Efficient Debugging of Large Algorithmic Modelica Applications

Adeel Asghar, Adrian Pop, Martin Sjölund and Peter Fritzson

Programming Environments Laboratory
Department of Computer and Information Science
Linköping University
{adeel.asghar,adrian.pop,martin.sjolund,peter.fritzson}@liu.se

Abstract: Modelica models often contain functions with algorithmic code. The fraction of algorithmic code is increasing in Modelica models since Modelica, in addition to equation-based modeling, is also used for embedded system control code and symbolic model transformations in compilers using the MetaModelica language extension. For these reasons, debugging of algorithmic Modelica code is becoming increasingly relevant.

Our earlier work in debuggers for the algorithmic subset of Modelica used trace-based techniques. These have the advantages of being very portable, but turned out to have too much overhead for very large applications.

The new debugger is the first Modelica debugger that can operate without trace information. Instead it communicates with a low-level C-language symbolic debugger, the Gnu debugger GDB, to directly extract information from a running executable, set and remove breakpoints, etc. This is made possible by the new bootstrapped OpenModelica compiler which keeps track of a detailed mapping from the high level Modelica code down to the generated C code compiled to machine code.

The debugger is operational, supports browsing of both standard Modelica data structures and tree/list data structures, and operates efficiently on large applications such as the OpenModelica compiler with more than 100 000 lines of code.

Keywords: Modelica, Run-time Debugging, Modeling and Simulation, Algorithmic code, Eclipse.

1. INTRODUCTION

Advanced development of today's complex products requires integrated environments and modeling languages for modeling and simulation. Equation-based object-oriented declarative (EOO) languages such as Modelica (Peter Fritzson 2004) is the key approach to physical system modeling and simulation. The increased ease of use, the high abstraction, and the expressivity of EOO languages are very attractive properties. However, these attractive properties come with the drawback that programming and modeling errors are often hard to find.

This paper presents a new efficient debugger for the algorithmic subset of Modelica that is applicable to large applications. To address debugging of full Modelica models, we plan to integrate this debugger with debugging mechanisms that support equation-based part of models.

The paper is structured as follows: Section 2 presents the background and related work, Section 3 presents an overview of the debugger, whereas Section 4 explains the implementation details. Conclusions and future work are given in Section 5 and finally Section 6 presents the acknowledgements.

2. BACKGROUND AND RELATED WORK

Any mature language implementation supports debugging facilities to simplify and support developers in the process of finding bugs in source code. The implementation of such debugging support can be based on several different techniques ranging from source code instrumentation to machine code instrumentation combined with hardware support.

Previous work by Peter Bunus and Peter Fritzson (2003; 2004) on debugging equation based languages such as Modelica has mostly targeted the structure of the equations to determine why a system is not balanced. Debugging the symbolic transformations performed by the model compiler for models in equation based languages is addressed by (Martin Sjölund and Peter Fritzson, 2011b). Previous work on debugging algorithmic Modelica code (Adrian Pop and Peter Fritzson, 2005; Adrian Pop, et al., 2006) was based on code instrumentation. While robust and portable it was too slow to efficiently use for very large application models. To our knowledge no other open-source or commercial Modelica tool currently supports algorithmic code debugging.

3. USING THE NEW EFFICIENT DEBUGGER

The debugger is developed as a debug plugin for the Modelica Development Tooling (MDT) which is a Modelica programming perspective for Eclipse. The Eclipse-based user interface of the new efficient debugger is depicted in *Fig. 1*.

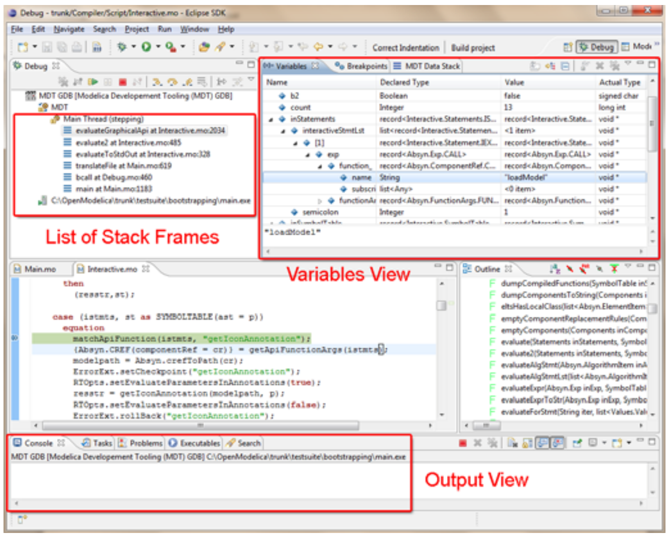


Fig. 1. The debug view of the new efficient debugger within the MDT Eclipse plugin.

