A. Pop, P. Fritzson
Linköping University, Sweden
A Portable Debugger for Algorithmic Modelica Code

Program Committee

- Prof. Gerhard Schmitz, Hamburg University of Technology, Germany (Program chair).
- Prof. Bernhard Bachmann, University of Applied Sciences Bielefeld, Germany.
- Dr. Francesco Casella, Politecnico di Milano, Italy.
- Dr. Hilding Elmqvist, Dynasim AB, Sweden.
- Prof. Peter Fritzson, University of Linkping, Sweden
- Prof. Martin Otter, DLR, Germany
- Dr. Michael Tiller, Ford Motor Company, USA
- Dr. Hubertus Tummescheit, Scynamics HB, Sweden

A Portable Debugger for Algorithmic Modelica Code

Adrian Pop, Peter Fritzson

PELAB – Programming Environment Lab, Dept. Computer Science
Linköping University, S-581 83 Linköping, Sweden
{adrpo, petfr}@ida.liu.se

Abstract

In this paper we present the first comprehensive debugger for the algorithmic subset of the Modelica language, which augments previous work in our group on declarative static and dynamic debugging of equations in Modelica. 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: setting and removing breakpoints, single-stepping, inspecting variables, back-trace of stack contents, tracing, etc.

1 Introduction and Related Work

Most language development environments provide some kind of support for debugging and profiling.

Such techniques have also been developed for Modelica at the prototype level with regards to supporting declarative debugging of equation-based models [1, 2]. The presented work complements the existing debugging work with the first (to our knowledge) portable debugger for the algorithmic part of the Modelica language. The debugger is part of the Open Modelica project [3, 4].

There are several techniques for creating debuggers. Most of them are not portable and rely heavily on knowledge of the operating system and the underlying machine [5].

The approach we use in this paper is based on source code instrumentation of the intermediate representation in the compiler, similar with the work described in [6-8]. One can view the Modelica algorithmic code as an event generator and the debugger as a collector of these events that reacts to them accordingly.

The compiler has intimate knowledge of the Modelica code in its internal representation. We augment this representation with debugging nodes (or sites) that actually performs calls to the debugging functions. We have introduced a new compiling phase in the compiler where we walk on the internal representation and augment it with calls to several debugging functions implemented in C.

We have experimented with several ways of augmenting the internal compiler representation with debug nodes in order to search for the best memory consumption vs. speed of the debugger. These augmentation choices deal with the way the variables and code position is passed to the debugger functions.

The paper is structured as follows: This section presented an introduction and related work. Next section presents a debugging session on a short Modelica example, concentrating on the debug functionality. Also the debugger commands are introduced here. Details about the debugger are presented in detail in Section 3. Section 4 presents our conclusion and future work.

2 A debugging Session

This section presents the debugger functionality presenting a debugging session on a short Modelica example. The functionality of the debugger is presented using pictures from the Emacs debugging mode for Modelica (`modelicadebug-mode`).

2.1 The Debugger Commands

The Emacs Modelica debug mode is implemented as a specialization of the Grand Unified Debugger (GUD) interface (`gud-mode`) from Emacs [9]. 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 with pressing the `Enter` key and `<SPC>` with pressing `Space` key.

2.1.1 Starting the Modelica Debugging Subprocess

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

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

2.1.2 Setting/Deleting Breakpoints

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

Figure 1: Using breakpoints

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

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

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

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

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

Alternatively one can delete all breakpoints using:

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

Showing all breakpoints:

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

2.1.3 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 2. The example uses an extended version of Modelica [10] and is briefly described in Section 3.1.

Figure 2: Stepping and running

2.1.4 Examining Data

There are no GUD keybindings for these commands but they are inspired from the GNU Project debugger (GDB) [2].

