Static and Dynamic Debugging of Modelica Models

Adrian Pop[1], Martin Sjölund[1], Adeel Asghar[1], Peter Fritzson[1], Francesco Casella[2]
[1]Programming Environments Laboratory
Department of Computer and Information Science
Linköping University, Linköping, Sweden
[2]Dipartimento di Elettronica e Informazione, Politecnico di Milano, Milano, Italy
{adrian.pop,martin.sjolund,adeel.asghar,peter.fritzson}@liu.se
casella@elet.polimi.it

Abstract

The high abstraction level of equation-based object-oriented languages (EOO) such as Modelica has the drawback that programming and modeling errors are often hard to find. In this paper we present 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 transformational and algorithmic code debugging.

Keywords: Modelica, Debugging, Modeling and Simulation, Transformations, Equations, Algorithmic Code, Eclipse

1 Introduction

Advanced development of today's complex products requires integrated environments and equation-based object-oriented declarative (EOO) languages such as Modelica [8][12] for modeling and simulation. The increased ease of use, the high abstraction, 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 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 100 000 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 a large number 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 cost of this performance gain is error-messages that 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 nonlinear 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 know what kind of equation system you have ended up with. 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 [18], 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 to 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 origin mapping provided by the static transformational debugging is used by the dynamic debugger to relate runtime errors to the original model sources. To our knowledge no other open-source or commercial Modelica tool currently supports static transformational debugging or algorithmic code debugging.

The paper is structured as follows: Section 2 the background and related work, 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 Background and Related Work

2.1 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, e.g. 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 [6][7]. 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 [6] 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 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 $Ax = b$, 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, e.g. by using Euler, Runge-Kutta or a BDF algorithm, errors such as those reported in section 3.1, 3.2, 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 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 [6][7] and further elaborated in [3]. However, our approach does not require the user to write diagnostic specifications of models. Also, 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.

