Debugging Symbolic Transformations in Equation Systems

Martin Sjölund[1] Peter Fritzson[1]

[1]Department of Computer and Information Science, Linköping University, Sweden,
{martin.sjolund,peter.fritzson}@liu.se

Abstract

How do you debug application models in an equation-based object-oriented (EOO) programming language? Compilers for these tools tend to optimize the model 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, yet manages to keep track of all the transformations/operations that the compiler performs on the model. The method also considers the ability to collapse certain operations so that they appear to the user as a single expandable operation.

Using such a method enables makers of compilers for EOO programming languages to create debugging tools that contain sufficiently detailed information while still being appealing to the user as they minimize duplicate information.

Keywords debugging, modeling, simulation, compilation, Modelica

1. Introduction

1.1 General

Equation-based object-oriented (EOO) programming languages have significant advantages in describing large models since it is easy to construct a hierarchy of models and connecting them.

However, in order to simulate such models efficiently, EOO simulation tools perform a lot 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 EOO simulation tools 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.

While some tools allow you to export a description of the translated system of equations [17], this is not enough. After symbolic manipulation, the resulting equations no longer need to contain the same variables or structure as the original equations. As a simple example, the resulting equation $r = t$ in (1) holds given that r does not change value during events and $t_{start} = r_{start} = 0$.

$$der(r) = 1.0 \Leftrightarrow r = 1.0 * (t - t_{start}) + r_{start} \Leftrightarrow r = t \quad (1)$$

There are two main aspects of debugging application models. One is debugging the simulation executable itself [18]. In general, simulation tools can emit the value of each variable at each time step so that it is possible to see if the results are correct. However, it is usually not possible to understand the cause(s) of slow simulations by studying such data.

By using profiling techniques [12], it is possible to detect which equations cause slowness in a simulation. By using simple techniques to sample and log high-precision clocks at each time step, you can see how much time is spent calculating each strongly connected component. This essentially means that the profiling returns a set equations that calculate a set of variables. But these are variables in the optimized equation system which makes it is hard to know their origin [6].

Thus, you need to look earlier in the compilation process where the equation system is symbolically optimized. Moreover, to understand the causes of possible erroneous variable values, you also need to be able to look into the chain of symbolic transformations of the original model. This can be regarded as debugging of the model compiler

4th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools. September, 2011, ETH Zürich, Switzerland.
 Proceedings available at:
http://www.ep.liu.se/ecp/056/
EOOLT 2011 website:
http://www.eoolt.org/2011/

regarding the efficiency of the generated code with your application model as input.

1.2 Comparison with Traditional Debugging

When most people hear the word debugging, they think of a statement or instruction-level debugger that uses breakpoints, like GDB [19]. A debugger is a computer program designed to help a programmer in the task of finding faults in programs. But debugging is not limited to just instruction-level debugging.

But there is a fundamental difference between our needs and that of a debugger for a general-purpose programming language. We need to be able to debug the symbolic transformations/optimizations performed on the equation system defined in our application model.

While you could use a statement-level debugger to find out what is happening in the compiler, it is hard to use for a regular user due to requiring compiler sources and a debug version of the compiler itself. You also need some knowledge about the data structures used since representing data in certain ways improves the performance of many algorithms used in these tools. This generally limits the use of such debuggers to the compiler developers themselves.

However, even if you are a compiler developer, it is problematic to use a regular debugger simply due to the size of the equation systems that you are required to debug as it becomes harder and harder to setup the debugger to only look at a single variable or equation. Thus, these traditional instruction-level debuggers are unsuitable also from a compiler developer's point of view.

Since equation-based modeling languages are quite different from general-purpose programming languages like C, it is understandable that debuggers made for conventional imperative languages do not work as well in our domain. Thus, we need a domain-specific debugger that can show the user the flow of transformations and calculations. This is analogous to how a tool such as ANTLRWorks [4] can show which rules were used to parse some input given a certain ANTLR grammar, where the language to write grammars is a domain-specific language.

We propose an approach to debugging the symbolic optimization and transformation of equation systems that is usable by tool developers and modelers alike. The approach can be used for debugging during translation (compilation), statically after translation and dynamically during simulation runtime. The focus of this article is on the method, producing the information required to later on write an efficient graphical debugger.