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

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

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

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

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

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

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

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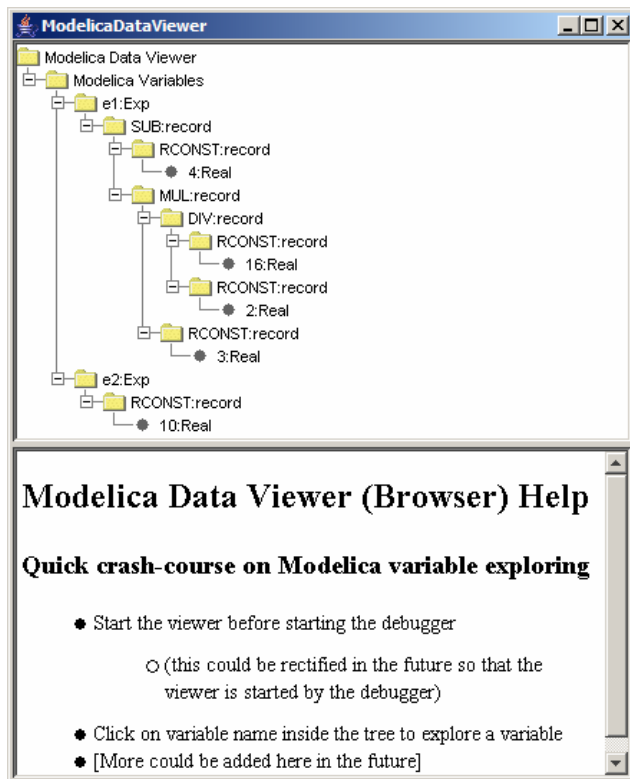

Figure 3: External variable browser

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

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

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

To print the entire display list:

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

Removing a display variable from the display list:

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

Removing all variables from the display list:

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

Printing the current live variables:

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

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

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

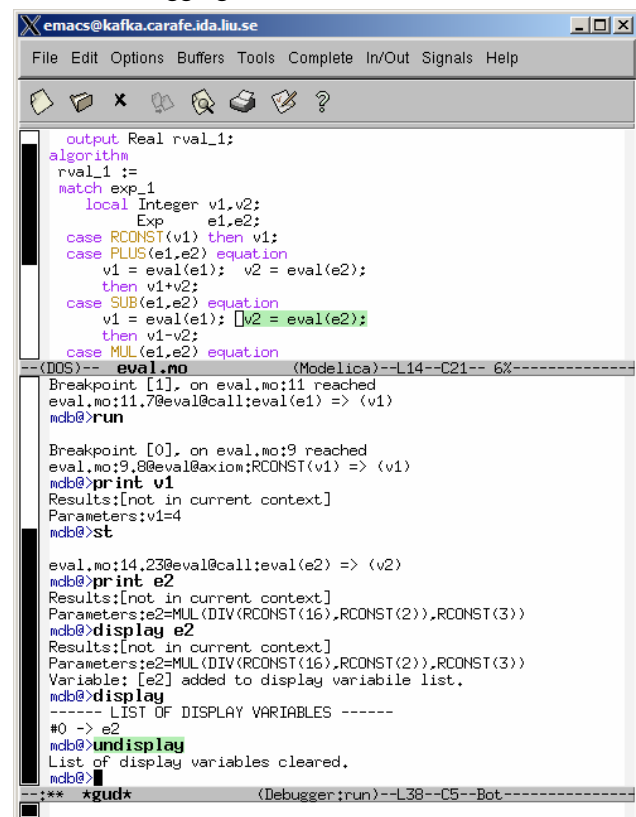Figure 4 shows examples of some of these commands within a debugging session:

Figure 4: Examining data

2.1.5 Additional commands

The stack contents (backtrace) can be displayed using:

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

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

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

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

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

For displaying the status of the Modelica runtime:

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

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

The current debugging settings can be displayed using:

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

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

One can invoke the debugging help by issuing:

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

For leaving the debugger one can use the command:

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

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

Figure 5: Additional commands

3   The Debugger Implementation

This section presents the debugging strategy in detail. We first start with two examples on how the debugger instruments the code, and then we enter into the details of the implementation. The examples illustrate Modelica algorithmic code and some of the new extensions of the Modelica language like pattern matching and union type declarations on a simple expression evaluator example.

