Integrated Debugging of Equation-Based Models

Martin Sjölund[1], Francesco Casella[2], Adrian Pop[1], Adeel Asghar[1], Peter Fritzson[1],
Willi Braun[3], Lennart Ochel[3], Bernhard Bachmann[3]
[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
[3] Dept. Mathematics and Engineering, University of Applied Sciences,
D-33609 Bielefeld, Germany
{adrian.pop,martin.sjolund,adeel.asghar,peter.fritzson}@liu.se
francesco.casella@elet.polimi.it, {bernhard.bachmann,lennart.ochel,willi.braun}@fh-bielefeld.de

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 the first integrated debugger for equation-based languages like Modelica, which can combine static and dynamic methods for run-time debugging of equation-based Modelica models during simulations. This builds on and extends previous results from a transformational static equation debugger and a dynamic debugger for the algorithmic subset of Modelica.

Keywords: Modelica, Debugging, Modeling and Simulation, Transformations, Equations, Algorithmic Code, Runtime Errors, Tracing, Solver Failures

1 Introduction

The advanced development of today's complex products requires integrated environments and equation-based object-oriented declarative (EOO) languages such as Modelica [10][14] for modeling and simulation.

The increased ease of use, the high abstraction, and the expressivity of such languages are very attractive properties. However, the downside of this high-level approach is that understanding the root causes of unexpected behavior and numerical errors of simulation model is very difficult, in particular for users who are not experts in simulation methods.

The main reason of this difficulty the fact that lots of sophisticated symbolic and numerical transformations are applied to the original model in order to eventually obtain the executable simulation code, in which errors and problems do occur. An effective debugging environment should then guide the end user back and forth through the numerical results and all the performed symbolic transformations of the model, in order to quickly find and correct the causes of errors. This paper presents the integrated debugger of the OpenModelica tool suite, including a graphical user interface integrated with the OpenModelica Connection Editor (OMEdit) GUI. This builds on and extends previous results from a transformational static equation debugger [6][7] and a dynamic debugger [1][3][4] for the algorithmic subset of Modelica.

Despite the fact that debugging environments have been the subject of extensive research and implementation work in the field of computer science, to the best of the authors' knowledge this is the first documented operational debugging environment for equation-based modeling languages supporting dynamic debugging of equation-based mathematical models as well as algorithmic code in an integrated way.

The rest of the paper is structured as follows: The debugging procedure is outlined in Section 2 and the GUI in Section 3. The tracing of equation transformation is discussed in Section 4, while Section 5 discusses the issues of interfacing with the run-time simulation executable. In Section 6, some example models are shown, illustrating how the debugger can help their troubleshooting. Section 7 discusses background and related work, Section 8 states the current implementation status at the time of this writing, and Section 9 concludes the paper.

2 Overall Debugging Procedure

The debugger should support three basic scenarios:

- The simulation stops at a certain time step, or during initialization, because of a numerical runtime error;

- A complete simulation run has been performed successfully, but some variables exhibit suspicious or clearly wrong values;
- A breakpoint is inserted to stop the integration either at a certain given value of the time variable, or when some user-supplied condition is triggered. In this case, it should be possible to restart the simulation (and possibly to set a new breakpoint)

The different functionalities of the debugger are specified in more detail in the following sub-sections.

2.1 Types of Debugging Activities

We divide the problem of debugging the execution (i.e., the numerical simulation) of an equation-based model into three different areas:

- *Initialization.* Before starting the simulation, consistent initial conditions are computed by solving a set of initial equations. In the following, it is assumed that this is done by using multiple optimization strategies, such as alias variable elimination, BLT partitioning, tearing, etc.
- *Causalization*. It is also assumed that the solution of the differential-algebraic equations over time is obtained by a two-stage strategy. In the casualization stage, the DAEs are solved for the derivatives by using multiple optimization strategies, such as symbolic index reduction as well as the ones previously mentioned.
- *Time integration*. The computed derivatives (and possibly their Jacobian matrix) are then passed to ODE solvers, such as DASSL, Runge-Kutta, Radau, etc., that advance the solution of the system over time

2.2 Debugging Initialization and Causalization Problems

For the purpose of debugging, initialization and causalization share a common structure despite using different numerical solvers. They can be represented using a similar GUI. The only difference is that the set of equations and unknowns for initialization is larger than for causalization, as it also includes the state variables and the parameters, as well as the initial equations and parameter-binding equations. Also, the simulation code to solve both problems is usually generated by the Modelica tool itself, so it is fairly straightforward for the tool developers to add all kind of instrumentation to it for debugging purposes.