The debugger provides the following general functionalities:

- Adding/Removing breakpoints.
- Step Over – moves to the next line, skipping the function calls.
- Step In – steps into the called function.
- Step Return – completes the execution of the function and comes back to the point from where the function is called.
- Suspend – interrupts the running program.
- Resume – continues the execution from the most recent breakpoint.
- Terminate – stops the debugging session.

It has several advantages compared to the old trace-based debugger:

- It is much faster, partly because it only fetches data on request, but primarily because the original compiled machine code of the application can be executed.
- It supports step over. (the old debugger only does step in).
- Supports step return. (the old debugger does not have that feature).
- Preserves stack frames and variables.

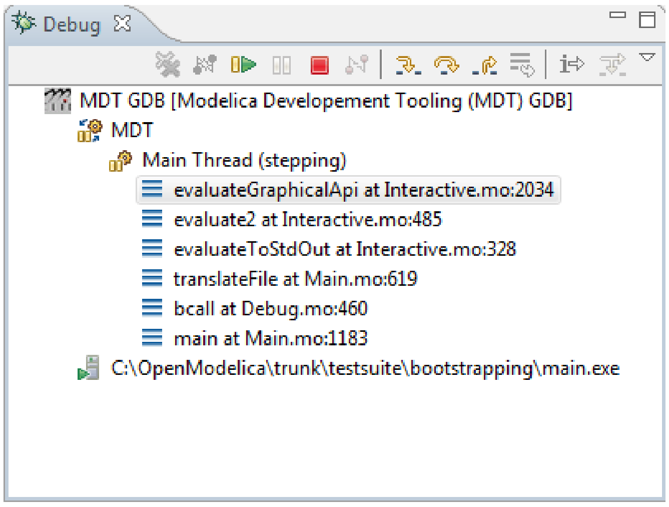
The debug view primarily consists of two main views:

- Stack Frames View
- Variables View

The stack frame view, shown in the *Fig. 2*, 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 possible to select the previous frame in the stack and inspect the values of the variables in that frame. However, it is not allowed to select any of the previous frames and start debugging from there.

Each frame is shown as *<function_name at file_name:line_number>*.



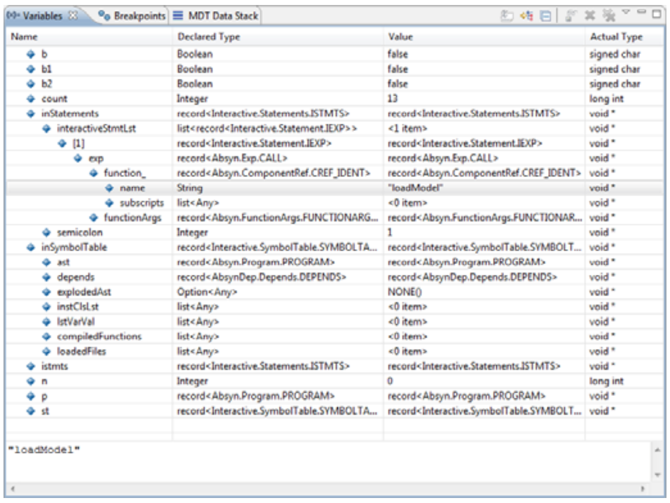
MDT GDB [Modelica Developement Tooling (MDT) GDB]
MDT
Main Thread (stepping)
evaluateGraphicalApi at Interactive.mo:2034
evaluate2 at Interactive.mo:485
evaluateToStdOut at Interactive.mo:328
translateFile at Main.mo:619
bcall at Debug.mo:460
main at Main.mo:1183
C:\OpenModelica\trunk\testsuite\bootstrapping\main.exe

Fig. 2. The stack frame view of the debugger.