3.1   Example Applications to Debug

In this section we present two examples of Modelica algorithms.

3.1.1   Bubble Sort function

The first example application we present for debugging is a BubbleSort function:

```
function bubbleSort
  input Real [:] unordElem;
  output Real [size(unordElem, 1)] ordElem;
  protected
    Real tempVal;
    Boolean isOver = false;
  algorithm
    ordElem := unordElem;
    while not isOver loop
      isOver := true;
      for i in 1:size(ordElem, 1)-1 loop
       if ordElem[i] > ordElem[i+1]
       then
          tempVal      := ordElem[i];
          ordElem[i]   := ordElem[i+1];
          ordElem[i+1] := tempVal;
          isOver := false;
        end if;
      end for;
    end while;
end bubbleSort;
```

The instrumented version of this function is presented below:

```
function bubbleSort
  input Real [:] unordElem;
  output Real [size(unordElem, 1)] ordElem;
  protected
    Real tempVal;
    Boolean isOver = false;
  algorithm
Debug.register_in("unordElem",unordElem);
Debug.step(...);
ordElem := unordElem;
Debug.register_out("ordElem", ordElem);
Debug.register_in("isOver", isOver);
Debug.step(...);
  while not isOver loop
      isOver := true;
      Debug.register_out("isOver", isOver);
      Debug.register_in("ordElem",ordElem);
      Debug.step(...);
      for i in 1:size(ordElem, 1)-1 loop
```

```
Debug.register_out("i", i);
Debug.register_in("i", i);
Debug.register_in("ordElem[i]",
            ordElem[i]);
Debug.register_in("ordElem[i+1]",
            ordElem[i+1]);
Debug.step(...);
if ordElem[i] > ordElem[i+1]
then
  Debug.register_in("i", i);
  Debug.register_in(
    "ordElem[i]",
     ordElem[i]);
  Debug.step(...);
  tempVal := ordElem[i];
  Debug.register_out(
    "tempVal",
     tempVal);
  Debug.register_in("i", i);
  Debug.register_in(
      "ordElem[i+1]",
       ordElem[i+1]);
  Debug.step(...);
  ordElem[i] := ordElem[i+1];
  Debug.register_out("i", i);
  Debug.register_out(
      "ordElem[i]",
       ordElem[i]);
  Debug.register_in("i", i);
  Debug.register_in(
      "tempVal",
       tempVal);
  Debug.step(...);
  ordElem[i+1] := tempVal;
  Debug.register_out("i", i);
  Debug.register_out(
      "ordElem[i+1]",
       ordElem[i+1]);
  Debug.step(...);
  isOver := false;
  Debug.register_out("isOver",
              isOver);
  Debug.step(...);
 end if;
 end for;
 Debug.register_out("i", i);
 Debug.register_out(
       "ordElem",
        ordElem);
 Debug.step(...);
 end while;
 Debug.register_out("isOver", isOver);
 Debug.register_out("ordElem",ordElem);
 Debug.step(...);
end bubbleSort;
```

As presented above, the debugger instruments all statements using the defined functions from the `Debug` package. A statement is analyzed for input and output variables which are registered with the debugging framework using `register_in` and `register_out` functions. The function `step` verify internally if we have to stop at a breakpoint or continue without stopping and also is responsible for parsing user commands addressed to the debugger. The instrumentation is better than what a programmer/user would do with print or assert statements because it provides better control through stop/inspect functionality. As one can see the code grows quite much when is instrumented, but this does not affect the final version of the code. For debugging purposes the user is rather interested in correctness of the code than in the speed/size of the code.

3.1.2 An expression evaluator

The second application is an expression evaluator implemented in the algorithmic Modelica subset extended with support for recursive tree data structures and a case-expression construct that allows pattern-matching and tree traversal. These language extensions are described in a companion paper [10] and are independent of the implemented debugger described here. For the sake of completeness we make present the extensions briefly in the following.