Variables are matched to the equations that are used to solve them. If an error has occurred while trying to compute a certain variable or a certain set of variables for strong components in the BLT, the error (e.g., division by zero, logarithm of a negative number, singular linear system of equations, etc.) is reported in the context of the equation as it has been transformed in order to solve it efficiently at run time. Then, it is possible to backtrack step-by-step each stage of the transformations of each equation, up to the original equations in the source code.

This activity can also be carried out in the absence of errors, either when a breakpoint is triggered, or when the values at a specific time step are inspected after the simulation run has been performed. Assuming that some variable(s) have suspicious, or maybe clearly wrong values, one starts analyzing the equations that were used to compute them, going backwards in the causality chain determined in the BLT, and trying to locate the model error that caused the computation of the wrong values.

The solution of the equation(s) also depends on the values taken by all the other known variables showing up in the equations, either states or other unknown variables previously computed in the BLT. The debugger allows to inspect the values taken by these variables, as well as the equation(s) in which they were solved for. Then, the same activities will be possible recursively on this new set of equations: understanding where they come from in the equation transformation chain, as well as inspecting the values of the variable(s) they depend upon.

2.3 Debugging Time Integration Problems

The requirements for the debugging of *time integration* problems are quite different. Unrecoverable errors generated by the ODE solver should be reported to the debugger using some kind of unified representation (e.g., using XML), which is as independent as possible from the specific solver used. Of course, some errors will only make sense for a subset of solvers; for example, singular Jacobians are only relevant in the case of implicit solvers; event chattering is only relevant for solver with state event detection.

The first kind of error that can arise in solvers with state event detection based on zero crossing function is chattering: if a large number of events takes place in a very short time interval, then the debugger reports the corresponding zero-crossing functions and allows to back-track them to their original formulation in the source code, as well to inspecting the values of all the variables involved in them in the last accepted time steps.

It may also be the case that chattering arises without any event being generated, if the `noEvent()` operator is incorrectly placed around a discontinuous expression

DOI 10.3384/ECP14096195

inside a model equation, or if some functions in the model generate results which are discontinuous w.r.t. their inputs (recall that Modelica functions do not generate events). This situation can be detected by monitoring the step size, and detecting the fact that the step size has been reduced to very small values for a very large number of step sizes.

In order to identify the root cause of the problem, it is necessary that the ODE solver can report which component(s) of the state vectors have the largest estimated errors, and are thus mainly responsible for the excessive step size reduction. The debugger will then point the end user to the equations that are used to compute the corresponding derivatives, using the same mechanism adopted for the initialization and casualization steps. Wildly oscillating values of the derivatives will be observed across the last time steps, and it will then be possible to analyze the expressions leading to these oscillations, eventually locating the root cause of the problem.

Another possible error can arise at the ODE solver level if the underlying differential equations have a finite escape time, i.e., one or more elements of the state vector go to infinity as time approaches a certain finite value. The main symptom in this case is very similar to the previous case, i.e., the step size is greatly reduced and the simulation seems stuck at a certain point in time.

The root cause can also be identified in this case if the solver reports the component(s) of the state record that mostly contribute to the error estimate, so that the debugger can allow the user to inspect the equation(s) that compute the corresponding derivatives. The values of these derivatives will constantly grow from one step to the next one, rather than oscillating wildly as in the previous case. Again, by careful inspection and analysis, it might be possible to understand the root cause of the problem and fix it.

2.4 Debugging Homotopy-based Initialization Problems

If the `homotopy()` operator is used for initialization, two extra stages are added to the debugging of the initialization problem. First, the set of initial equations using the simplified expression is presented. The BLT structure of this problem might be substantially different (and hopefully simpler) than that of the actual initialization problem, but the way it is presented in the GUI to the user for analysis is the same as for the actual initialization problem.