The Variables view shows the list of variables at a certain point in the program. It contains four columns:

- 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 the 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).



Name	Declared Type	Value	Actual Type
b	Boolean	false	signed char
b1	Boolean	false	signed char
b2	Boolean	false	signed char
count	Integer	13	long int
inStatements	record<Interactive.Statements.ISTMTS>	record<Interactive.Statements.ISTMTS>	void *
interactiveStmtLst	list<record<Interactive.Statement.IEXP> >	<1 item>	void *
[1]	record<Interactive.Statement.IEXP>	record<Interactive.Statement.IEXP>	void *
exp	record<Absyn.Exp.CALL>	record<Absyn.Exp.CALL>	void *
function_	record<Absyn.ComponentRef.CREF_IDENT>	record<Absyn.ComponentRef.CREF_IDENT>	void *
name	String	"loadModel"	void *
subscripts	list<Any>	<0 item>	void *
functionArgs	record<Absyn.FunctionArgs.FUNCTIONARG...	record<Absyn.FunctionArgs.FUNCTIONAR...	void *
semicolon	Integer	1	void *
inSymbolTable	record<Interactive.SymbolTable.SYMBOLTA...	record<Interactive.SymbolTable.SYMBOLT...	void *
ast	record<Absyn.Program.PROGRAM>	record<Absyn.Program.PROGRAM>	void *
depends	record<AbsynDep.Depends.DEPENDS>	record<AbsynDep.Depends.DEPENDS>	void *
explodedAst	Option<Any>	NONE()	void *
instClsLst	list<Any>	<0 item>	void *
lstVarVal	list<Any>	<0 item>	void *
compiledFunctions	list<Any>	<0 item>	void *
loadedFiles	list<Any>	<0 item>	void *
istmts	record<Interactive.Statements.ISTMTS>	record<Interactive.Statements.ISTMTS>	void *
n	Integer	0	long int
p	record<Absyn.Program.PROGRAM>	record<Absyn.Program.PROGRAM>	void *
st	record<Interactive.SymbolTable.SYMBOLTA...	record<Interactive.SymbolTable.SYMBOLT...	void *

```
"loadModel"
```

Fig. 3. The variable view of the new debugger.

4. IMPLEMENTATION ASPECTS

In order to keep track of Modelica source code positions, the Modelica source-code line numbers are inserted into the transformed C source-code. This information is used by the Gnu Compiler GCC to create the debugging symbols that can be read by the Gnu debugger GDB (Richard Stallman, Roland Pesch, Stan Shebs, et al., 2011).

Through the bootstrapped OpenModelica Compiler by Martin Sjölund, Peter Fritzson and Adrian Pop (2011a) the line number information is propagated all the way from the high level Modelica representation to the low level intermediate representation and the generated code.

This approach was developed for the symbolic model transformation debugger described in (Martin Sjölund and Peter Fritzson, 2011b) and is also used in this debugger.

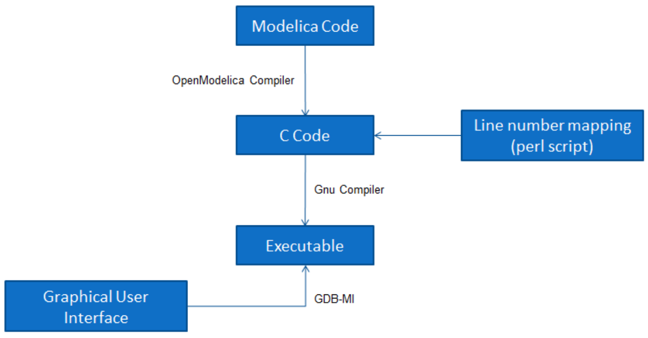
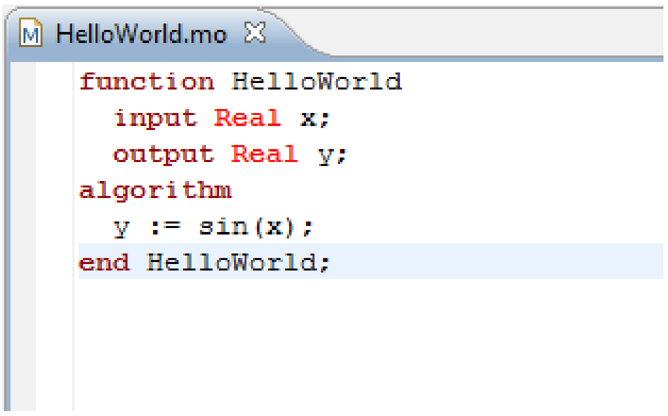


