset of the Modelica language. This replaces debugging of algorithmic code using primitive means such as print statements or asserts which is complex, time-consuming and error- prone.

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

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

In this section we present parts of the debugger functionality by showing a debugging session on a short Modelica example. The functionality of the debugger is shown using pictures from the Emacs debugging mode for Modelica (`modelicadebug-mode`).

6.1 The Debugger Commands

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

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

If the debugger commands have several alternatives these are presented using the notation:

```
alternative1|alternative2|....
```

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

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

6.2 Starting the Modelica Debugging Subprocess

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

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

6.3 Setting/Deleting Breakpoints

A part of a session using this type of commands is shown in Figure 8 below. The presentation of the commands follows.

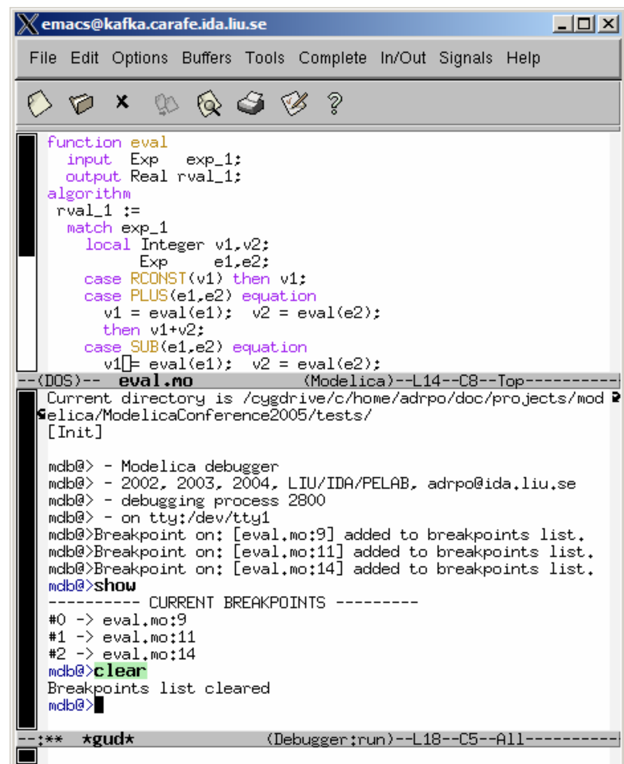

Figure 8. Using breakpoints.

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

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

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

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

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

Alternatively one can delete all breakpoints using:

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

Showing all breakpoints:

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

6.4 Stepping and Running

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

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

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

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

Examples of using these commands are presented in Figure 9.

```
emacs@kafka.carafe.ida.liu.se
File Edit Options Buffers Tools Complete In/Out Signals Help
function eval
  input  Exp   exp_1;
  output Real rval_1;
algorithm
 rval_1 :=
 match exp_1
    local Integer v1,v2;
          Exp     e1,e2;
  case RCONST(v1) then v1;
  case PLUS(e1,e2) equation
      v1 = eval(e1);  v2 = eval(e2);
      then v1+v2;
  case SUB(e1,e2) equation
      v1 = eval(e1);  v2 = eval(e2);
      then v1-v2;
  case MUL(e1,e2) equation
      v1 = eval(e1);  v2 = eval(e2);
      then v1*v2;
  case DIV(e1,e2) equation
--(DOS)--  eval.mo          (Modelica)--L9--C6--Top---------------
Current directory is /cygdrive/c/home/adrpo/doc/projects/modelica
/ModelicaConference2005/tests/
[Init]

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

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

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

Figure 9. Using command `run`.

This is only a brief presentation of a subset of the debugger functionality. See the OpenModelica Users Guide for a more complete description.

7 Conclusion

We have presented some aspects of the OpenModelica environment, including facilities for modeling, simulation, and debugging Modelica code. A number of objectives of the OpenModelica environment were given and some example illustrated. It has been shown that the OpenModelica environment includes many valuable features for engineers and researchers, and it is the only Modelica environment so far with good support for debugging Modelica algorithmic code as well as support for meta-programming integrated in the language. We believe that this open source platform can be part of forming the foundation of the next generation of the Modelica language and environment development efforts, both from a research perspective and a system engineering usage point of view.

8 Acknowledgements

This work was supported by Vinnova in the GRID-Modelica and SWEBProd projects, by SSF under the VISIMOD and RISE project, by the CUGS graduate school, and by MathCore Engineering AB.

References

[1] Peter Fritzson, *et al*. The Open Source Modelica Project. In Proceedings of The 2nd International Modelica Conference, 18-19 March, 2002. Munich, Germany See also: http://www.ida.liu.se/projects/OpenModelica.

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

[3] The Modelica Association. The Modelica Language Specification Version 2.2, March 2005. http://www.modelica.org.

[4] The OpenModelica Users Guide, version 0.2, April 2005. www.ida.liu.se/projects/OpenModelica

[5] The OpenModelica System Documentation, version 0.2, April 2005. www.ida.liu.se/projects/OpenModelica

[6] MathCore Engineering AB, www.mathcore.com.

[7] Adrian Pop and Peter Fritzson: A Portable Debugger for Algorithmic Modelica Code. In *Proceedings of the 4th International Modelica Conference*, Hamburg, Germany, March 7-8, 2005.

[8] Peter Fritzson, Adrian Pop, and Peter Aronsson. Towards Comprehensive Meta-Modeling and Meta-Programming Capabilities in Modelica. In *Proceedings of the 4th International Modelica Conference*, Hamburg, Germany, March 7-8, 2005.

[9] Michael Tiller. *Introduction to Physical Modeling with Modelica*. 366 pages. ISBN 0-7923-7367-7, Kluwer Academic Publishers, 2001.