The declaration of an abstract syntax tree (AST) data type Exp for representing simple expressions:

```
uniontype Exp
  record RCONST Real x1; end RCONST;
  record PLUS  Exp x1; Exp x2; end PLUS;
  record SUB   Exp x1; Exp x2; end SUB;
  record MUL   Exp x1; Exp x2; end MUL;
  record DIV   Exp x1; Exp x2; end DIV;
  record NEG   Exp x1;          end NEG;
end Exp;
```

The union type declaration above is defining record constructors for the nodes of the simple expression representation. Examples of expressions represented in this way can be found in the following table:

Expression	Modelica constructor form
1+2	`PLUS(RCONST(1),` `    RCONST(2))`

1-2/3	`SUB(RCONST(1),`
	 `    DIV(RCONST(2),`
	 `       RCONST(3)))`

Table 1: Representing simple expression trees

To be able to evaluate simple expression trees we need an evaluation function. The evaluation function will apply pattern matching on the constructors of the expression language and then perform the actual evaluation on the components of the constructor.

Below we present the evaluation function `eval` of our simple expression evaluator:

```
function eval
  input  Exp   exp_1;
  output Real rval_1;
algorithm
 rval_1 :=
  match exp_1
    local Real v1,v2;
          Exp  e1,e2;
    case RCONST(v1) then v1;
```

```
case PLUS(e1,e2) equation
  v1 = eval(e1);  v2 = eval(e2);
  then v1+v2;
case SUB(e1,e2) equation
  v1 = eval(e1);  v2 = eval(e2);
  then v1-v2;
case MUL(e1,e2) equation
  v1 = eval(e1);  v2 = eval(e2);
  then v1*v2;
case DIV(e1,e2) equation
  v1 = eval(e1);  v2 = eval(e2);
  then v1/v2;
case NEG(e1) equation
  v1 = eval(e1);
  then -v1;
else
  assert("Bad expression!"));
end match;
end eval;
```

This function has as input an expression in the form presented in Table 1, second column. The expressions are represented as trees using constructors defined in the union type `Exp`. A model that uses this function is presented below:

```
model Example
  Exp exp=PLUS(
          SUB(RCONST(4),
             MUL(DIV(RCONST(16),
                    RCONST(2)),
                RCONST(3))),
          RCONST(10));
  Real result;
algorithm
  result := eval(exp);
end Example
```

The first component of the `Example` model defines a simple tree that corresponds to `4-16/2*3+10` expression. We used this simple expression in the examples and figures in Section 2.

The instrumented code of the eval function is presented below. The debugging code is underlined to be more visible:

```
function eval  // instrumented version
  import Modelica.Debugging;
  input Exp      exp_1;
  output Real realval_1;
algorithm
Debug.register_in("exp1", exp_1);
Debug.step(...);
realval_1 :=
  match exp_1
    local Real v1,v2;
    Exp e1,e2;
    case RCONST(v1) equation
      Debug.register_out("v1", v1);
      Debug.step(...);
    then v1;
    case PLUSop(e1,e2) equation
      Debug.register_out("e1", e1);
      Debug.register_out("e2", e2);
      Debug.register_in("e1", e1);
      Debug.step(...);
      v1 = eval(e1);
```

```
      Debug.register_out("v1", v1);
      Debug.resister_in("e2", e2);
      Debug.step(...);
      v2 = eval(e2);
      Debug.register_out("v2", v1);
      Debug.register_out("v1+v2",v1+v2);
      Debug.step(...);
    then v1+v2;
    ...
  else
    Debug.step(...);
    assert("Bad expression!"));
end match;
Debug.register_out("realval_1",realval_1);
Debug.step();
end eval;
```

As one can see, debugging code is added for each variable. This style of debug code instrumentation can be changed into one where all the debugging calls are collapsed into just one call `Debug.step(...)` with more arguments specifying in or out variables. We have experimented with different debug instrumentation styles in order to choose the best speed vs. memory consumption for the debugger.