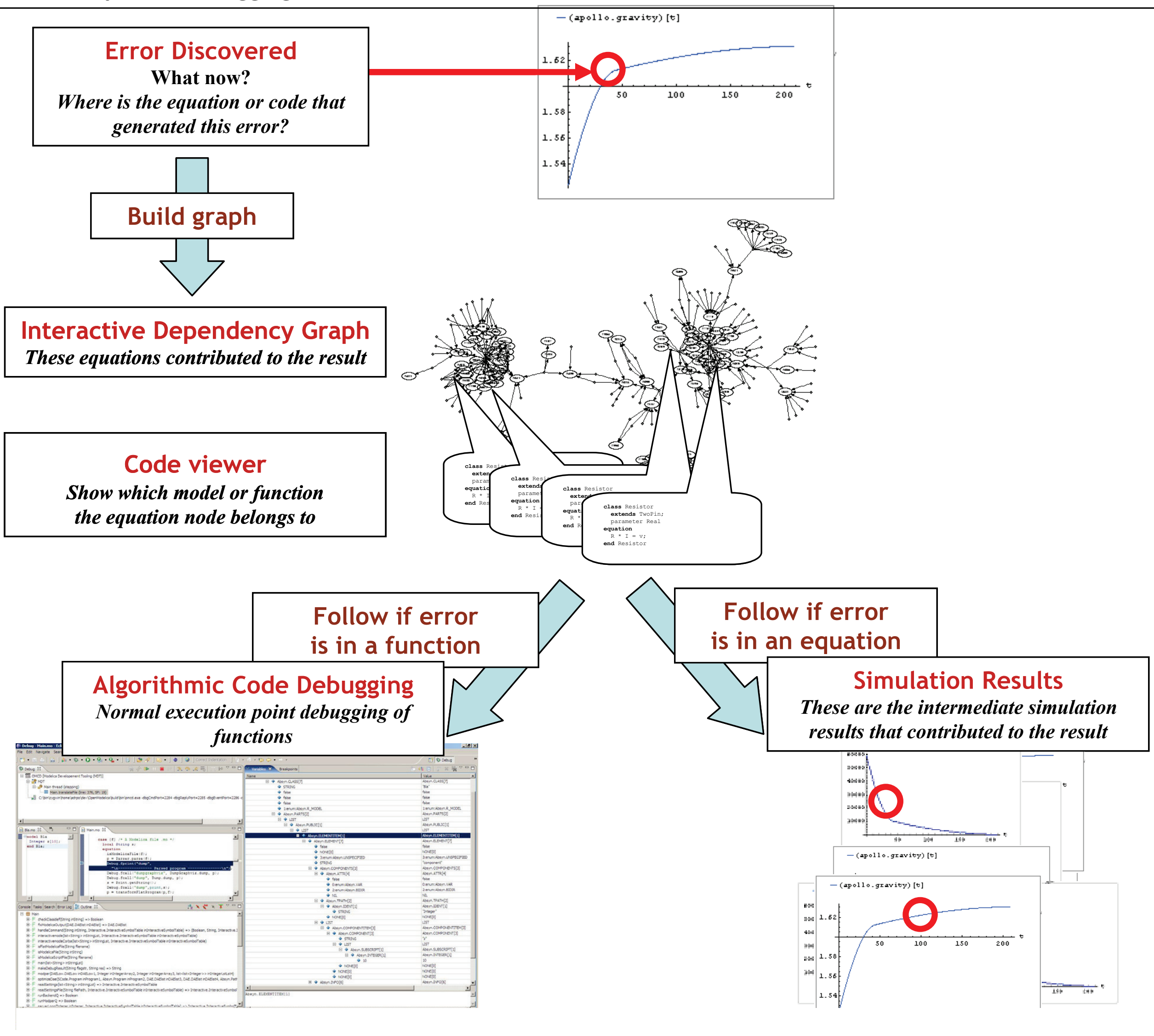

Error Discovered
What now?
Where is the equation or code that generated this error?
Build graph
Interactive Dependency Graph
These equations contributed to the result
Code viewer
Show which model or function the equation node belongs to
Follow if error is in a function
Follow if error is in an equation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

Figure 1. Integrated debugging approach overview.

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 (IDG) interactive dependency graph with respect to the variable with the wrong value or the expression where the fault occurred. The dependency edges in 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, e.g., 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 (e.g. which variables influence the current variable of interest), and origin edges (edges pointing from the generated executable simulation code to the original equations/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 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 perform certain operations are performed. There are of course many more operations that can be performed than the ones listed below, which are however deemed most important, and which the debugger for models translated by the OpenModelica Compiler [11] 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 effort-variable `v` and flow-variable `i`, a connection (2) will generate alias equations (3) and (4).

```
connect(a, b)                        (2)
a.v = b.v                            (3)
a.i + b.i = 0 ⇒ b.i = -a.i           (4)
```

In a 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.v`, which may lead to further optimizations of the equations.

5.1.2 Known variables

Known variables are similar to alias variables in that you may perform variable substitutions on the rest of the equation system if you find such an occurrence. For example, (5) and (6) can be combined into (7). In the result-file, you no longer need to store a value for each time step; once is enough for known variables (which have values that can be computed statically at compile-time), parameters and constants.

```
a = 4.0                              (5)
b = 4.0 - a + c                      (6)
b = 4.0 - 4.0 + c                    (7)
```

5.1.3 Equation Solving

If the tool has determined that `x` needs to be solved for in (8), we need to symbolically solve the equation, producing a simple equation with `x` on one side as in (9). Solving for `x` is not always straightforward, and it is not always possible to invert user-defined functions such as (10). Since `x` is present in the call arguments and the function cannot be inverted or inlined, it is not possible to solve the equation symbolically, so it is necessary to resort to a numerical non-linear solver during runtime.

```
15.0 = 3.0*(x + y)                   (8)
x = 15.0/3.0 - y                     (9)
0 = f(3*x)                           (10)
```

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. The order in which arguments are evaluated may be changed (11). Constant subexpressions are evaluated during compile-time (12). Non-constant subexpressions may be rewritten (13) and functions may be evaluated fewer times than in the original expression (14). It is also possible to use special knowledge about an expression in order to make it run faster (15) and (16).

```
and(a,false,b) ⇒ false                  (11)
4.0 - 4.0 + c ⇒ c                       (12)
max(a,b,7.5,a,15.0) ⇒ max(a,b,15,0)     (13)
f(x) + f(x) + f(x) ⇒ 3*f(x)             (14)
if cond then a else a ⇒ a               (15)
if cond then false else true ⇒ cond     (16)
```

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 (17) can be solved using one step of symbolic Gaussian elimination (18), generating two separate equations that can be solved individually after causalization (19). A simple linear equation system as (17) may also be solved numerically using e.g. LAPACK [1] routines.

```
[1, 2; 2, 1] * [x; y] = [4; 5]          (17)
[1, 2; 0,-3] * [x; y] = [4; -3]         (18)
x = 2; y = 1;                           (19)
```