Fig. 4. Debugger flow.

Consider the following Modelica code shown in *Fig. 5*.



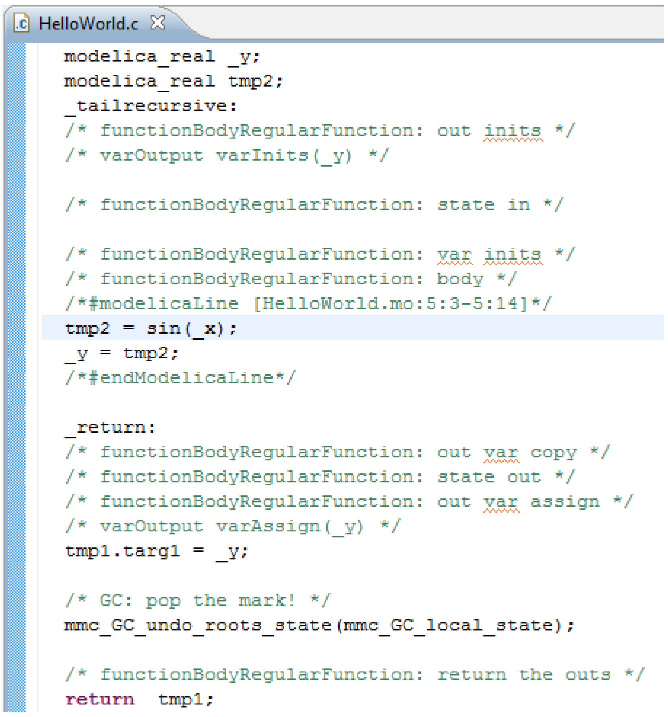
```
HelloWorld.mo
function HelloWorld
  input Real x;
  output Real y;
algorithm
  y := sin(x);
end HelloWorld;
```

Fig. 5. Modelica Code.

The OpenModelica Compiler compiles this HelloWorld function into the C source-code depicted in *Fig. 6*.

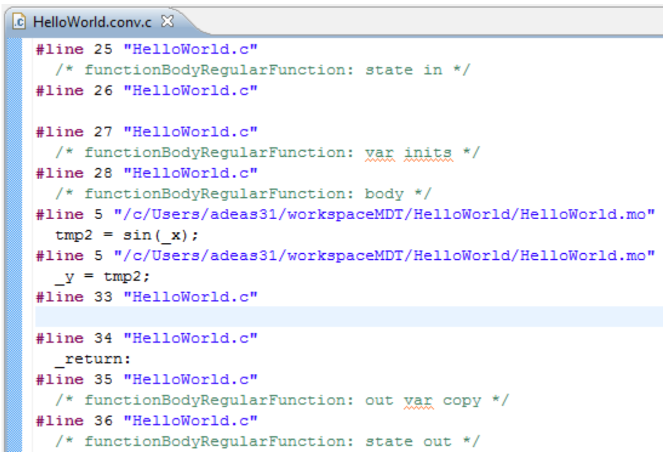
The generated code contains blocks which represent the Modelica code lines. The blocks are mentioned as comments in the following format */*#modelicaLine [modelica_source_file:line_number_info]*/*.