The second stage is the homotopy transformation. From a GUI perspective, this is very similar to the simulation phase as there are several steps involved. Each might be accepted, rejected, or eventually fail if the errors cannot be recovered by taking shorter steps. Also, similarly to the simulation phase, errors might be reported that arise while solving the equations in the BLT sequence (as in the initialization and causalization problems), but also some system-level errors might be reported by the homotopy solver itself, e.g., in case of homotopy path bifurcations, similarly to problems reported by the ODE solver during time integration.

The GUI is therefore similar to the one used for debugging errors during simulation, with the following differences:

- The set of unknowns includes states and parameters;
- the set of equations include initial equations and parameter-binding equations
- All occurrences of the homotopy operator [14] in the equations are transformed into λ**actual_expr* + (1 – λ)**simplified_expr*;
- The independent variable which is stepped is not time but rather the λ homotopy parameter.

3 Debugger Graphical User Interface

In order to visualize the transformations performed and the operations taken by the solver to solve for a variable and its corresponding equation(s), a *transformations browser* (Figure 1; Figure 2; Figure 3) has been created.

The transformations browser lists the variables along with their respective types hierarchy, operations performed, equations which defines the variable and equations which are using the variable. The types can be used to navigate to the specific class.

Double clicking on the equation updates the transformation browser and shows the list of operations and variables involved in the solution of the equation. See Figure 3.

The transformation browser provides two views:

- Variables view
- Equation View

The data needed to build the structures shown in the GUI, i.e., the structural information about the equation systems, and the equation transformation traces, are loaded from an XML file which is generated by the OpenModelica compiler, see Section 4 for more details.