5.1.6 Differentiation

Symbolic differentiation [16] is used for many purposes. It is used to expand known derivatives (20) or as one operation in index reduction. Jacobian matrices have many applications, e.g. to speed up simulation runtime [14]. The matrix is often computed using automatic differentiation [14][16] which combines symbolic differentiation with other techniques to achieve fast computation.

```
  der(t^2, t) = 2*t                     (20)
```

5.1.7 Index reduction

In order to solve DAE's numerically, discretization techniques and methods to numerically compute derivatives are used (often referred to as solvers). Certain DAE's need to be differentiated symbolically to enable a 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, Chapter 18 in [8]. While there are techniques to solve DAE's of higher index than 1, most of them require index-1 DAE's (no second derivatives). This makes it more convenient to reformulate the problem using index reduction algorithms, Chapter 18 in [8]. One such technique uses dummy derivatives [15]; this is the algorithm currently used in the OpenModelica Compiler.

5.1.8 Function inlining

Writing functions to do common operations is a great way to reduce the burden of maintaining code. When a function call is inlined (21), it can be treated as a macro expansion (22) and may increase the number of symbolical manipulations that can be perform on an expression such as (23).

```
2*f(x, y)/pi                            (21)
2*pi*((sin(x+y)+cos(x+y-y)/pi           (22)
2*(sin(x+y) + cos(x))                   (23)
```

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 you perform. 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 metadata on equations, variables or equation systems. Other metadata 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 operation kind 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 (24) could be represented as a more detailed version (25) instead of the shorter (26) depending on the order in which the substitutions were performed.

```
a = b; b = -c; c = 4.5                        (24)
a = b ⇒ a = -c ⇒ a = -4:5                     (25)
a = b ⇒ a = -4.5                              (26)
```

In equation systems that originate from a Modelica model it is preferable to see a substitution as a single operation rather than a longer 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 (27), which means that you need a work-around for substitutions performed in a non-optimal order.

```
a = b - c + d ⇒ a = b - b + d
⇒ a = d                                      (27)
```

Thus, we compare the previous operation with the new one and if we detect a link in the chain, we store this relation. When displaying the operations of an equation system, 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 (28), you assert that the divisor is non-zero and eliminate it in order to solve for x. We record a list of the assertions made (and their sources for traceability). An assertion may be removed if we later determine that it always holds or if it overlaps with another assertion (29).

```
x/y = 1 ⇒ x = y (y != 0)                      (28)
y!=0, 4.0 < y < 8.0 ⇒ 4.0 < y < 8.0           (29)
```

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 30) it is hard to use it as a fixed-point algorithm.

```
2*x ⇒ x*2 ⇒ 2*x ⇒ ...                        (30)
```

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. It is preferable to store this as a single global operation (see Figure 2) since the only cost is only some indirection when reading the data. It is also recommended 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.

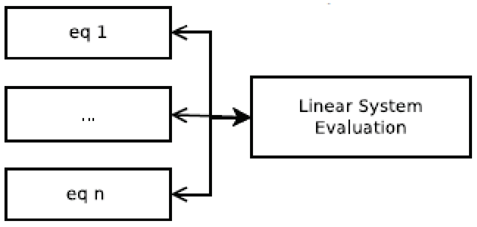


Figure 2. Sharing Results of Linear System Evaluation.

As the tool we are using performs only limited simplification of these strongly connected components, we are currently limited to only recording evaluation of constant linear systems. As more of these optimizations are added to the compiler, they will also need to be traced and support added for them in the debugger.

5.3.5 Differentiation

Whenever we perform symbolic differentiation in an expression, e.g. to expand known derivatives (31), we record this operation in the equation. OpenModelica currently does not eliminate this state variable as in (32), but if it did the operation would also be recorded.

```
der(x) = der(time) ⇒ der(x) = 1.0        (31)
der(x) = 1.0 ⇒
  x = time + (xstart-timestart)          (32)
```

5.3.6 Index reduction

For the index reduction algorithm, any performed substitution is recorded, source information is added to the newly introduced dummy derivative variable, and the operations are 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` (33) into (34) and (35). After the index reduction is complete, further optimizations such as variable substitution (37), are performed to reduce the complexity of the complete system.

```
x^2 + y^2 = L^2                              (33)
der(x^2 + y^2) ⇒ 2*(der(x)*x + der(y)*y)
                                             (34)
der(L^2) ⇒ 0                                 (35)
2*(der(x)*x + der(y)*y) ⇒ 2*(u*x + v*y)
                                             (36)
```

5.3.7 Function inlining

Since inlining functions may cause a new function call to be added to the expression, functions are inlined until a fixed point is reached (with a maximum depth to avoid problems with recursive functions). Expressions are also simplified in order to reduce the size of the final expression. When inlining calls in an equation have been completed, this is recorded as an inline operation with the expression before and after.

