Modeling, Identification and Control, Vol. 35, No. 2, 2014, pp. 93–107, ISSN 1890–1328

Integrated Debugging of Modelica Models

A. Pop[1] M. Sjölund[1] A. Asghar[1] P. Fritzson[1] F. Casella[2]

[1]*Department of Computer and Information Science, Linköping University, 581 83 Linköping, Sweden. E-mail:* {adrian.pop,martin.sjolund,adeel.asghar,peter.fritzson}@liu.se

[2]*Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milanodi Milano, Via Ponzio 34/5, 20133 Milano, Italy. E-mail:* casella@polimi.it

Abstract

The high abstraction level of equation-based object-oriented (EOO) languages such as Modelica has the drawback that programming and modeling errors are often hard to find. In this paper we present integrated static and dynamic debugging methods for Modelica models and a debugger prototype that addresses several of those problems. The goal is an integrated debugging framework that combines classical debugging techniques with special techniques for equation-based languages partly based on graph visualization and interaction.

To our knowledge, this is the first Modelica debugger that supports both equation-based transformational and algorithmic code debugging in an integrated fashion.

Keywords: Modelica, Debugging, Modeling and Simulation, Transformations, Equations

1 Introduction

Advanced development of today's complex products requires integrated environments and equation-based object-oriented declarative (EOO) languages such as Modelica (Fritzson, 2014; Modelica Association, 2012, 2013) for modeling and simulation. The increased ease of use, the high abstraction level, and the expressivity of such languages are very attractive properties. However, these attractive properties come with the drawback that programming and modeling errors are often hard to find.

To address these issues we present static (compile-time) and dynamic (run-time) debugging methods for Modelica models and a debugger prototype that addresses several of those problems. The goal is an integrated debugging framework that combines classical debugging techniques with special techniques for equation-based languages partly based on graph visualization and user interaction.

The static transformational debugging functionality addresses the problem that model compilers are optimized so heavily that it is hard to tell the origin of an equation during runtime. This work proposes and implements a prototype of a method that is efficient with less than one percent overhead, yet manages to keep track of all the transformations/operations that the compiler performs on the model.

Modelica models often contain functions and algorithm sections with algorithmic code. The fraction of algorithmic code is increasing since Modelica, in addition to equation-based modeling, is also used for embedded system control code as well as symbolic model transformations in applications using the MetaModelica language extension.

Our earlier work in debuggers for the algorithmic subset of Modelica used high-level code instrumentation techniques which are portable but turned out to have too much overhead for large applications. The new dynamic algorithmic code debugger is the first Modelica debugger that can operate without high-level code instrumentation. Instead, it communicates with a low-level C-language symbolic debugger 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 dynamic algorithmic code debugger is operational, supports both standard Modelica data structures and tree/list data structures, and operates efficiently on large applications such as the OpenModelica compiler with more than 200 000 lines of code.

The attractive properties of high-level object-oriented equation-based languages come with the drawback that programming and modeling errors are often hard to find. For example, in order to simulate models efficiently, Modelica simulation tools perform a large amount of symbolic manipulation in order to reduce the complexity of models and prepare them for efficient simulation. By removing redundancy, the generation of simulation code and the simulation itself can be sped up significantly. The drawback of this performance gain is that error-messages often are not very user-friendly due to symbolic manipulation, renaming and reordering of variables and equations. For example, the following error message says nothing about the variables involved or its origin:

```
Error solving non-linear system 2
  time = 0.002
  residual[0] = 0.288956
  x[0] = 1.105149
  residual[1] = 17.000400
  x[1] = 1.248448
  ...
```

It is usually hard for a typical user of the Modelica tool to determine what symbolic manipulations have been performed and why. If the tool only emits a binary executable this is almost impossible. Even if the tool emits source code in some programming language (typically C), it is still quite hard to understand what kind of equation system was produced by the symbolic transformation process. This makes it difficult to understand where the model can be changed in order to improve the speed or stability of the simulation. Some tools allow the user to export the description of the translated system of equations (Casella et al., 2009; Parrotto et al., 2010), but this is not enough. After symbolic manipulation, the resulting equations no longer need to contain the same variables or structure as the original equations.

This work proposes and develops a combination of static and dynamic debugging techniques to address these problems. The static (compile-time) transformational debugging efficiently traces the symbolic transformations throughout the model compilation process and provides explanations regarding the origin of problematic code. The dynamic (run-time) debugging allows interactive inspection of large executable models, stepping through algorithmic parts of the models, setting breakpoints, inspecting and modifying data structures and the execution stack.

An integrated approach is proposed where the mapping from generated code to source code provided by the static transformational debugging is used by the dynamic debugger to relate run-time errors to the original model sources. To our knowledge no other open-source or commercial Modelica tool currently supports static transformational debugging and algorithmic code debugging of an equation-based object-oriented (EOO) language.

The paper is structured as follows: Section 2 gives a background to debugging techniques, Section 3 analyzes sources of errors and faults, Section 4 proposes an integrated static and dynamic debugging approach, Section 5 presents the static transformational debugging method and implementation, whereas Section 6 presents the algorithmic code debugging functionality. Conclusions and future work are given in Section 7.

2 Debugging techniques for EOO Languages

In the context of debugging declarative equation-based object-oriented (EOO) languages such as Modelica, both the static (compile-time) and the dynamic (run-time) aspects have to be addressed.

The static aspect of debugging EOO languages deals with inconsistencies in the underlying system of equations:

1. Errors related to the transformations of the models to an optimized flattened system of equations suitable for numeric solution. For example symbolic solutions leading to division by a constant zero stemming from a singular system of equations or (very rarely) errors in the symbolic transformations themselves.

2. Overconstrained models (too many equations) or underconstrained models (too few equations). The number of variables needs to be equal to the equations is required for solution.

The dynamic (run-time) aspect of debugging EOO languages addresses run-time errors that may appear due to faults in the model:

1. *model configuration*: when the parameters values and start attributes for the model simulation are incorrect.

2. *model specification*: when the equations and algorithm sections that specify the model behavior are incorrect.

3. *algorithmic code*: when the functions called from equations return incorrect results.

Methods for both static and dynamic (run-time) debugging of EOO languages such as Modelica have been proposed earlier by Bunus and Fritzson (2003) and Bunus (2004). With the new Modelica 3.0 language specification, the static overconstrained/underconstrained debugging of Modelica presents a rather small benefit, since all models are required to be balanced. All models from already checked libraries will already be balanced; only newly written models might be unbalanced, which is particularly useful if new models contain a significant number of unknowns.