1.3 Common Operations on Continuous Equation Systems

In order to create a debugger adapted for debugging the symbolic transformations performed on equation systems, we need to list its requirements. There are many symbolic operations that we may perform on equation systems. The description of operations described below also include a rationale for each operation since it is not always apparent why we perform such operations. There are of course many more operations we may perform than the ones listed below. The operations shown below are the ones we are most concerned with, and the ones that our debugger for models translated by the OpenModelica Compiler [10] should be able to handle.

1.3.1 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).

$$connect(a, b) \tag{2}$$

$$a.v = b.v \tag{3}$$

$$a.i + b.i = 0 \Leftrightarrow b.i = -a.i \tag{4}$$

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 $-a.i$, which may lead to further optimizations of the equations.

1.3.2 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 for each time step; once is enough for known variables, parameters and constants.

$$a = 4.0 \tag{5}$$

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

$$b = 4.0 - 4.0 + c \tag{7}$$

1.3.3 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 straight-forward, and it is not always possible to invert user-defined functions such as (10). Since x is present in the call arguments and we cannot invert or inline the function, we fail to solve the equation symbolically and instead solve it numerically using a non-linear solver during runtime.

$$15.0 = 3.0 * (x + y) \tag{8}$$

$$x = 15.0/3.0 - y \tag{9}$$

$$0 = f(3 * x) \tag{10}$$

1.3.4 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. We may change the order in which arguments are evaluated (11). Constant subexpressions are evaluated during compile-time (12). We may also rewrite non-constant subexpressions (13) and choose to evaluate functions fewer times than in the original expression (14). We may also use special knowledge about an expression in order to make it run faster (15) and (16).

$$and(a, false, b) \Rightarrow and(false, a, b) \Rightarrow false \quad (11)$$

$$4.0 - 4.0 + c \Rightarrow c \quad (12)$$

$$max(a, b, 7.5, a, 15.0) \Rightarrow max(a, b, 15.0) \quad (13)$$

$$f(x) + f(x) + f(x) \Rightarrow 3 * f(x) \quad (14)$$

$$if\ cond\ then\ a\ else\ a \Rightarrow a \quad (15)$$

$$if\ not\ cond\ then\ false\ else\ true \Rightarrow cond \quad (16)$$

1.3.5 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.

$$\begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 4 \\ 5 \end{pmatrix} \quad (17)$$

$$\begin{pmatrix} 1 & 2 \\ 0 & -3 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 4 \\ -3 \end{pmatrix} \quad (18)$$

$$\begin{aligned} x &= 2 \\ y &= 1 \end{aligned} \quad (19)$$

1.3.6 Differentiation

Symbolic differentiation [7] 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 [5]. The matrix is often computed using automatic differentiation [7] which combines symbolic differentiation with other techniques to achieve fast computation.

$$\frac{\partial}{\partial t} t^2 \Rightarrow 2t \quad (20)$$

1.3.7 Index reduction

In order to solve DAE's numerically, we use discretization techniques and methods to numerically compute derivatives (often referred to as solvers). Certain DAE's need to be differentiated symbolically to enable stable numeric solution. The differential *i*ndex 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 [9]. 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 [2]. One such technique uses dummy derivatives [13]; this is the algorithm currently used in the OpenModelica Compiler.

1.3.8 Function inlining

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

$$2 * f(x, y)/\pi \quad (21)$$

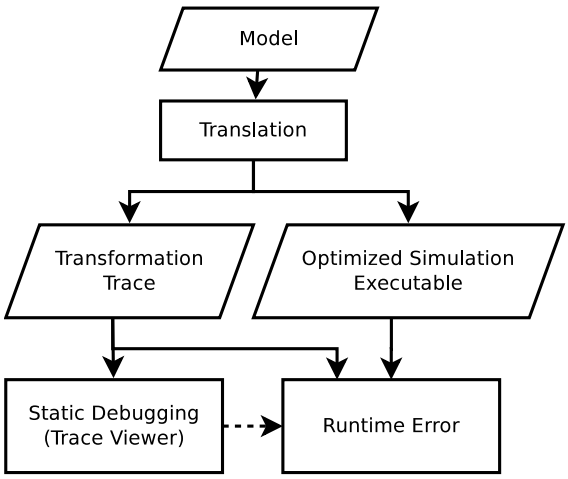
$$2 * \pi * (sin(x + y) + cos(x + y - y))/\pi \quad (22)$$

$$2 * (sin(x + y) + cos(x)) \quad (23)$$

2. Debugging Equation-Based Models

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 so 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 runtime could use the debug information from the translation and optimization stages to give more understandable and informative messages to the user (see Figure 1).