5.4 Presentation of Operations

Until now the focus has been on collecting operations as data structured in the equation system. What is it 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). That way the data can be accessed by various applications and presented in different ways according to the user needs for all of them. The current OpenModelica prototype only outputs text at present; in the future this information will be presented in the origin edge introduced in Section4.

The number of operations stored for each equation varies widely. The reason is that when a known variable `x` is replaced with, e.g., the number 0.0, one may start removing subexpressions. One then ends up with a chain of operations that loops over variable substitutions and expression simplification. The number of operations performed may scale with the total number of variables in the equation system if the the number of iterations that the optimizer may take is not limited [17]. 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].

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;
```

Using a real-world example, the Engine1a model from the Modelica MultiBody library, [12], the majority of equations have less than 10 operations (Figure 3), which is a manageable number to go through if one needs to debug a model and to find out which equations are problematic.

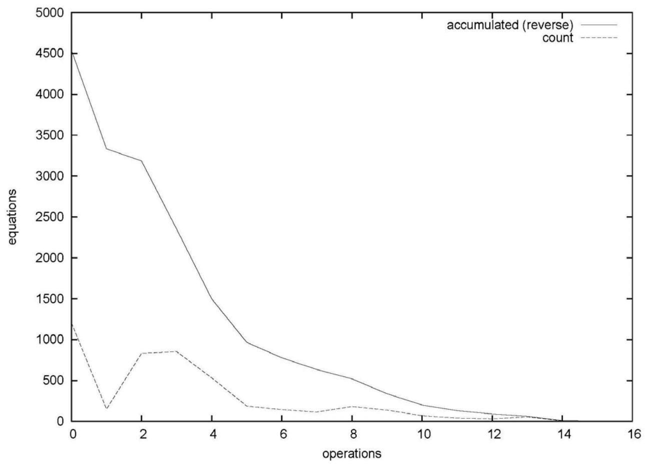


Figure 3. The number of symbolic operations performed on equations in the Engine1a model.

5.5 Runtime supported by static information

In order to produce better error messages during runtime, it is 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, nonlinear, ODE or algebraic does not matter). Here, the displayed error comes from the algebraic part of the solver. It clearly shows that log(0.0) is not defined and the source of the error in the concrete syntax (the Modelica code that the user may influence) as well as the name of the component (which may be used as a link by a graphical editor to quickly switch view to the diagram view of this component). The symbolic transformations performed on the equation are also displayed, which can help debug the model better.

Listing 2. Runtime Error

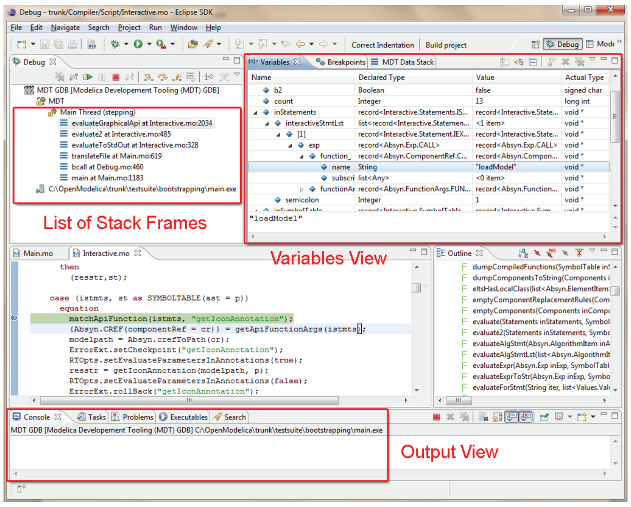
```
Error: At t=0.5, block1.u = 0.0 is not in
the domain of log (>0)
Source equation: [Math.mo :2490:9-2490:33]
y = log(u)
Source component: block1 (MyModel
Modelica.Blocks.Math.Log)
Flattened equation: block1.y = log(
block1.u)
Manipulated equation: y = log(u)
<Operations>
variable substitution: log(block1.u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

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.