Regarding dynamic (run-time) debugging of models, Bunus and Fritzson (2003) proposes a semi-automated declarative debugging solution in which the user has to provide a correct diagnostic specification of the model which is used to generate assertions at runtime. Moreover, starting from an erroneous variable value the user explores the dependent equations (a slice of the program) and acts like an "oracle" to guide the debugger in finding the error.

3 Sources of Errors and Faults

There are a number of sources of errors and faults in a simulation system. Some errors can be recovered automatically by the system, whereas others should be reported and allow the users to enter debugging mode. An error can also be a wrong value pointed out manually by a user.

Every solver employed within a simulation system at all levels should be equipped with an error reporting mechanism, allowing error recovery by the master solver, or error reporting to the end-user in case of irrecoverable error:

- the ODE solvers
- the functions computing the derivatives and the algebraic functions given the states, time, and inputs
- the functions computing the initial states and the values of parameters
- the linear equation solvers
- the nonlinear equation solvers

If some equation can be solved symbolically, without resorting to numerical solvers, then the symbolic solution code should be equipped with diagnostics to handle errors as well.

In the next section we give causes of errors that can appear during the model simulation.

3.1 Errors in the evaluation of expressions

During the evaluation of expressions, faults may occur for example due to the following causes:

- Division by zero
- Evaluation of non-integer powers with negative argument
- Functions called outside their domain (e.g.: sqrt(-1), log(-3), asin(2)). For non built-in functions, these errors can be triggered by assertions within the algorithm, or by calls to the pre-defined ModelicaError() function in the body of external functions.
- Errors manifesting as computed wrong value of some variable(s), where the error is manually pointed out by a user or automatically detected as being outside min/max bounds.

3.2 Assertion violations in models

During initialization or simulation, assertions inside models can be triggered when the condition being asserted becomes false.

3.3 Errors in the solution of implicit algebraic equations

During initialization or simulation of DAE systems, implicit equations (or systems of implicit equations, corresponding to strong components in the BLT decomposition) must be solved. In the case of linear systems, the solver might fail because there is some error in evaluating the coefficients of the A matrix and of the b vector of the linear equation $A * x = b$, or because said problem is singular. In the case of nonlinear equations $f(x) = 0$, the solver might fail for several reasons: the evaluation of the residual $f(x)$ or of its Jacobian gives errors; the Jacobian becomes singular; the solver fails to converge after a maximum number of iterations.

3.4 Errors in the integration of the ODEs

In OpenModelica, the DAEs are brought to index-1 ODE form by symbolic and numerical transformation, and these equations are then solved by an ODE solver, which iteratively computes the next state given the current state. During the computation of the next state, for example by using Euler, Runge-Kutta or a BDF algorithm, errors such as those reported in sections 3.1, 3.2, and 3.3 might occur. Furthermore, the solver might fail because of singularity in the ODE, as in the case of finite escape time solutions, or because of discontinuities leading to chattering.

4 Integrated Debugging Approach

In this section we propose an integrated debugging method combining information from a static analysis of the model with dynamic debugging at run-time.

4.1 Integrated Static-Dynamic Debug Method

This method partly follows the approach proposed in Bunus and Fritzson (2003) and Bunus (2004) and further elaborated in Pop et al. (2007). However, our approach does not require the user to write diagnostic specifications of models. The approach we present here can also handle the debugging of algorithmic code using classic debugging techniques.

An overview of this debugging strategy is presented in Figure 1. In short, our run-time debugging method is based on the integration of the following:

1. Dependency graph visualization and interaction.
2. Presentation of simulation results and modeling code.
3. Mapping of errors to model code positions.
4. Execution-based debugging of algorithmic code.

A possible debugging session might be as follows.

During the simulation phase, the user discovers an error in the plotted results, or an irrecoverable error is triggered by the run-time simulation code. In the former case, the user marks either the entire plot of the variable that presents the error or parts of it and starts the debugging framework. The debugger presents an interactive dependency graph (IDG) with respect to the variable with the wrong value or the expression where the fault occurred. The dependency edges in the IDG are computed using the transformation tracing that is described in Section 5. The nodes in the graph consist of all the equations, functions, parameter value definitions, and inputs that were used to calculate the wrong variable value, starting from the known values of states, parameters and time. The variable with the erroneous value (or which cannot be computed at all) is displayed in a special node which is the root of the graph. The IDG contains two types of edges:

1. *Calculation dependency edges*: the directed edges labeled by variables or parameters which are inputs (used for calculations in this equation) or outputs (calculated from this equation) from/to the equation displayed in the node.
2. *Origin edges*: the undirected edges that tie the equation node to the actual model which this equation belongs to.

The user interacts with the dependency graph in several ways:

- *Displaying simulation results* through selection of the variables (or parameters) names (edge labels). The plot of a variable is shown in a popup window. In this way the user can quickly see if the plotted variable has erroneous values.
- *Displaying model code* by following origin edges.
- *Invoking the algorithmic code debugging subsystem* when the user suspects that the result of a variable calculated in an equation which contains a function call is wrong, but the equation seems to be correct.

Using these interactive dependency graph facilities the user can follow the error from its manifestation to its origin. Note that in most cases of irrecoverable errors arising when trying to compute a variable, the root cause of the error does not lie in the equation itself being wrong, but rather in some of the values of previously computed variables appearing in it being wrong, for example because of erroneous initialization or parameterization.