This information is now used to generate debug symbols that are recognized by GDB. The generated C source-code is used by a small Perl script to create another version of the same source-code with different line number blocks, see *Fig. 7*.



```
HelloWorld.c
  modelica_real _y;
  modelica_real tmp2;
  _tailrecursive:
  /* functionBodyRegularFunction: out inits */
  /* varOutput varInits(_y) */

  /* functionBodyRegularFunction: state in */

  /* functionBodyRegularFunction: var inits */
  /* functionBodyRegularFunction: body */
  /*#modelicaLine [HelloWorld.mo:5:3-5:14]*/
  tmp2 = sin(_x);
  _y = tmp2;
  /*#endModelicaLine*/

  _return:
  /* functionBodyRegularFunction: out var copy */
  /* functionBodyRegularFunction: state out */
  /* functionBodyRegularFunction: out var assign */
  /* varOutput varAssign(_y) */
  tmp1.targ1 = _y;

  /* GC: pop the mark! */
  mmc_GC_undo_roots_state(mmc_GC_local_state);

  /* functionBodyRegularFunction: return the outs */
  return  tmp1;
```

Fig. 6. Generated C source-code.



```
HelloWorld.conv.c
#line 25 "HelloWorld.c"
  /* functionBodyRegularFunction: state in */
#line 26 "HelloWorld.c"

#line 27 "HelloWorld.c"
  /* functionBodyRegularFunction: var inits */
#line 28 "HelloWorld.c"
  /* functionBodyRegularFunction: body */
#line 5 "/c/Users/adeas31/workspaceMDT/HelloWorld/HelloWorld.mo"
  tmp2 = sin(_x);
#line 5 "/c/Users/adeas31/workspaceMDT/HelloWorld/HelloWorld.mo"
  _y = tmp2;
#line 33 "HelloWorld.c"

#line 34 "HelloWorld.c"
  _return:
#line 35 "HelloWorld.c"
  /* functionBodyRegularFunction: out var copy */
#line 36 "HelloWorld.c"
  /* functionBodyRegularFunction: state out */
```

Fig. 7. Converted C source-code.

The converted C source-code contains a line number mapping between the generated C source-code and the actual Modelica source-code in the GDB specific format. Examine the lines starting with *#line* in *Fig. 7*.

The executable is created from the converted C source-code and is debugged at the C code level using GDB, which is communicating with the Modelica debugger. GDB provides two kinds of interpreters; the command line interface (CLI) and the machine interface (MI). If GDB is invoked using the CLI interpreter then the MI commands cannot be used. However, the vice versa is not true which means if GDB is invoked using the MI interpreter both the MI and the CLI commands can be used. The CLI interpreter is more user friendly. On the other hand the MI interpreter is more convenient for detailed control of the execution at the low level.

By utilizing the capabilities of MI, the MDT (Modelica Development Tooling) debug plugin for Eclipse has been enhanced in such a way that the old and the new debugger can run simultaneously.

The Eclipse interface allows adding/removing breakpoints. The breakpoints are created by sending the <*-break-insert filename:linenumber*> command to GDB. At the moment only line number based breakpoints are supported. Other alternatives to set the breakpoints are; <*-break-insert function*>, <*–break-insert filename:function*>.

The program execution is handled by sending asynchronous commands to GDB via the Eclipse interface. These commands includes the following:

- <*-exec-run*> – starts the program.
- <*-exec-continue*> – continue program execution.
- <*-exec-next*> – performs step over.
- <*-exec-step*> – performs step in.
- <*-exec-finish*> – performs step return.

These program execution commands are asynchronous because they do not send back any acknowledgement. However, GDB raises signals;

- as a response to those asynchronous commands.
- for notifying program state.

The debugger uses the following signals to perform specific actions:

- *breakpoint-hit* – raised when a breakpoint is reached.
- *end-stepping-range* – raised when a step into or step over operations are finished.
- *function-finished* – raised when a step return operation is finished.

These signals are utilized by the debugger to extract the line number information and highlight the line in the source-code editor. They are also used as notifications for the debugger to start the routines to fetch the new values of the variables.

The suspend functionality which interrupts the running program is implemented in the following way. On Windows GDB interrupts do not work. Therefore a small program *BreakProcess* is written to allow interrupts on Windows. The debugger calls *BreakProcess* by passing it the process ID of the debugged program. *BreakProcess* then sends the *SIGTRAP* signal to the debugged program so that it will be interrupted. (Interrupts on Linux and MAC are still untested).

The debugger is operational and works without performance degradation on large algorithmic Modelica/MetaModelica applications such as the OpenModelica compiler, with more than 100 000 lines of code.