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 4.



```
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");
```

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

Figure 4. The debug view of the new efficient algorithmic code debugger within the MDT Eclipse plugin.

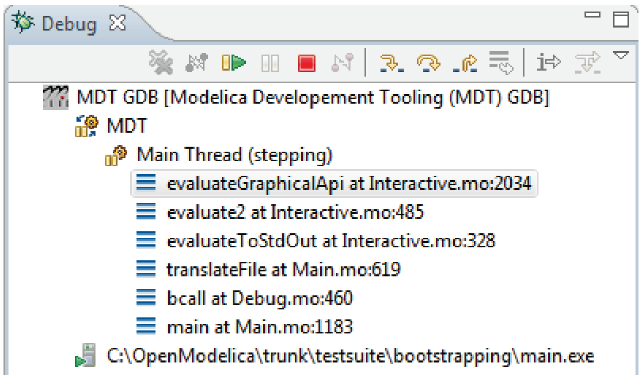
The algorithmic code debugger provides the following general functionalities:

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

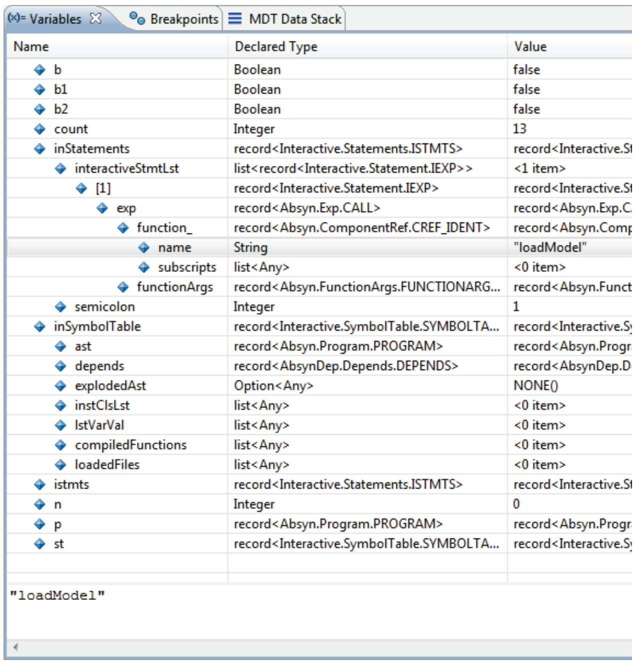
- Stack Frames View
- Variables View

The stack frame view, shown in Figure 5, 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.



```
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 5. The stack frame view of the debugger.



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.S
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.S

```
"loadModel"
```

Figure 6. The variable view of the new debugger.

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 6) 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).

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 [10].

Through the bootstrapped OpenModelica Compiler [4] 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 [5] and is also used in this debugger.

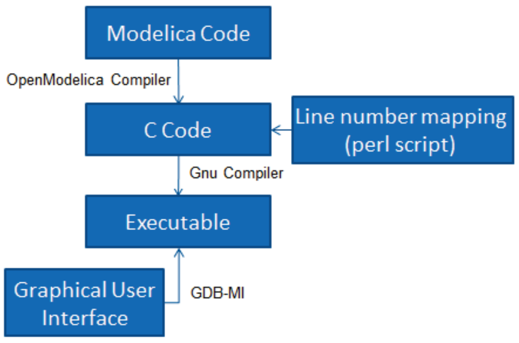


Figure 7. Dynamic debugger flow of control.

Consider the Modelica code shown in Figure 8:

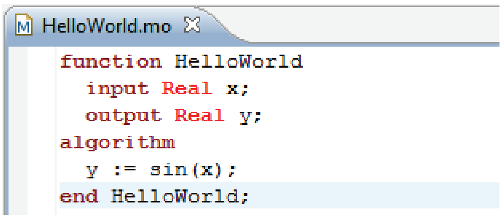