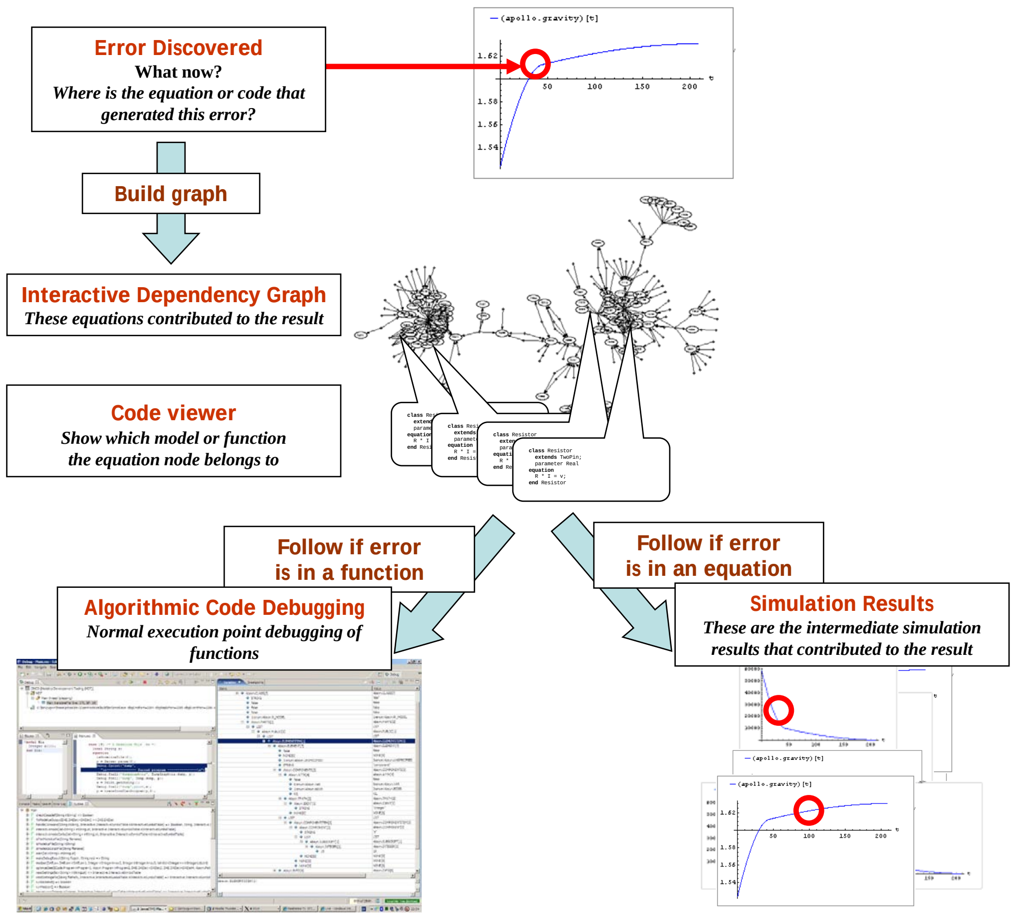
The proposed debugging method can also start from multiple variables with wrong values with the premise that the error might be at the confluence of several dependency graphs.

Note that the debugger can handle both data dependency edges (for example which variables influence the current variable of interest), and origin edges (edges pointing from the generated executable simulation code to the original equations or parts of equations contributing to this code). Both are computed by the transformational debugger mentioned in Section 5.

5 Static Transformational Debugging

Transformational debugging is a static compile-time technique since it does not need run-time execution

Figure 1: Integrated debugging approach overview



Error Discovered
What now?
Where is the equation or code that generated this error?
Build graph
Interactive Dependency Graph
These equations contributed to the result
Code viewer
Show which model or function the equation node belongs to
Follow if error is in a function
Follow if error is in an equation
Algorithmic Code Debugging
Normal execution point debugging of functions
Simulation Results
These are the intermediate simulation results that contributed to the result

of a model. The method keeps track of symbolic transformations, can explain and display applied transformations, and compute dependence edges between the original model and the generated executable code.

5.1 Common Operations on Continuous Equation Systems

In order to create a debugger adapted for debugging the symbolic transformations performed on equation systems, its requirements should be stated. There are many symbolic operations that may be performed on equation systems. The following descriptions of operations also include a rationale for each of them, since it is not always apparent why certain operations are performed. There are of course many more operations that can be performed than the ones listed below, which are however deemed the most important, and which the debugger for models translated by the OpenModelica Compiler (Open Source Modelica Consortium, 2014b) should be able to handle.

5.1.1 Variable aliasing

An optimization that is very common in Modelica compilers is variable aliasing. This is due to the connection semantics of the Modelica language. For example, if a and b are connectors with the potential-variable v and flow-variable i, a connection (1) will generate alias equations (2) and (3).

$$connect(a, b) \tag{1}$$

$$a.v = b.v \tag{2}$$

$$a.i + b.i = 0 \Leftrightarrow b.i = -a.i \tag{3}$$

In a simulation result-file, this alias relation can be stored instead of a duplicate trajectory, saving both space and computation time. In the equation system, $b.v$ may be substituted by $a.v$ and $b.i$ by $-a.i$, which may lead to further optimizations of the equations.

5.1.2 Known variables

Known variables are similar to aliased variables in that the compiler may perform variable substitutions on the rest of the equation system if it finds such an occurrence. For example, (4) and (5) can be combined into (6). In the result-file, there is no longer a need to store the value of a at each time step; once is enough for known variables, which in Modelica are parameters and constants.

$$a = 4.0 \tag{4}$$

$$b = 4.0 - a + c \tag{5}$$

$$b = 4.0 - 4.0 + c \tag{6}$$

5.1.3 Equation Solving

If the tool has determined that x needs to be solved for in (7), it is needed to symbolically solve the equation to produce a simple equation with x on one side as in (8). Solving for x is not always straight-forward and it is not always possible to invert user-defined functions such as called in (9). Since x is present in the call arguments and the tool cannot invert or inline the function, it fails to solve the equation symbolically and instead solves it numerically using a non-linear solver during runtime.

$$15.0 = 3.0 * (x + y) \tag{7}$$

$$x = 15.0/3.0 - y \tag{8}$$

$$0 = f(3 * x) \tag{9}$$

5.1.4 Expression Simplification

Expression simplification is a symbolic operation that does not change the meaning of the expression, while making it faster to calculate. It is related to many different optimization techniques such as constant folding. It is possible to change the order in which arguments are evaluated (10). Constant sub-expressions are evaluated during compile-time (11). Regarding Modelica models it is also allowed to rewrite non-constant sub-expressions (12) and choose to evaluate functions fewer times than in the original expression (13) since functions may not have side-effects. It is also possible for the compiler to use knowledge about the execution model in order to make expressions run faster (14) and (15).

$$and(a, false, b) \Rightarrow and(false, a, b) \Rightarrow false \tag{10}$$

$$4.0 - 4.0 + c \Rightarrow c \tag{11}$$

$$max(a, b, 7.5, a, 15.0) \Rightarrow max(a, b, 15.0) \tag{12}$$

$$f(x) + f(x) + f(x) \Rightarrow 3 * f(x) \tag{13}$$

$$if\ cond\ then\ a\ else\ a \Rightarrow a \tag{14}$$

$$if\ not\ cond\ then\ false\ else\ true \Rightarrow cond \tag{15}$$

5.1.5 Equation System Simplification

It is of course also possible to solve some equation systems statically. For example a linear system of equations with constant coefficients (16) can be solved using one step of symbolic Gaussian elimination (17), generating two separate equations that can be solved individually after causalisation (18). A simple linear equation system such as (16) may also be solved numerically using for example LAPACK (Anderson et al., 1999) routines.