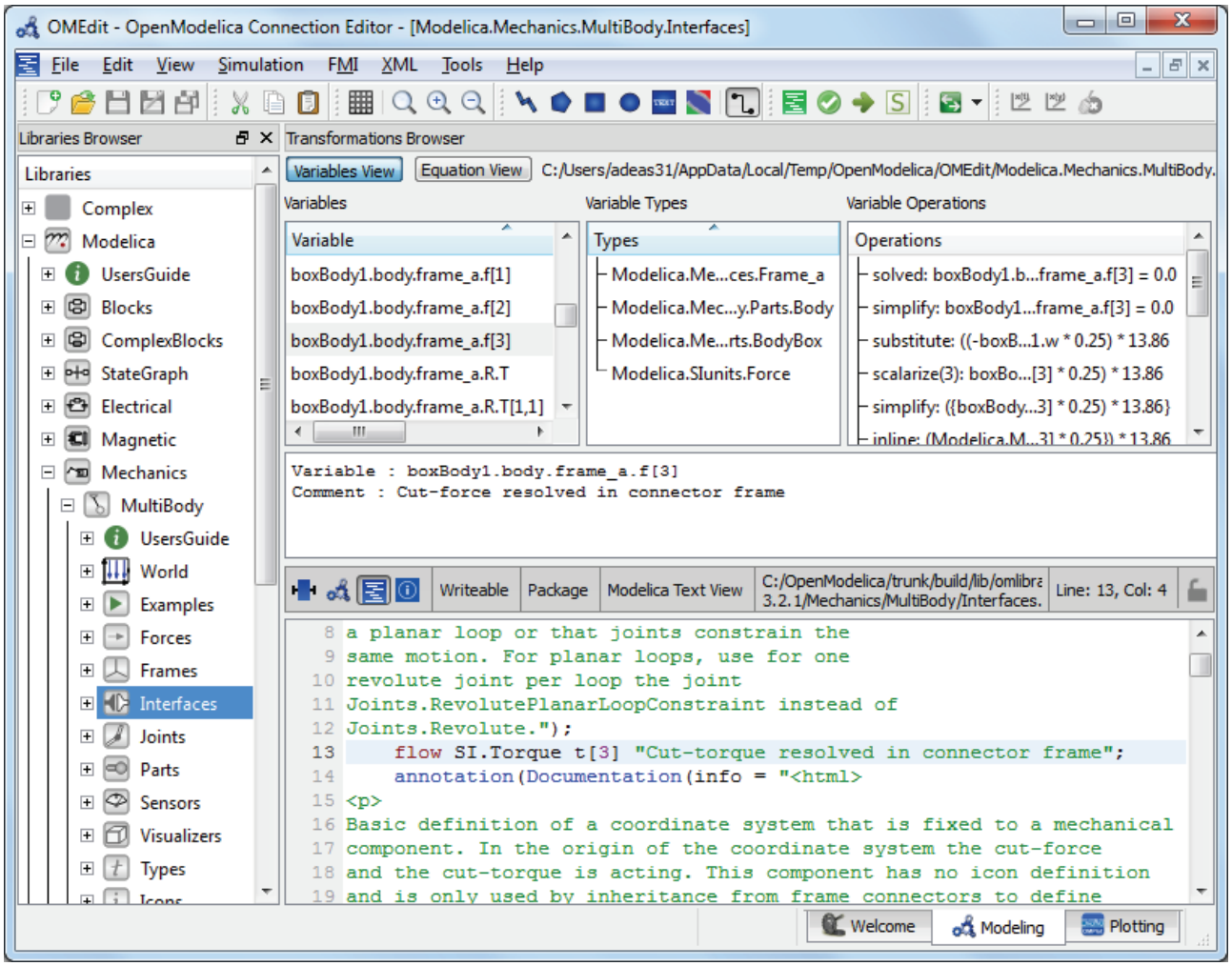

Figure 1. Transformations browser variables view with columns: `Variables`, `Variable Types`, `Variable Operators`.

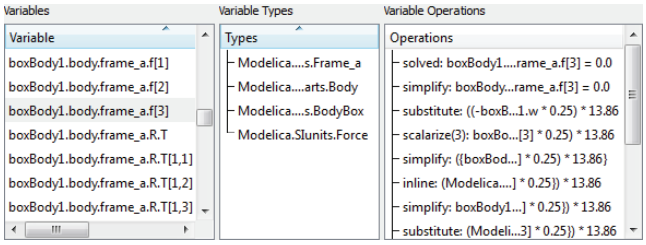

Figure 2. Enlarged left part of variable info in transformations browser *variable view* with columns: `Variables`, `Variable Types`, `Variable Operators`.

When a numerical error is reported, clicking on the "Debug more" link at the end of the error report brings up the debugger showing the equation(s) involved in the error.

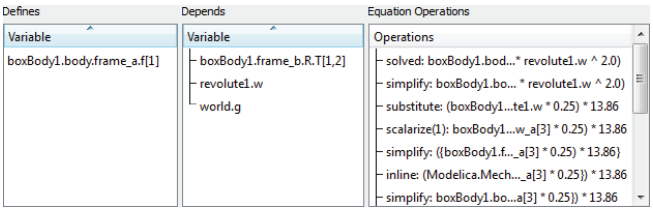

Figure 3. Enlarged part of transformation browser *equation view* with `Defines` variable, `Depends` on variable, `Equation Operators` operations like solved, simplify, substitute, etc.

4 Transformation Tracing

The underlying implementation of the transformation tracing mechanism is described in more detail in [7]. Some further improvements are present in this version.

The key idea introduced in [7] is to encode and store in a list all transformations that are performed by the Modelica compiler on the model equations, such as symbolic solution, alias elimination, symbolic differen-

DOI 10.3384/ECP14096195

tiation, etc. Because every operation is stored, it is possible to replay the operations and verify that the tool only performed sane operations during translation. This list of operations is then output to an XML-file (Figure 4) which is parsed by the debugger.

```
<simplify>
 <before>
  Nand.TP1.G.i + Nand.TN1.G.i + (-Nand.x2.i)
  = 0.0
 </before>
 <after>
  Nand.TP1.G.i + Nand.TN1.G.i - Nand.x2.i
  = 0.0
 </after>
</simplify>
<substitution>
 <before>
  Nand.TP1.G.i + Nand.TN1.G.i - Nand.x2.i
 </before>
 <!-- list of intermediate results -->
 <exp>0.0 + 0.0 - (-VIN2.i)</exp>
</substitution>
<simplify>
 <before>0.0 + 0.0 - (-VIN2.i) = 0.0</before>
 <after>VIN2.i = 0.0</after>
</simplify>
<solved>
 <lhs>VIN2.i</lhs>
 <rhs>0.0</rhs>
</solved>
```

Figure 4. List of equation transformations in the model `Modelica.Electrical.Analog.Examples.NandGate`.

The XML-file contains all the variables and equations used to solve the model, as well as variables that have already been solved for, alias relations, and so on. The equations are split into several groups, such as start-value equations, initial equations, regular equations, since the same variable may have different equations defined for it in different phases of the program.