Figure 1. Using Information from the Translation in Subsequent Phases



A technique that is commonly used for debugging is tracing (sometimes called logging depending on context). Some programs that output a trace in order to help in the debugging of C applications are `strace` and `valgrind`.

`strace` (`truss` on some UNIX systems) logs all system calls and signals that the attached process receives. `valgrind` is a suite of tools for debugging (and profiling) programs. Its default tool, `MEMCHECK`, helps C programmers that use pointers in unintended ways. It is capable of reporting unreachable blocks of data, double free, accessing recently free'd data, accessing non-allocated data and calling standard C functions with input that has undefined result. Both `strace` and `valgrind` display the information to the user by streaming text.

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 you roll back an operation, the log-file will still contain the operation that was rolled back. You also need to post-process the data if you require the operations to 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 we perform an operation, we store the operation kind and input/output 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 metadata 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 of 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). If these expressions now referenced by being stored as metadata would normally become garbage and subsequently be deallocated, the memory usage increases, but this is a small amount even for large models[1].

2.1 Bookkeeping of Operations

2.1.1 Variable Substitution

We will consider the elimination of variable aliasing and variables with known values (constants) as the same operation that can be done in a single phase. It can be performed as a fixed-point algorithm where you collect substitutions and 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.

$$a = b, b = -c, c = 4.5 \tag{24}$$

$$a = b \Rightarrow a = -c \Rightarrow a = -4.5 \tag{25}$$

$$a = b \Rightarrow a = -4.5 \tag{26}$$

In equation systems that originate from a Modelica model we prefer 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).

We also plan 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.

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

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.

2.1.2 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).

$$x/y = 1 \Rightarrow x = y \quad (y \neq 0) \tag{28}$$

$$y \neq 0, 46.0 < y < 173.3 \Rightarrow 46.0 < y < 173.3 \tag{29}$$

2.1.3 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 your simplification algorithm oscillates (as in 30) it is hard to use it as a fixed-point algorithm.

$$2 * x \Rightarrow x * 2 \Rightarrow 2 * x \tag{30}$$

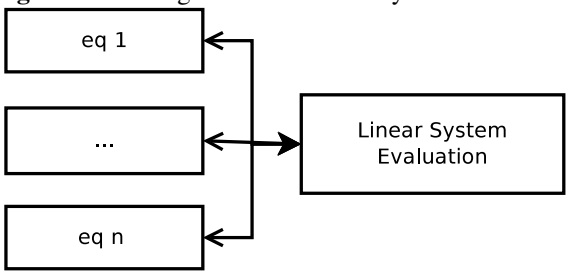
The simple solution is to use an algorithm that is fixed-point, or conservative (reporting no change made when performing changes that may cause oscillating behaviour). Finding where this behaviour occurs is not hard to find for a compiler developer (simply print an error message after 10 iterations). If it is hard to detect if a change has actually occurred (due to changing data representation to use more advanced techniques), you 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.

[1] In the OpenModelica Compiler, the memory overhead for such tracing is roughly 40MB in a model with 30,000 equations. Most of that is shared with other data structures.

2.1.4 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 Result 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, e.g. solving for y in (31), are added to the compiler, they will also need to be traced and support added for them in the debugger.

$$\begin{pmatrix} 1 & 1 & 2 \\ 1 & i & 1 \\ & -i & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 15 \\ 18 \\ 18 \end{pmatrix} \tag{31}$$

2.1.5 Differentiation

Whenever we perform symbolic differentiation in an expression, e.g. to expand known derivatives (32), we record this operation in the equation. We currently do not eliminate this state variable as in (33), but if we did the operation would also be recorded.

$$der(x) = der(time) \Rightarrow der(x) = 1.0 \tag{32}$$

$$der(x) = 1.0 \Rightarrow x = time + (x_{start} - time_{start}) \tag{33}$$

2.1.6 Index reduction

For the index reduction algorithm, we record any substitutions made, add source information to the new variable, and 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 (34) into (35) and (36). After the index reduction is complete, further optimizations such as variable substitution (37), are performed to reduce the complexity of the complete system.

$$x^2 + y^2 = L^2 \tag{34}$$

$$der(x^2 + y^2) \Rightarrow 2 * (der(x) * x + der(y) * y) \tag{35}$$

$$der(L^2) \Rightarrow 0.0 \tag{36}$$

$$2 * (der(x) * x + der(y) * y) \Rightarrow 2 * (u * x + v * y) \tag{37}$$

2.1.7 Function inlining

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

2.2 Presentation of Operations

Until now we have focused on collecting operations as data structured in the equation system. What can we do with this information? During the translation phase we can use it 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. Our OpenModelica prototype only outputs text at present; this is sufficient for a proof of concept as the output is human-readable, but it is not suitable for user tools that require structured data in order to be efficient.