$$\begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 4 \\ 5 \end{pmatrix} \tag{16}$$

$$\begin{pmatrix} 1 & 2 \\ 0 & -3 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 4 \\ -3 \end{pmatrix} \tag{17}$$

$$\begin{aligned} x &= 2 \\ y &= 1 \end{aligned} \tag{18}$$

5.1.6 Differentiation

Symbolic differentiation (Elliott, 2009) is used for many purposes. It is used to symbolically expand known derivatives (19) or as an operation during index reduction. Symbolic Jacobian matrices consisting of derivatives have many applications, for example to speed up simulation runtime (Braun et al., 2011). If there is no symbolic Jacobian available, a numerical one might instead be estimated by the numerical solvers. Such a matrix is often computed using automatic differentiation (Elliott, 2009) which combines symbolic and/or automatic differentiation with other techniques to achieve fast computation.

$$\frac{\partial}{\partial t} t^2 \Rightarrow 2t \tag{19}$$

5.1.7 Index reduction

In order to solve (hybrid) differential algebraic equations (DAEs) numerically, simulation tools use discretisation techniques and methods to numerically compute derivatives and solve differential equations. These parts of the tools are often referred to as solvers. Certain DAEs need to be differentiated symbolically to enable stable numeric solution. The differential *index* of a general DAE system is the minimum number of times that certain equations in the system need to be differentiated to reduce the system to a set of ODEs, which can then be solved by the usual ODE solvers (Fritzson, 2014). While there are techniques to solve DAEs of higher index than 1, most of them require index-1 DAEs or ODEs (no second derivatives). A common index-reduction technique uses dummy derivatives as described by Mattsson and Söderlind (1993). The OpenModelica Compiler default method currently combines the Pantelides (1988) method for index reduction enhanced by Soares and Secchi (2005) with dynamic state selection (Mattsson et al., 2000; Mattsson and Söderlind, 1992, 1993).

5.1.8 Function inlining

Writing functions to perform common operations is a great way to reduce the burden of maintaining code since each operation is defined by a function in only one place. The problem is that for function calls there is some overhead. This becomes a noticeable fraction of the computational cost for the whole invocation and computation for small functions. By inlining a function call (20) and (21), it is treated as a macro expansion (22) which avoids the overhead of calling the function and may increase the number of symbolic manipulations that can performed by the compiler on expressions such as (23).

In Modelica, the compiler may inline the call before or after index reduction. Both methods have advantages and disadvantages. Doing it after index reductions may provide a better result if the modeller has provided an analytic solution in the form of a derivative function. This causes a smaller expression to be differentiated if index reduction is required.

$$f(x, y) = sin(x) + cos(x - y) \tag{20}$$

$$2 * f(x + y, y)/\pi \tag{21}$$

$$2 * \pi * (sin(x + y) + cos(x + y - y))/\pi \tag{22}$$

$$2 * (sin(x + y) + cos(x)) \tag{23}$$

5.1.9 Scalarization

Scalarization is the process of expanding array equations into a number of scalar equations, usually one equation for each element of the corresponding array. By keeping array equations together instead of scalarising them early, the compiler backend saves time since it needs to perform a symbolic operation on only one equation instead of n equations for an array of size n. However, if enough information is known about an equation (24), it can be beneficial to split it into scalar equations, one for each array element (25).

$$(a, b, c) = (x, y, z) \tag{24}$$

$$a = x \quad b = y \quad c = z \tag{25}$$

5.2 Debugging

The choice of techniques for implementation of a debugger depends on where and for what it is intended to be used. Translation and optimization of large application models can be very time-consuming. Thus it would be good if the approach has such a low overhead that it can be enabled by default. It would also be good if error messages from the runtime could use the debug information from the translation and optimization stages to give more understandable and informative messages to the user.

A technique that is commonly used for debugging is tracing. The simplest way of implementing tracing is to print a message to the terminal or file in order to log the operations that are performed. The problem here is that if an operation is rolled back, the log-file will still contain the operation that was rolled back. The data also need to be post-processed if the operations should be grouped by equation.

A more elegant technique is to treat operations as meta data on equations, variables or equation systems. Other meta data that should already be propagated from source code to runtime include the name of the component that an equation is part of, which line and column that the equation originates from, and more. Whenever an operation is performed, the kind of operation and input/output is stored inside the equation as a list of operations. If the structure used to store equations is persistent this also works if the tool needs to roll back execution to an earlier state.

The cost of adding this meta data is a constant runtime factor from storing a new head in the list. The memory cost depends a lot on the compiler itself. If garbage collection or reference counting is used, the only cost is a small amount to describe the operation (typically an integer and some pointers to the expressions involved in the operation).

5.3 Bookkeeping of Operations

5.3.1 Variable Substitution

The elimination of variable aliasing and variables with known values (constants) is considered as the same operation that can be done in a single phase. It can be performed as a fixed-point algorithm where substitutions are collected which record if any change was made (stop if no substitution is performed or no new substitution can be collected). For each alias or known variable, merge the operations stored in the simple equation $x = y$ before removing it from the equation system. For each successful substitution, record it in the list of operations for the equation.

The history of the variable a in the equation system

(26) could be represented as a more detailed version (27) instead of the shorter (28) depending on the order in which the substitutions were performed.

$$a = b, b = -c, c = 4.5 \tag{26}$$

$$a = b \Rightarrow a = -c \Rightarrow a = -4.5 \tag{27}$$

$$a = b \Rightarrow a = -4.5 \tag{28}$$

In equation systems that originate from a Modelica model it is preferable to view a substitution as a single operation rather than as a long chain of operations (chains of 50 cascading substitutions are not unheard of and makes it hard to get an overview of the operations performed on the equation, even though sometimes all the steps are necessary to understand the reason for the final substitution).

It is also possible to collect sets of aliases and select a single variable (doing everything in one operation) in order to make substitutions more efficient. However, alias elimination may still cascade due to simplification rules (29), which means that a work-around is needed for substitutions performed in a non-optimal order.

$$a = b - c + d \Rightarrow a = b - b + d \Rightarrow a = d \tag{29}$$

To efficiently handle this case, the previous operation is compared with the new one and if a link in the chain is detected, this relation is stored. When displaying the operations of an equation system to the user, it is then possible to expand and collapse the chain depending on the user's needs.