These groups are related to how the compiler decided to numerically solve simulations. For example, the file includes the equations generated for the Jacobian, which is not used by all numerical ODE/DAE solvers.

Each equation knows the variables it solves for, as well as the variables it uses. This enables fast lookup of parents, children, and siblings in the BLT matrix. When reading the file, information is propagated to variables in a way such that each variable also knows the equation(s) where it is defined. This is again to ensure that the debugger can perform cheap lookup operations.

In the case of strongly connected components, an equation index will point to a set of equations (linear and nonlinear systems of equations in OpenModelica are defined as a set of equations and variables to solve for). The generated code knows the index of an equation in the XML-file, so in case error or diagnostic messages are generated, a link to the equations and variables associated with this index can be provided to the debugger.

The message routines have been updated to take a list of equation indexes as an option, as well as output the messages as structured XML. This enables the debugger to read the messages and insert links to equations as appropriate.

This approach allows a user to debug simulations even if he/she did not run the simulation through the debugger, because it is possible to perform post-mortem debugging only based on the messages and diagnostics produced by the simulation executable.

There is no additional overhead during regular execution except reading and writing the additional information in the XML-file. This can be done by a thread running in the background and takes only a few seconds even for the large `EngineV6` model which both has many equations and many symbolic operations performed on each equation.

For error-messages there is an additional overhead of creating an error message that contains all the relevant information. This is a small one-time cost for error, which are hopefully infrequent. Consequently, the detailed error messages are output even if the user had not decided to debug the simulation before he started it since it will help him figure out why things went wrong.

5 Run-Time and Event Related Implementation

The run-time system performs the actual simulation of a Modelica model, in which the solution process is done by different solvers that cooperate in a master-slave hierarchical configuration, with the ultimate master being the end-user:

- ODE solver
- Functions computing the derivatives and algebraic variables
- Function computing the initial states and the values of parameters
- Function computing event points
- Linear equation solvers
- Nonlinear equation solvers

All of them may fail with different kinds of errors depending on the solver, generally because of numerical issues (e.g. singular Jacobian, no convergence, too tight

tolerance). However, at the bottom level they all share particular error types:

- Evaluation of expressions
  - Division by zero.
  - Functions called outside their domain (e.g.: `sqrt`(-1), `log`(-3), `asin`(2)).
  - Evaluation of non-integer powers with negative argument
- Assertion violations for the model

In general some errors can be recovered automatically by the system (e.g. by re-trying with a shorter time step), whereas others abort the simulation and are reported to the user, which can then enter the debugging mode.

If an error cannot be recovered by the solver hierarchy, informative diagnostics are provided to the user. The diagnostic error message includes the corresponding equation block, the involved variables and their values. Furthermore the hierarchical context of the error is important to be able to classify it.

In the next step the user may be able to enter the debugging mode, where the simulation can be re-run to an accepted step just before the error occurs again. The last accepted step corresponds to the last point in time in the result file created in the first run. This point in time can be a breakpoint for debugging mode.

In the debug mode breakpoints are interpreted like zero-crossings, but without the time-consuming search process which the numerical solver does — the simulation just breaks if the condition becomes true.

Then the step that caused the failure is executed in a verbose mode, where informative diagnostic is provided for every equation that needs to be solved till the error occurs again. This allows the user to trace the solution process and if necessary, to engage by changing the model.

6 Example Models for Debugging

In this section some simple test cases are shown which demonstrate various possible error scenarios, and how a debugger can help their troubleshooting.

6.1 Chattering Models

In the model `ChatteringEvents1`, chattering takes place after t = 0.5, due to the discontinuity in the right hand side of the first equation. Chattering can be detected because lots of tightly spaced events are generated. The debugger allows to identify the equation from which the zero crossing function that generates the events originates.

```
model ChatteringEvents1
  Real x(start=1, fixed=true);
  Real y;
  Real z;
equation
  z = if x > 0 then -1 else 1;
  y = 2*z;
  der(x) = y;
end ChatteringEvents1;
```

Also in the model `ChatteringNoEvents1`, chattering takes place after t = 0.5, due to the discontinuity in the right hand side of the first equation. However, events are not generated in this case, because of the noEvent operator. If a variable-step-size integration algorithm with error control is used, the time step will be reduced to very small values once the discontinuity is hit, and this can be detected by monitoring the value of time at each time step.