The number of operations stored for each equation varies widely. The reason is that when you replace a known variable x with for example the number 0.0, you may start removing subexpressions. You then end 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 you do not limit the number of iterations that the optimizer may take [8]. 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

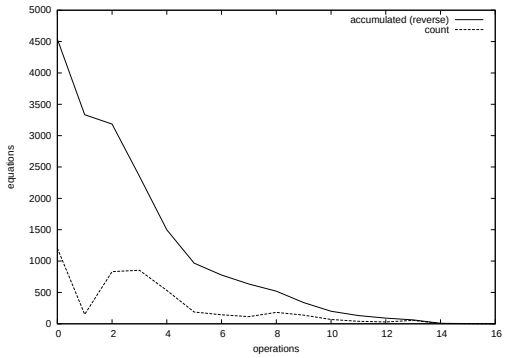
```
model AliasClass_N
  constant Integer N=60;
  Real a[N];
equation
  der(a[1]) = 1.0;
  a[2] = a[1];
  for i in 3:N loop
    a[i] = i*a[i-1]-sum(a[j] for j in
       1:i-1);
  end for;
end AliasClass_N;
```

Using a real-world example, the Engine1a model[2], the majority of equations have less than 10 operations (Figure 3), which is a manageable number to go through when

[2] Modelica.Mechanics.MultiBody.Examples.Loops.Engine1a from MSL 3.1 [15]

you need to debug your model and to know which equations are problematic.

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



2.3 Runtime

In order to produce better error messages during runtime, it would be beneficial to be able to trace the source of the problem. We use the toy example in Listing 2 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). We also display the symbolic transformations performed on the equation, which will help us debug some problems with the model.

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>
```

The equation that threw an assertion also depends on other variables. We display these as a link (which opens up a view with the information in Listing 3), enabling a user to debug the input of the equation. Using this view, we quickly realize that v is a constant, which causes the expression $v * (1.5 - v) - time$ to become 0 at some point in time. If v was time-variant, it is possible that the equation could always be calculated.

Listing 3. Runtime Error Dependency

```
Source equation: [MyModel.mo:X:Y-Z:A] u
    = p-time
Source component: <top> (MyModel)
Flattened equation: u = v*(1.5-v)-time
Manipulated equation: u = 0.5-time
  <Expand operations>
    simplify: 0.5*(1.5-0.5)-time =>
        0.5-time
    variable substitution: v*(1.5-v)-
        time => 0.5*(1.5-0.5)-time
```

If the user interface is intuitive enough (i.e. it is possible to follow links), this should give a user all the information he needs to debug his model. All possible error sources (including but not limited to solvers not converging, domain errors, singular equations, assertions) should be able to display such error messages and open up a browser/debugger of the optimized equation system.

This information can be easily made available in generated code by loading the database containing the transformation traces and annotating the source code with the key to the correct row in the table.

3. Conclusions and Further Work

A prototype design and implementation for tracing symbolic transformations and operations in a compiler (the OpenModelica Compiler [11]) for an EOO language (Modelica [14]) has been constructed. It has an overhead in the order of 0.01%. The rest is simply writing the database to file, which is relatively cheap compared to generation and compilation of the generated C code. The implementation also considers that some operations, e.g. cascading alias elimination, should be considered as an expandable operation that combines a simple overview and a detailed view. Further, a design for constructing a debugger using this data has been provided. This design can be used for static debugging, runtime debugging and to provide understandable runtime errors.

The runtime system of OpenModelica [10] simulations needs to be extended to take advantage of the collected information. It currently uses a mixture of text-files and compiled C sources to set parameters, start values, variable names and source information. The choice of using SQL or XML to store the description of the symbolic operations is a difficult one to make since many other tools rely on being able to set parameters in the OpenModelica text-based init-file (this is scheduled to change once OpenModelica supports the FMI interface [3], which uses XML files for initialization of data). While XML could also be used together with an template written in XSLT to display the information in a regular web browser, the overhead of writing and parsing huge XML files (hundreds of MB). We plan to implement our database using SQL as it is a more practical choice given that the database is usually never

needed (if the simulation succeeds and gives the correct result noone needs the debugger). The savings in storage and computation time should make up for the flexibility XML would give us.

A graphical debugger/browser for the transformations also needs to be designed as the current prototype only relies on searching in text-based files. Without this, a regular user will not be capable of taking advantage of the trace. At the time of this writing, the trace is rather cryptic and only available from the command-line, but an advanced user could still take advantage of the current trace (Listing 4) as it is more readable than the source code (Listings 5 and 6).

Listing 4. OpenModelica Trace (Snipped for Brevity)