The following instrumentation functions are inserted into the generated code:

- The functions: `Debug.register_in("name", var)` and `Debug.register_out("name", var)` register in a data structure the variables which are live at a certain moment during the execution.
- The `Debug.step(...)` function then performs a query of this data structure to show which variables are available in the current context.
- The function `Debug.register_in(...)` registers variables that are used in the next statement or expression.
- The function `Debug.register_out(...)` registers variables that result from the execution of the previous statement or expression.

Note that the debug instrumentation functions are low-level C functions that do not fulfill the Modelica requirement of being mathematical functions.

3.2 Overview

In this section we present the compilation path followed by the compiler when instrumenting the code with debugging calls. The debugger is actually the executable generated by the compiler when instructed to generate debugging calls before and after each relevant Modelica statement or expression.

Figure 6 presents both the normal compilation path performed by the compiler when compiling algorithmic code and also the path followed by the compiler when compiling algorithm sections that include debugging information.
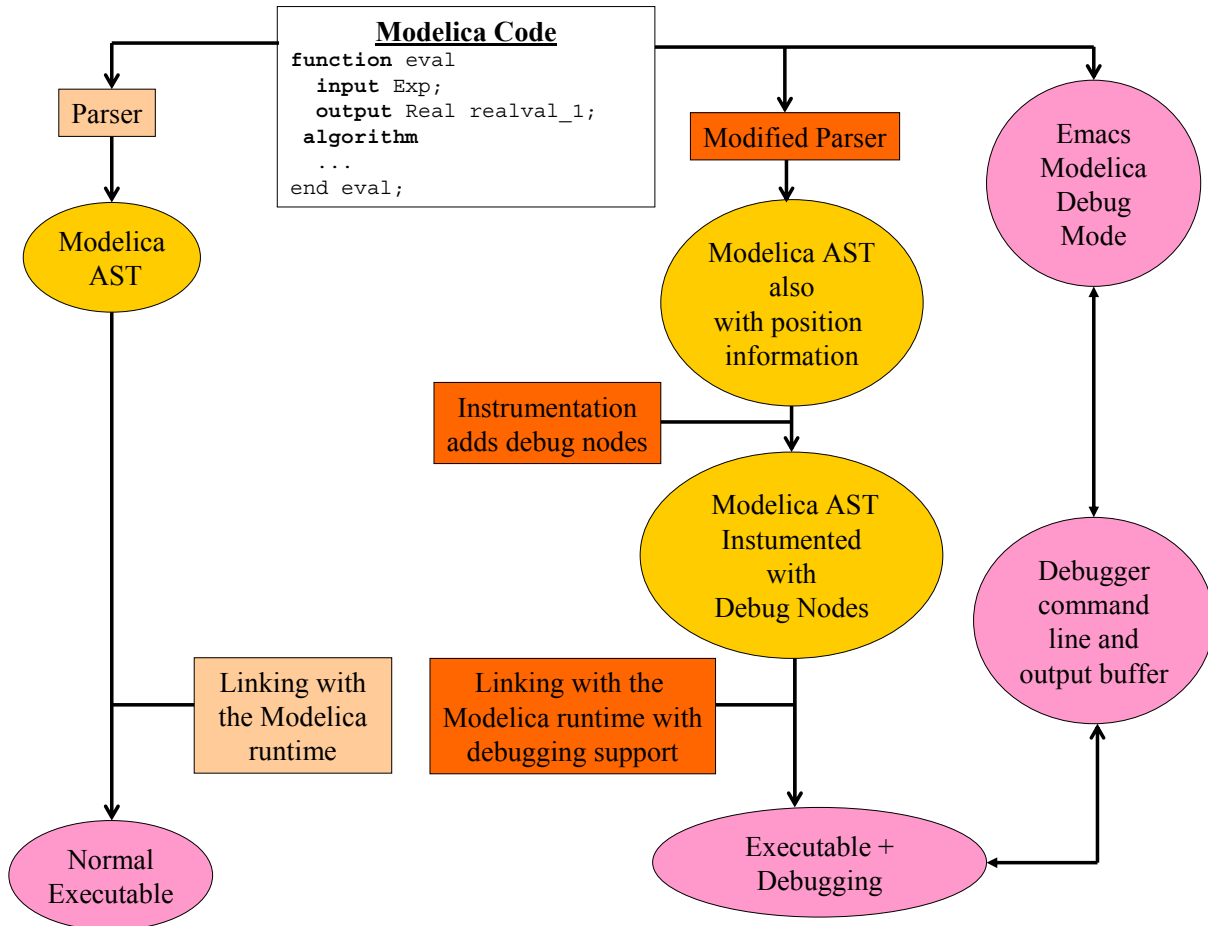
Figure 6: Normal compilation (left) and compilation with debug support (right)