The debugging framework is developed in Eclipse as a plugin that is integrated into our existing OpenModelica Modelica Development Tooling (MDT) (for code browsing and algorithmic code debugging). As mentioned, the OpenModelica Compiler (Peter Fritzson, et al. 2005; Martin Sjölund, Peter Fritzson and Adrian Pop, 2011a) has been adapted to produce the needed line number annotations in the generated C code and run-time system enhancements for GDB communication.

The debugger currently supports the standard primitive Modelica types Integer, Real, Boolean, and String as well as all the additional MetaModelica data types including ragged arrays, lists, and tree data types.

It currently does not support the following regular Modelica data types: arrays, enumerations, and records, which will be added in the near future.

5. CONCLUSIONS AND FUTURE WORK

The debugger is in operation and is being regularly used to debug very large applications such as the OpenModelica compiler with more than 100 000 lines of code. The user experience is very positive. It has been possible to quickly find bugs which previously were very difficult and time consuming to locate. The debugger is very quick even on very large applications, without noticeable delays compared to normal execution.

The most important future work involves developing a suitable integration with run-time debugging of code generated from equation sections in models. Some approaches for such integration are described in (Adrian Pop, David Akhvlediani, and Peter Fritzson, 2007). In such an integrated debugger the recently developed transformation debugging mechanisms (Martin Sjölund and Peter Fritzson 2011b) will be helpful since it provides a mapping from the generated transformed code back to the original equations as well as explanations of the transformations. Such generated code can be very different from the original equations. Some usage experimentation will be needed to determine what run-time debugging features are useful in that context.

However, it will be rather straight-forward to support debugging of algorithm sections intermixed with equation sections in acausal models, since the algorithmic code style of debugging can be used.

6. ACKNOWLEDGEMENTS

This work has been supported by the Open Source Modelica Consortium (OSMC), by VINNOVA in the RTSIM and OPENPROD projects, and by SSF in the ProViking HiPO project.

7. REFERENCES

Adrian Pop and Peter Fritzson (2005). A Portable Debugger for Algorithmic Modelica Code. In Proceedings of the 4th International Modelica Conference, Hamburg, Germany.

Adrian Pop, Peter Fritzson, Andreas Remar, Elmir Jagudin, and David Akhvlediani (2006). OpenModelica Development Environment with Eclipse Integration for Browsing, Modeling, and Debugging. In Proc of the Modelica'2006, Vienna, Austria.

Adrian Pop, David Akhvlediani, and Peter Fritzson (2007). Towards Run-time Debugging of Equation-based Object-oriented Languages. In Proceedings of the 48th Scandinavian Conference on Simulation and Modeling (SIMS'2007), available at www.scan-sims.org and http://www.ep.liu.se. Göteborg, Sweden.

Martin Sjölund, Peter Fritzson, and Adrian Pop (2011a). Bootstrapping a Modelica Compiler aiming at Modelica 4. In Proceedings of the 8th International Modelica Conference (Modelica'2011), Dresden, Germany.

Martin Sjölund and Peter Fritzson (2011b). Debugging Symbolic Transformations in Equation Systems. In Proceedings of the 4th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools, (EOOLT'2011), Zürich, Switzerland.

Peter Bunus and Peter Firtzson (2003). Semi-Automatic Fault Localization and Behavior Verification for Physical System Simulation Models. In Proceedings of the 18th IEEE International Conference on Automated Software Engineering, Montreal, Canada.

Peter Bunus (2004). Debugging Techniques for Equation-Based Languages. PhD Thesis. Department of Computer and Information Science, Linköping University.

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

Peter Fritzson, Peter Aronsson, Håkan Lundvall, Kaj Nyström, Adrian Pop, Levon Saldamli, and David Broman (2005). The OpenModelica Modeling, Simulation, and Software Development Environment. In Simulation News Europe, 44/45.

Richard Stallman, Roland Pesch, Stan Shebs, et al. (2011). Debugging with GDB. Free Software Foundation. [online] Available at: < http://unix.lsa.umich.edu/HPC201/refs/gdb.pdf> [Accessed 30 October 2011].