The variable step size solver should be able to report which state variable(s) give the biggest contribution to the error estimate, thus causing the step size reduction. The corresponding derivative shows very high frequency oscillations between two values. The end user can then use the BLT navigation functionality of the debugger to investigate which variable/equation is introducing the discontinuity.

```
model ChatteringNoEvents1
  Real x(start=1, fixed=true);
  Real y;
  Real z;
equation
  z = noEvent(if x > 0 then -1 else 1);
  y = 2*z;
  der(x) = y;
end ChatteringNoEvents1;
```

Regarding `ChatteringFunction1`, after t = 0.5, chattering takes place due to the discontinuity in the right hand side of the first equation. The discontinuity is caused by a discontinuous function, which does not generate events.

The considerations regarding variable-step solvers, derivatives, and debugger BLT navigation are the same as for the previous example `ChatteringNoEvents1`.

```
model ChatteringFunction1
  Real x(start=1, fixed=true);
  Real y;
  Real z;

function f_sign
  input Real x;
  output Real y;
algorithm
  if x > 0 then
    y := 1;
  elseif x < 0 then
    y := -1;
  else
    y := 0;
  end if;
```


DOI 10.3384/ECP14096195

```
end f_sign;

equation
  z = Functions.f_sign(x);
  y = 2*z;
  der(x) = y;
end ChatteringFunction1;
```

6.2 Models with Different Numerical Failure Modes

The `NonlinearSolverFailureInitial`, model describes a simple hydraulic system with a pump, followed by a valve, which fills a reservoir.

The initial value of the level of the reservoir is too high for the pump sizing, so the pressure `p2` is too high and consequently the nonlinear algebraic system of equations that determines `p1` and `w_pump` has no solution.

It is possible to find a solution to the system either by lowering the initial value of `y`, and thus the pressure `p2`, or by increasing the value of the parameter `dp0`, increasing the head the pump can provide.

The debugger can show the dependency of the nonlinear system of equations on the parameters `dp0`, `a1`, `a2`, `a3`, and `Kv` (also showing their values), as well as the dependency on `p2` (which has a too high value). Once one understands that `p2` is too high, it should be possible to continue the analysis, looking at the equation that determines `p2`, which in turn depends on the value of the state `y`, which is the root cause of the problem.

The nonlinear system that cannot be solved has five unknowns: `w_pump`, `dp_pump`, `dp_valve`, `sqrt_dp`, and `p1`, which can be easily reduced to one by using `dp_pump` as a tearing variable. The debugger can show the torn variables and the tearing variables, as well as the corresponding torn equations and implicit residual equations, and allows to track the values of all five variables during the iterations of the Newton algorithm.

```
model NonlinearSolverFailureInitial
  parameter SI.Pressure patm=101325
    "Atmospheric pressure";
  parameter Real Kv=1e-2 "Valve coefficient";
  parameter Real dp_small=1
    "Small dp for valve equation";
  parameter Real dp0=3e5 "Pump dp @ zero flow";
  parameter Real a1=1e6 "Pump coefficient";
  parameter Real a2=3e2 "Pump coefficient";
  parameter Real a3=3e2 "Pump coefficient";
  parameter SI.Temperature T0=20 + 273.15
    "Temperature of incoming fluid";
  parameter SI.Density rho=995
    "Density of fluid";
  parameter SI.Area A=0.01
    "Storage tank cross section";
  parameter SI.MassFlowRate w_extra=0
    "Extra mass flow rate into reservoir";
  constant SI.Acceleration g= 9.81
    "Acceleration of gravity";
  parameter SI.Temperature Tref=273.16
    "Reference temperature for specific
       enthalpy computation";
  parameter SI.SpecificHeatCapacity cp=4186
    "Cp of the fluid";
  SI.MassFlowRate w_pump
    "Mass flow rate from the pump";
  SI.Pressure p1 "Pump discharge pressure";
  SI.Pressure p2 "Storage tank inlet pressure";
  SI.Pressure dp_pump "Pump dp";
  SI.Pressure dp_valve "Valve dp";
  Real sqrt_dp "Regularized sqrt(dp)";
  SI.SpecificEnthalpy h0
    "Pump inlet specific enthalpy";
  SI.SpecificEnthalpy h1
    "Pump discharge specific enthalpy";
  SI.Power W "Pump power consumption";
  SI.Length y(start=40, fixed=true)
    "Reservoir level";
  Real eta(final unit="1") =
    (p1 - patm)*w_pump/rho/W "Pump efficiency";
  SI.Temperature T1
    "Pump discharge temperature";
  SI.Time tau=1
    "Time constant of temperature sensor";
equation
  dp_pump = p1 - patm "Pump dp";
  dp_valve = p1 - p2 "Valve dp";
  dp_pump = dp0 - a1*w_pump^2;
  w_pump = Kv*sqrt_dp;
  sqrt_dp = dp_valve/
            (dp_valve^2 + dp_small^2)^0.25;
  W = a2 + a3*w_pump;
  w_pump*(h1 - h0) = W;
  rho*A*der(y) = w_pump + w_extra;
  p2 = rho*g*y + patm;
  h0 = cp*(T0 - Tref)";
  h1 = cp*(T1 - Tref)";
end NonlinearSolverFailureInitial;
```