An overview of our debugging strategy is presented on the right side of Figure 6. The path taken by the debugger comprises several more steps in order to instrument the Modelica AST with debug nodes and live variable information. Also, the runtime system is extended with several data structures that support debugging and a parser for the debugger commands.

3.3 Augmenting the Modelica AST with Debug Call Nodes

The modified Modelica parser saves additional position information about each statement or expression. This information is used by the debug instrumentation phase in order to generate calls to the debugger functions with exact information on where the current execution is taking place.

We use a very simple and effective algorithm when instrumenting the ModelicaAST with debug nodes. We sketch a pseudo code of this algorithm below:

```
foreach ModelicaAST expression or
        statement Node
  if not filter(Node)
  then select next Node;
```

```
else
  variables_in = collect_variables(Node);
  variables_out = collect_variables(Node);
  position = collect_position(Node);
  construct new tree with these nodes:
    Debug.register_in(variables_in);
    Debug.step(position);
    Node;
    Debug.register_out(variables_out);
  replace Node with the new tree;
  end else;
end foreach;
```

The compiler can be instructed to generate debugging nodes only when reaching certain nodes that are conform to a filter. Using this facility one can tell the compiler to perform debug instrumentation only on a certain function or a certain statement of the code. In this way the delay in the execution speed introduced by the debugging code can be kept to a minimum.

We have experimented with several ways of creating the added debug nodes:

- Each variable is registered using a debug function call either as `in` or `out` variable, as in the pseudo code presented.
- All `in` variables are collected in a list and passed to a single function call. The same approach is used also for `out` variables. This has an impact on mem-

ory consumption, but uses fewer function calls, so it is faster.

- The `in` and `out` variables are collected in two lists which are passed as arguments to the `step` function directly.

The best speed vs. memory consumption is highly dependent on the algorithmic code. However one can experiment with all these choices and choose the best debug instrumentation way for a specific code.

3.4   Short Presentation of the Debugger Library

The debugger library contains several functions implementing the actual debugger functionality and data structures for bookkeeping of breakpoints, live variables, call stack, types of live variables, etc. The library is implemented in C.

The debugger library has the following available functions:

- `Debug.step(...)` function with fixed parameters `file_name`, `lineno`, `columno`, `function_name`, `next_statement` is stopping the execution of the algorithmic code if a breakpoint is reached or one step was performed and waits for commands from the user. If the commands are not `step` or `run` it waits for more commands from the user in a cycle. Additional parameters like the live variables can be also passed to the `Debug.step(...)` function depending on the debug instrumentation choice.
- `Debug.register_in(...)` function and `Debug.register_out(...)` function are used only when no live variables are passed to the `Debug.step()` function. These functions are registering variables, either each variable at a time or several variables as a list.
- `Debug.parse_command()` is called by `Debug.step()` either at a step or when a breakpoint is reached.
- `Debug.execute_command()` is called by the `Debug.parse_command()` when the user issues a command. Depending on the command several other functions are executed.
- `Debug.set_breakpoint()` adds the breakpoint into the breakpoint list.
- `Debug.delete_breakpoint()` deletes the breakpoint from the breakpoint list.
- `Debug.clear()` and `Debug.show()` clears or shows all current breakpoints, respectively.
- `Debug.print_variable()` prints the specified variable to the debugger output.
- `Debug.print_livevars_names()` prints the names of the variables available in the current con-

text. The distinction between `in` (parameters) and `out` (results) variables is made when printing variable names.