```
nonlinear: x,y
  residual: xloc[0] - xloc[1] * time;
    operations (3):
      subst:
        x - y * time
        =>
        xloc[0] - xloc[1] * time
      simplify:
        y * (time * 1.0)
        =>
        y * time
      subst:
        y * (time * z)
        =>
        y * (time * 1.0)
  residual: xloc[1] - sin(xloc[0] *
     time);
    operations (1):
      subst:
        y - sin(x * time)
        =>
        xloc[1] - sin(xloc[0] * time)
```

Listing 5. OpenModelica ODE code fragment

```
start_nonlinear_system(2);
nls_x[0] = extraPolate($Px);
nls_xold[0] = $P$old$Px;
nls_x[1] = extraPolate($Py);
nls_xold[1] = $P$old$Py;
solve_nonlinear_system(residualFunc2);
$Px = nls_x[0];
$Py = nls_x[1];
end_nonlinear_system();
```

Listing 6. OpenModelica residualFunc2

```
res[0] = (xloc[0] - (xloc[1] * time));
tmp1008 = sin((xloc[0] * time));
res[1] = (xloc[1] - tmp1008);
```

This work focused mainly on operations performed on individual equations. By analyzing the BLT matrix of the equation system, a user could additionally be presented with equations that need to be solved before or after the selected equation. With more bookkeeping it should even be possible to show how the relationships change during the optimization process.

We also did not consider operations performed on algorithmic code or optimizations that can be performed on the hybrid parts of the system, e.g. dead code elimination of unreachable discrete events.

References

[1] Ed Anderson et al. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, 1999.

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

[3] Torsten Blochwitz et al. The functional mockup interface for tool independent exchange of simulation models. In Modelica'2011 [16].

[4] Jean Bovet and Terence Parr. ANTLRWorks: an ANTLR grammar development environment. *Software: Practice and Experience*, 38:1305--1332, 2008.

[5] Willi Braun, Lennart Ochel, and Bernhard Bachmann. Symbolically derived Jacobians using automatic differentiation - enhancement of the OpenModelica compiler. In Modelica'2011 [16].

[6] Peter Bunus. *Debugging and Structural Analysis of Declarative Equation-Based Languages*. Licentiate thesis No 964, Linköping University, Department of Computer and Information Science, 2002.

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

[8] Jens Frenkel, Christian Schubert, Günter Kunze, Peter Fritzson, and Adrian Pop. Towards a benchmark suite for modelica compilers: Large models. In Modelica'2011 [16].

[9] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley-IEEE Press, February 2004.

[10] Peter Fritzson, Peter Aronsson, Håkan Lundvall, Kaj Nyström, Adrian Pop, Levon Saldamli, and David Broman. The OpenModelica Modeling, Simulation, and Software Development Environment. *Simulation News Europe*, 44/45, December 2005.

[11] Peter Fritzson et al. Openmodelica 1.7.0, April 2011.

[12] Michaela Huhn, Martin Sjölund, Wuzhu Chen, Christan Schulze, and Peter Fritzson. Tool support for Modelica real-time models. In Modelica'2011 [16].

[13] 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.

[14] Modelica Association. The Modelica Language Specification version 3.2, 2010.

[15] Modelica Association. Modelica Standard Library version 3.1, 2010.

[16] *Proceedings of the 8th International Modelica Conference*. Linköping University Electronic Press, March 2011.

[17] 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.

[18] Adrian Pop and Peter Fritzson. Towards run-time debugging of equation-based object-oriented languages. In *Proceedings of the 48th Scandinavian Conference on Simulation and Modeling (SIMS)*, October 2007.

[19] Richard Stallman, Roland Pesch, Stan Shebs, et al. *Debugging with GDB*. Free Software Foundation, 2011.