5.3.2 Equation Solving

Some equations are only valid for a certain range of input. When solving an equation like (30), is is assumed by the compiler that the divisor is non-zero and it is eliminated in order to solve for x. The compiler records a list of such implicit assertions made (and their data sources for traceability). Such an assertion may be removed if it is later determined that it always holds or if it overlaps with another assertion (31).

$$x/y = 1 \Rightarrow x = y \quad (y \neq 0) \tag{30}$$

$$y \neq 0, 4.0 < y < 8.0 \Rightarrow 4.0 < y < 8.0 \tag{31}$$

5.3.3 Expression Simplification

Tracking changes to an expression is easy if you have a working fixed-point algorithm for expression simplification (record a simplification operation if the simplification algorithm says that the expression changed). However, if the simplification algorithm oscillates (as in 32) there is no canonical form and it is hard to use the simplification procedure as a fixed-point algorithm, which simplifies until nothing changes in the next iteration.

$$2 * x \Rightarrow x * 2 \Rightarrow 2 * x \tag{32}$$

The simple solution is to use an algorithm that is fixed point, or conservative (reporting no change made when performing changes that may cause oscillating behavior). Finding where this behavior occurs is not hard for a compiler developer (simply print an error message after 10 iterations). If it is hard to detect if a change has actually occurred (due to changing data representation to use more advanced techniques), one may need to compare the input and output expression in order to determine if the operation should be recorded. While comparing large expressions may be expensive, it is often possible to let the simplification routine keep track of any changes at a smaller cost.

5.3.4 Equation System Simplification

It is possible to store these operations as pointers to a shared and more "global" operation or as many individual copies of the same operation. We would also recommend to store reverse pointers (or indices) from the global operation back to each individual operation as well, so that reverse lookup can be performed at a low cost.

Since the OpenModelica Compiler only performs limited simplification of strongly connected components, it is currently limited to only recording evaluation of constant linear systems. As more of these optimizations, for example solving for y in (33), are added to the compiler, they will also need to be traced and support for them added in the debugger. Another example would be tracing the tearing operation described by Elmqvist and Otter (1994), which causes the solution of a nonlinear system to be found more efficiently. Support for tearing was recently added to the OpenModelica Compiler but is not yet part of the trace.

$$\begin{pmatrix} 1 & 1 & 2 \\ 1 & i & 1 \\ & -i & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 15 \\ 18 \\ 18 \end{pmatrix} \tag{33}$$

5.3.5 Differentiation

Whenever the compiler performs symbolic differentiation in an expression, for example to expand known derivatives (34), this operation is recorded and associated to the equation in the internal representation. Currently the state variable is not eliminated as in (35), but if it would be done that operation would also be recorded.

$$der(x) = der(time) \Rightarrow der(x) = 1.0 \tag{34}$$

$$der(x) = 1.0 \Rightarrow x = time + (x_{start} - time_{start}) \quad (35)$$

5.3.6 Index reduction

For the used index reduction algorithm, dummy derivatives (Mattsson and Söderlind, 1993), any substitutions made are recorded, source position information is added to the new dummy variable, as well as the operations performed on the affected equations. As an example for the dummy derivatives algorithm, this includes differentiation of the Cartesian coordinates (x, y) of a pendulum with length L (36) into (37) and (38). After the index reduction is complete, further optimizations such as variable substitution (39), are performed to reduce the complexity of the complete system.

$$x^2 + y^2 = L^2 \quad (36)$$

$$der(x^2 + y^2) \Rightarrow 2 * (der(x) * x + der(y) * y) \quad (37)$$

$$der(L^2) \Rightarrow 0.0 \quad (38)$$

$$2 * (der(x) * x + der(y) * y) \Rightarrow 2 * (u * x + v * y) \quad (39)$$

5.3.7 Function inlining

Since inlining functions may cause one or more new function calls to be added to the expression, functions are inlined until there are no more functions to inline in the expression or a maximum recursion depth is reached when dealing with recursive functions. Expressions are also simplified in order to reduce the size of the final expression as well as cope with a few recursive functions that have a known depth after inlining. When the compiler has completed inlining of calls in an equation, this is recorded as an inline operation together with the expression before and after the operation.

5.3.8 Scalarization

When the compiler expands an equation into scalar equations, it stores the initial array expression, the index of the new equation, and the new expression.

5.4 Presentation of Operations

Until now the focus has been on collecting operations as data structured in the equation system. What is possible to do with this information? During the translation phase, it can be used directly to present information to the user. Assuming that the data is well structured, it is possible to store it in a static database (e.g. SQL) or simply as structured data (e.g. XML or JSON). That way the data can be accessed by various applications and presented in different ways according to the user needs for all of them.

Listing 1: Alias Model with Poor Scaling

```
model AliasClass_N
  constant Integer N=60;
  Real a[N];
equation
  der(a[1]) = 1.0;
  a[2] = a[1];
  for i in 3:N loop
    a[i] = i*a[i-1]-sum(a[j] for j
       in 1:i-1);
  end for;
end AliasClass_N;
```

The current OpenModelica prototype outputs XML text at present, soon changed to JSON. In the future this information will be presented in the origin edge introduced in Section 4.

The number of operations stored for each equation varies widely. The reason is that when a known variable, for example x, is replaced by for example the number 0.0, the compiler may start removing subexpressions. It may then end up with a chain of operations that loops over variable substitutions and expression simplification. Frenkel et al. (2011) prove that the number of operations performed may scale with the total number of variables in the equation system if there is no limitation of the number of iterations that the optimizer may take. This makes some synthetic models very hard to debug. The example model in Listing 1 performs $1+2+...+N$ substitutions and simplifications in order to deduce that $a[1] = a[2] = ... = a[n]$.

When testing these methods on a real-world example, the EngineV6 model[1], the majority of equations have less than 4 operations (Figure 2), which means most equations were very easy to solve. The highest number of operations was 16 which is a manageable number to go through when there is a need to debug the model and to understand which equations are problematic. The 16 operations still require a nice presentation, like a visual diff, to quickly get an overview of what happened and why. Note that Figure 2 is a cumulative graph that includes both the initial equation system, the continuous-time equation system, the

[1] Modelica.Mechanics.MultiBody.Examples.Loops.EngineV6 from the MSL (Modelica Association, 2013)

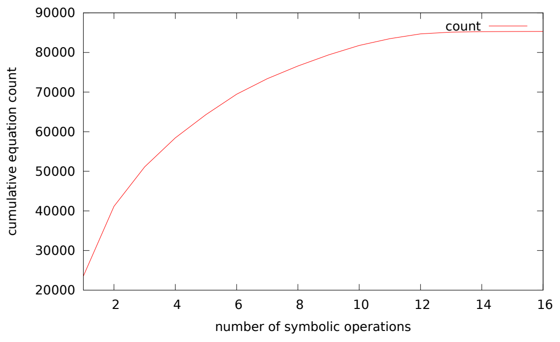


Figure 2: The cumulative number of symbolic operations performed on each equation in the EngineV6 model

discrete-time equation system, and the known variables. Since known variables were included, most of these equations will actually not be part of the generated code and will not be interesting to debug unless it is suspected that the back-end produced the wrong result for a constant equation.