- `Debug.browse_variable()` connects to the external viewer, and sends on demand the value of a variable. This function stops the debugger until the external viewer is done with the browsing.
- `Debug.set_print_depth()` sets the depth of variable printing.
- `Debug.max_remembered_stack_entries()` will set the maximum number of entries of the logged stack trace.
- `Debug.display_variable()` will add the display variable to a display list to be printed at each step or breakpoint.
- `Debug.undisplay_variable()` performs the inverse action of the `Debug.display_variable()`.
- `Debug.display()` prints the list of variable names present in the display list.
- `Debug.undisplay()` clears the display list.
- `Debug.stack_add_node()` pushes a node name on the stack trace.
- `Debug.stack_remove_node()` pops a node name from the stack trace.
- `Debug.status()` prints status information on the extended Modelica runtime, e.g., garbage collections performed, amount of allocated memory, etc.
- `Debug.settings()` prints the current debugger settings.

More functions are actually present in the debug library (dealing with variable – type mapping, connection to the external viewer, etc). Here we have only presented a limited set which has direct connections with the debugger commands presented in the paper.

4   Conclusions and Future Work

We have presented a portable and highly configurable debugger for extended Modelica algorithmic code. Debugging of large algorithmic Modelica codes is now possible using our debugger.

As future work we consider extension of the current debugging scheme and also tighter integration of the debugger with other Modelica tools.

Integration with declarative equation debugger techniques [1, 2] will be provided in the future, in order to address debugging of the entire Modelica language from a central debugger.

We have also started work to integrate the debugger and the OpenModelica [3] compiler within the Eclipse Development platform [11] which will provide integrated editing, navigation, simulation and debugging for the Modelica language.

Acknowledgements

References

1. Bunus, P., *Debugging and Structural Analysis of Declarative Equation-Based Languages*, in *Department of Computer and Information Science*. 2002, Linköping University: Linköping, Licentiate Thesis.

2. Bunus, P., *Debugging Techniques for Equation-Based Languages*, in *Department of Computer and Information Science*. 2004, May, Linköping University: Linköping, PhD Thesis.

3. Fritzson, P., et al. *The Open Source Modelica Project*. in *Proceedings of The 2th International Modelica Conference*, 18-19 March, 2002. Munich, Germany, http://www.ida.liu.se/~pelab/modelica/OpenModelica.html.

4. Fritzson, P., *Principles of Object-Oriented Modeling and Simulation with Modelica*. 2003: Wiley-IEEE Press.

5. GNU, T.F.S.F., *The GNU Project debugger*, Last Accessed: December, 2004, http://www.gnu.org/software/gdb/gdb.html.

6. Tolmach, A.P., *Debugging Standard ML*. 1992, October, Princeton University, PhD. Thesis.

7. Pettersson, M. *Portable Debugging and Profiling*. in *7th International Conference on Compiler Construction*, 1998.

8. Hanson, D.R. and M. Raghavachari, *A Machine-Independent Debugger—Revisited*. Software—Practice and Experience, 1999. **29**(10): p. 849-862.

9. GNU, T.F.S.F., *Emacs, The Grand Unified Debuger (GUD)*, Last Accessed: December, 2004, http://www.gnu.org/software/emacs/manual/html_node/Debuggers.html#Debuggers.

10. Fritzson, P., A. Pop, and P. Aronsson. *Towards Comprehensive Meta-Modeling and Meta-Programming Capabilities in Modelica, (to appear)*. in *4th International Modelica Conference*, 7-8 March, 2005. Hamburg, Germany.

11. *Eclipse Development Platform*, Last Accessed: December, 2004, http://www.eclipse.org/.