```
HelloWorld.mo
function HelloWorld
  input Real x;
  output Real y;
algorithm
  y := sin(x);
end HelloWorld;
```

Figure 8. Modelica Code.

The OpenModelica Compiler compiles this HelloWorld function into the C source-code depicted in Figure 9.

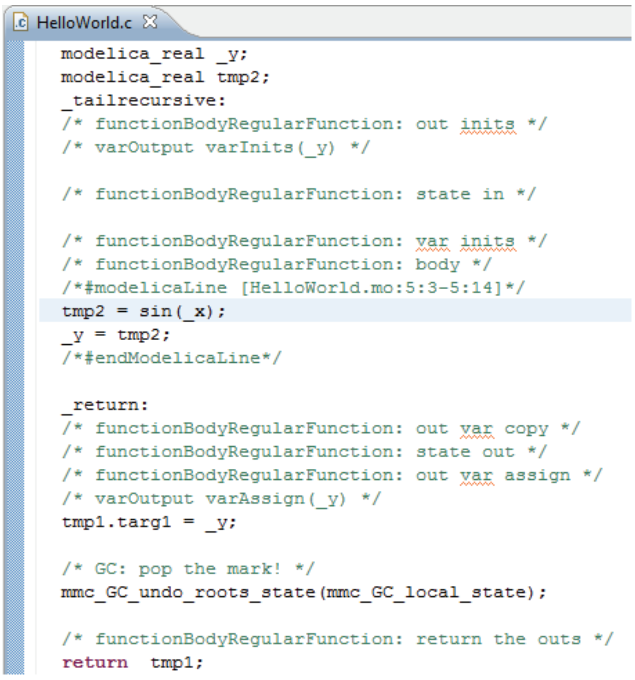

```
HelloWorld.c
  modelica_real _y;
  modelica_real tmp2;
  _tailrecursive:
  /* functionBodyRegularFunction: out inits */
  /* varOutput varInits(_y) */

  /* 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;
```

Figure 9. Generated C source-code.

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 Figure 10.

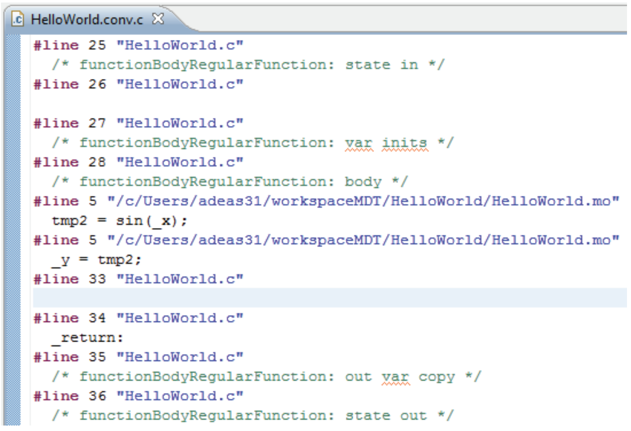

```
HelloWorld.conv.c
#line 25 "HelloWorld.c"
  /* functionBodyRegularFunction: state in */
#line 26 "HelloWorld.c"

#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 */
```

Figure 10. 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 Figure 10.

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 100 000 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 it 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%.

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 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 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.

8 Acknowledgements

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 projects. The Open Source Modelica Consortium supports the OpenModelica work.

References

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

[2] 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.

[3] 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), see http://www.scan-sims.org, http://www.ep.liu.se. Göteborg, Sweden.

[4] 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.

[5] 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.

[6] Peter Bunus and Peter Fritzson (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.

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

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

[9] 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.

[10] 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].

[11] Open Source Modelica Consortium. *OpenModelica System Documentation Version 1.8.1*, April 2012. http://www.openmodelica.org

[12] Modelica Association. *The Modelica Language Specification Version 3.2*, March 24th 2010. http://www.modelica.org. Modelica Association. *Modelica Standard Library 3.1*. Aug. 2009. http://www.modelica.org.

[13] Uri Ascher and Linda Petzold. Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations. Society for Industrial and Applied Mathematics, 1998.

[14] Willi Braun, Lennart Ochel, and Bernhard Bachmann. Symbolically derived Jacobians using automatic differentiation - enhancement of the OpenModelica compiler. In Modelica'2011.

[15] Sven Erik Mattsson and Gustaf Söderlind. Index reduction in differential algebraic equations using dummy derivatives. Siam Journal on Scientific Computing, 14:677--692, May 1993.

[16] Conal Elliott. Beautiful differentiation. In International Conference on Functional Programming (ICFP), 2009.

[17] Jens Frenkel, Christian Schubert, Günter Kunze, Peter Fritzson, and Adrian Pop. Towards a benchmark suite for Modelica compilers: Large models. In Modelica' 2011.

[18] Roberto Parrotto, Johan Åkesson, and Francesco Casella. An XML representation of DAE systems obtained from continuous-time Modelica models. In Proceedings of the 3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools, pages 91--98. Linköping University Electronic Press, October 2010.