5.5 Runtime supported by static information

In order to produce better error messages during simulation runtime, it would be beneficial to be able to trace the source of the problem. The toy example in Listing 2 is used to show the information that the augmented runtime can display when an error occurs. The user should be presented with an error message from the solver (linear, non-linear, ODE, or algebraic equation does not matter). Here, the displayed error comes from the algebraic equation handling part of the solver. It clearly shows that $log(0.0)$ is not defined and the source position of the error in the concrete textual Modelica syntax form (the Modelica code that the user makes changes for example to fix the problem) as well as the class names of the instances at this point in the instance hierarchy (which may be used as a link by a graphical editor to quickly switch view to the diagram view of this component). The tool also displays the symbolic transformations performed on the equation, which can help in debugging additional problems with the model.

Currently we are working on extending the information we collect during the static analysis to build the Interactive Dependency Graph from Figure 1, Section 4.

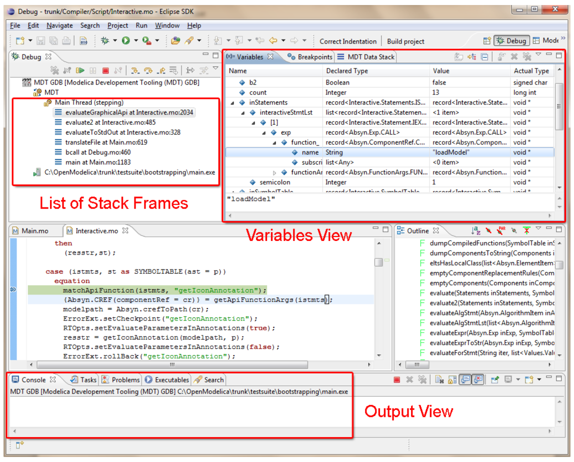
Listing 2: Runtime Error

```
Error: At t=0.5, block1.u = 0.0 is
    not in the domain of log (>0)
Source equation: [Math.mo
    :2490:9-2490:33] y = log(u)
Source component: block1 (MyModel
    Modelica.Blocks.Math.Log)
Flattened equation: block1.y = log(
    block1.u)
Manipulated equation: y = log(u)
  <Operations>
    variable substitution: log(
        block1.u) = log(u)
  <Depending on equations (from BLT)
    >
    u <:link>
```

6 Dynamic Debugging

6.1 Using the Algorithmic Code Debugger

The debugger part for algorithmic Modelica code is implemented within the OpenModelica environment 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 Figure 3.



```
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
```

Name	Declared Type	Value	Actual Type
b2	Boolean	false	signed char
count	Integer	13	long int
inStatements	record<Interactive.Statements.IS...	record<Interactive.State...	void *
interactiveStmtLst	list<record<Interactive.Statemen...	<1 item>	void *
[1]	record<Interactive.Statement.IEX...	record<Interactive.State...	void *
exp	record<Absyn.Exp.CALL>	record<Absyn.Exp.CALL>	void *
function_	record<Absyn.ComponentRef.C...	record<Absyn.Compon...	void *
name	String	"loadModel"	void *
subscri	list<Any>	<0 item>	void *
functionAr	record<Absyn.FunctionArgs.FUN...	record<Absyn.Function...	void *
semicolon	Integer	1	void *

```
"loadModel"
```

```
      then
        (resstr,st);

    case (istmts, st as SYMBOLTABLE(ast = p))
      equation
        matchApiFunction(istmts, "getIconAnnotation");
        {Absyn.CREF(componentRef = cr)} = getApiFunctionArgs(istmts);
        modelpath = Absyn.crefToPath(cr);
        ErrorExt.setCheckpoint("getIconAnnotation");
        RTOpts.setEvaluateParametersInAnnotations(true);
        resstr = getIconAnnotation(modelpath, p);
        RTOpts.setEvaluateParametersInAnnotations(false);
        ErrorExt.rollBack("getIconAnnotation");
```

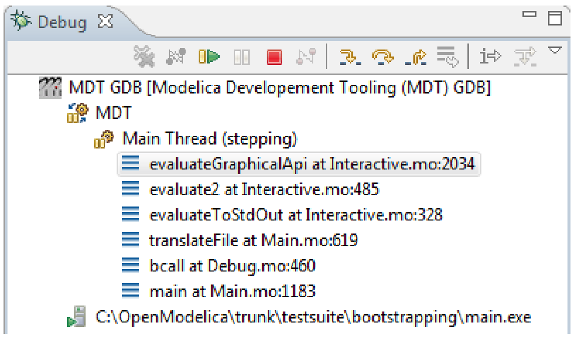
```
dumpCompiledFunctions(SymbolTable in
dumpComponentsToString(Components i
eltsHasLocalClass(list<Absyn.ElementItem
emptyComponentReplacementRules(Com
emptyComponents(Components inComp
evaluate(Statements inStatements, Symbol
evaluate2(Statements inStatements, Symb
evaluateAlgStmt(Absyn.AlgorithmItem inA
evaluateAlgStmtLst(list<Absyn.AlgorithmI
evaluateExpr(Absyn.Exp inExp, SymbolTabl
evaluateExprToStr(Absyn.Exp inExp, Symbo
evaluateForStmt(String iter, list<Values.Val
```

```
MDT GDB [Modelica Developement Tooling (MDT) GDB] C:\OpenModelica\trunk\testsuite\bootstrapping\main.exe
```

Figure 3: The debug view of the new efficient algorithmic code debugger within the MDT Eclipse plugin.

The algorithmic code debugger provides the following general functionalities:

- Adding/Removing breakpoints.
- Step Over – moves to the next line, skipping the function calls.



```
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
```

Figure 4: The stack frame view of the debugger.

- 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.

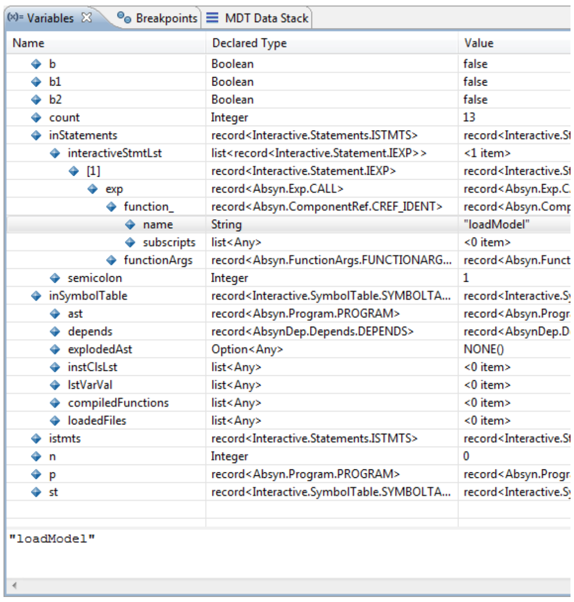
It is much faster and provides several stepping options compared to the old dynamic debugger described by Pop (2008) because the old debugger was based on high-level source code instrumentation which made the code grow by a factor of the number of variables. The debug view primarily consists of two main views:

- Stack Frames View
- Variables View

The stack frame view, shown in Figure 4, shows a list of frames that indicates how the flow has 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>. The Variables view (Figure 5) 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.



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

```
"loadModel"
```

Figure 5: The variable view of the new debugger.

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).

6.2 Dynamic Debugger Implementation

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 (Stallman et al., 2014).

Figure 6 shows how the bootstrapped OpenModelica Compiler (Sjölund et al., 2014) propagates the line number information 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 by Sjölund and Fritzson (2011) and is also used in this debugger.

Consider the Modelica code shown in Figure 7. The OpenModelica Compiler compiles this HelloWorld function into the intermediate C source-code depicted in Figure 8. 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]*/. The generated intermediate C source-code is used to create another version of the same source-code with standard C line pre-

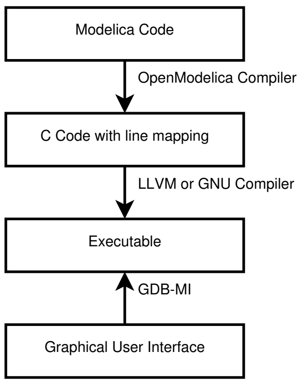
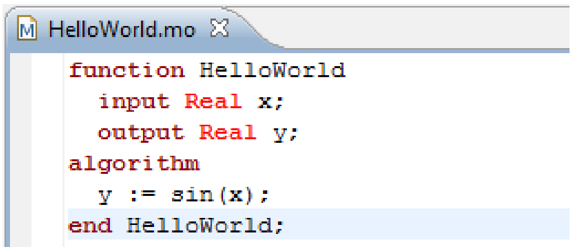
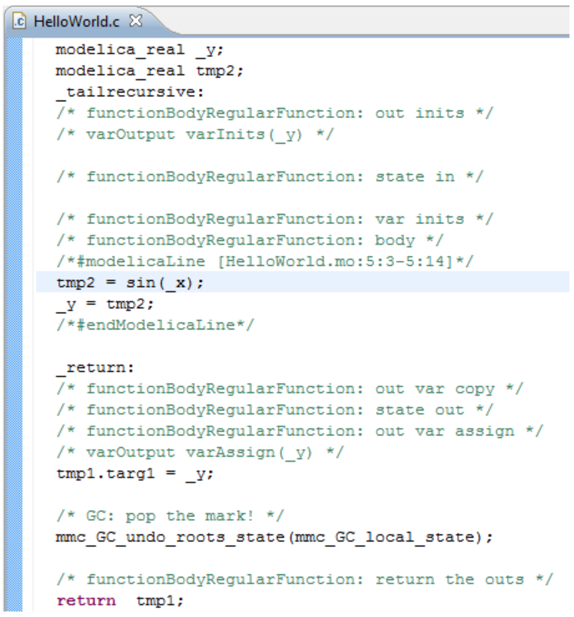


Figure 6: Dynamic debugger flow of control.



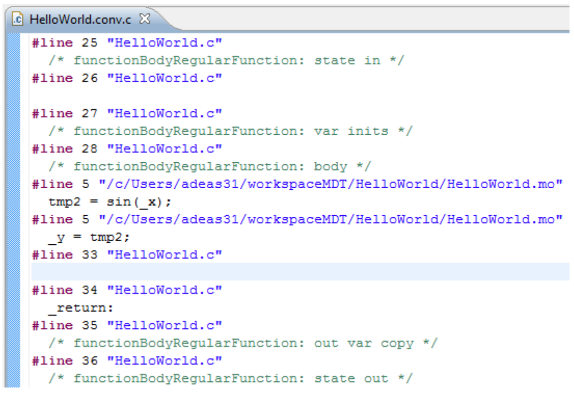
```
function HelloWorld
  input Real x;
  output Real y;
algorithm
  y := sin(x);
end HelloWorld;
```

Figure 7: Modelica Code.



```
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;
```

Figure 8: Generated C source-code.



```
#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 */
```

Figure 9: Converted C source-code.

processor directives, see Figure 9.

The executable is created from the converted C source-code and is debugged from the Eclipse-based Modelica debugger which converts Modelica-related commands to low-level GDB commands at the C code level.

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>.

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 working by default.

The algorithmic code debugger is operational and works without performance degradation on large algorithmic Modelica/MetaModelica applications such as the OpenModelica compiler, with more than 200 000 lines of code.

The algorithmic code debugging framework graphical user interface is developed in Eclipse as a plugin that is integrated into the existing OpenModelica Modelica Development Tooling (MDT). The tracking of line number information and the runtime part of the debugging framework is implemented as part of the OpenModelica compiler and its simulation runtime.

The algorithmic code debugger currently supports the standard Modelica data types including arrays and records as well as all the additional MetaModelica data types such as ragged arrays, lists, and tree data types. It supports algorithmic code debugging of both simulation code and MetaModelica code.

Furthermore, in order to make the debugging practical (as a function could be evaluated in a time step several hundred times) the debugger supports conditional breakpoints based on the time variable and/or hit count.

The algorithmic code debugger can be invoked from the model evaluation browser and breaks at the execution of the selected function to allow the user to debug its execution.

7 Conclusions and Future Work

We have presented static and dynamic debugging methods to bridge the gap between the high abstraction level of equation-based object-oriented models compared to generated executable code. Moreover, an overview of typical sources of errors and possibilities for automatic error handling in the solver hierarchy has been presented.

Regarding static transformational debugging, a prototype design and implementation for tracing symbolic transformations and operations has been made in the OpenModelica Compiler. It is very efficient with an overhead of the order of 0.01% if the collected information is not output to file.

Regarding dynamic algorithmic code debugging, this part of the debugger is in operation and is being regularly used to debug very large applications such as the OpenModelica compiler with more than 200 000 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 and efficient even on very large applications, without noticeable delays compared to normal execution.

A design for an integrated static-dynamic debugging has been presented, where the dependency and origin information computed by the transformational debugger is used to map low-level executable code positions back to the original equations. Realizing the integrated design is work-in-progress and not yet completed.

To our knowledge, this is the first debugger for Modelica that has both static transformational symbolic debugging and dynamic algorithmic debugging.

The tracing of symbolic operations as described in Section 5 is available in the 1.9.0 release of OpenModelica (Open Source Modelica Consortium, 2014b). Nightly builds and development releases of OpenModelica contain a graphical user interface to better browse the transformations. You can download packages for the most common operating systems from `https://openmodelica.org/` or compile from source.

The algorithmic debugger is part of MDT (Open Source Modelica Consortium, 2014a) and can be installed by following the instructions at `https://trac.openmodelica.org/MDT/`. Moreover, there is ongoing work to make both the algorithmic code debugger and the equation model debugger from the OMEdit graphical user interface.

Acknowledgments

This work has been supported by the Swedish Strategic Research Foundation in the EDOp and HIPo projects and Vinnova in the RTSIM and ITEA2 OPENPROD and MODRIO projects. The Open Source Modelica Consortium supports the OpenModelica work.

References

Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., and Sorensen, D. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

Braun, W., Ochel, L., and Bachmann, B. Symbolically derived Jacobians using automatic differentiation - enhancement of the OpenModelica compiler. In Clauß (2011), 2011. doi:10.3384/ecp11063.

Bunus, P. *Debugging techniques for Equation-Based languages*. Doctoral thesis No 873, Linköping University, Department of Computer and Information Science, 2004. URL `http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-35555`.

Bunus, P. and Fritzson, P. Semi-automatic fault localization and behavior verification for physical system simulation models. In *18th IEEE International Conference on Automated Software Engineering*. pages 253–258, 2003. doi:10.1109/ASE.2003.1240315.

Casella, F., Donida, F., and Åkesson, J. An XML representation of DAE systems obtained from Modelica models. In F. Casella, editor, *Proceedings of the 7th International Modelica Conference*. Linköping University Electronic Press, 2009. doi:10.3384/ecp09430073.

Clauß, C., editor. *Proceedings of the 8th International Modelica Conference*. Linköping University Electronic Press, 2011.

Elliott, C. M. Beautiful differentiation. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09. ACM, New York, NY, USA, pages 191–202, 2009. doi:10.1145/1596550.1596579.

Elmqvist, H. and Otter, M. Methods for tearing systems of equations in object-oriented modelling. In A. Guasch and R. M. Huber, editors, *Proceedings of the European Simulation Multiconference*, ESM'94. Society for Computer Simulation, pages 326–332, 1994. URL `http://www.robotic.de/fileadmin/control/shared/publications/1994/elmqvist_esm.ps.gz`.

Frenkel, J., Schubert, C., Kunze, G., Fritzson, P., Sjölund, M., and Pop, A. Towards a benchmark suite for Modelica compilers: Large models. In Clauß (2011), 2011. doi:10.3384/ecp11063232.

Fritzson, P. *Principles of Object-Oriented Modeling and Simulation with Modelica 3.3: A Cyber-Physical Approach*. Wiley-IEEE Press, 2014.

Mattsson, S. E., Olsson, H., and Elmqvist, H. Dynamic selection of states in Dymola. In M. Otter, editor, *Modelica Workshop 2000*. Modelica Association, pages 61–67, 2000. URL `https://modelica.org/events/workshop2000/proceedings/Mattsson.pdf`.

Mattsson, S. E. and Söderlind, G. A new technique for solving high-index differential-algebraic equations using dummy derivatives. In *Computer-Aided Control System Design, 1992. (CACSD), 1992 IEEE Symposium on*. pages 218–224, 1992. doi:10.1109/CACSD.1992.274429.

Mattsson, S. E. and Söderlind, G. Index reduction in differential algebraic equations using dummy derivatives. *Siam Journal on Scientific Computing*, 1993. 14:677–692. doi:10.1137/0914043.

Modelica Association. Modelica: A unified object-oriented language for physical systems modeling, language specification version 3.3. 2012. URL `http://www.modelica.org/`.

Modelica Association. Modelica Standard Library version 3.2.1. 2013. URL `https://modelica.org/libraries`.

Open Source Modelica Consortium. MDT – Modelica Development Tooling. 2014a. URL `https://trac.openmodelica.org/MDT/`.

Open Source Modelica Consortium. Openmodelica. 2014b. URL `https://openmodelica.org/`.

Pantelides, C. C. The consistent initialization of differential-algebraic systems. *SIAM Journal on Scientific and Statistical Computing*, 1988. 9(2):213–231. doi:10.1137/0909014.

Parrotto, R., Åkesson, J., and Casella, F. An XML representation of DAE systems obtained from continuous-time Modelica models. In P. Fritzson, E. Lee, F. Cellier, and D. Broman, editors, *Proceedings of the 3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*. Linköping University Electronic Press, pages 91–98, 2010. URL `http://www.ep.liu.se/ecp/047/`.

Pop, A. *Integrated Model-Driven Development Environments for Equation-Based Object-Oriented Languages*. Doctoral thesis No 1183, Department of Computer and Information Science, Linköping University, Sweden, 2008.

Pop, A., Akhvlediani, D., and Fritzson, P. Towards run-time debugging of equation-based object-oriented languages. In *Proceedings of the 48th Scandinavian Conference on Simulation and Modeling (SIMS)*. 2007.

Sjölund, M. and Fritzson, P. Debugging symbolic transformations in equation systems. In F. Cellier, D. Broman, P. Fritzson, and E. Lee, editors, *Proceedings of the 4th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*. Linköping University Electronic Press, 2011. URL `http://www.ep.liu.se/ecp_home/index.en.aspx?issue=056`.

Sjölund, M., Fritzson, P., and Pop, A. Bootstrapping a Compiler for an Equation-Based Object-Oriented Language. *Modeling, Identification and Control*, 2014. 35(1):1–19. doi:10.4173/mic.2014.1.1.

Soares, R. P. and Secchi, A. R. Direct initialisation and solution of high-index dae systems. In L. Puigjaner and A. Espuña, editors, *European Symposium on Computer-Aided Process Engineering-15, 38th European Symposium of the Working Party on Computer Aided Process Engineering*, volume 20 of *Computer Aided Chemical Engineering*, pages 157–162. Elsevier, 2005. doi:10.1016/S1570-7946(05)80148-8.

Stallman, R., Pesch, R., Shebs, S., et al. *Debugging with GDB*. Free Software Foundation, 2014. URL http://www.gnu.org/software/gdb/documentation/.