A simple modification of the previous model allows demonstration of the failure of the nonlinear solver in the causalization stage during simulation. The initial value of the level is reduced to 20, so that an initial solution can be found.

```
model NonlinearSolverSimulation
  extends NonlinearSolverFailureInitial(
            y(start=20), w_extra=0.2);
end NonlinearSolverSimulation;
```

In this case the reservoir is filled both by the pump and by an extra source. The mass flow rate of the pump `w_pump` is determined by a nonlinear system with five unknowns: `w_pump`, `dp_pump`, `dp_valve`, `sqrt_dp`, and `p1`, which basically computes the operating point of the pump as the intersection between the pump head curve and the load (valve + reservoir head) curve. Note that these curves have two intersections (also see `NonlinearSolverFailure3` later on). As the level increases, `w_pump` is reduced, and the two intersections get closer to each other, until at time t = 269 they collide, making the system singular. As the level increases further due to the extra source, this system ceases to have any solution. This is a typical bifurcation pattern in nonlinear systems.

The debugger can show that the condition number of the Jacobian of the nonlinear system gets bigger and bigger as the critical time when the two operating

curves become tangent to each other, suggesting that this system becomes singular for some reason. Understanding the reason why this happens requires physical insight into the model.

The model can be fixed by adding some mass storage depending on the pressure `p1`, in order to avoid the singularity in determining `p1`, and also by using a more realistic cubic curve for the pump model, so that when the limit level is reached, the solution will jump to a big negative pump flow. Again, this requires physical insight into the validity range of the implemented model.

Another slight variation of the model allows demonstrating the case of finite escape time.

```
model FiniteEscapeTime
  extends NonlinearSolverFailureInitial(
            y(start=20));
  SI.Temperature Ts(start=T0);
equation
  tau*der(Ts) = T1 - Ts;
initial equation
  der(Ts) = 0;
end FiniteEscapeTime;
```

As the reservoir level increase, the flow rate `w_pump` goes to zero. When it does, the energy balance equation causes the specific enthalpy `h1`, and thus the temperature `T1`, to go to infinity.

The temperature `T1` is the input of a first-order linear system, representing the temperature sensor dynamics. If a variable step-size solver with error control is used, it will try to compute the state trajectory, which also goes to infinity, so the solver eventually gets stuck at time t = 664.

If the ODE solver reports information on the state whose error estimate is causing the step size to be reduced, (`Ts`, in this case), then the debugger can point the end user to its derivative `der(Ts)`. It will be shown that it depends on `T1`, whose values can be seen to grow indefinitely over time. `T1` is shown to depend on `h1`, which also goes to infinity. Finally, `h1` depends on the energy balance equation, which depends on `w_pump`. At that point it will become apparent that as the flow rate `w_pump` goes to zero, the model becomes ill-posed. The solution in this case is to change the pump model, by adding to the energy balance some dynamic energy storage and/or some heat transfer to the ambient, in order to avoid the zero-flow singularity.

Finally, another small change to the original model presented in this section allows to demonstrate the debugging of models where the wrong initial solution is picked by the nonlinear solver.

```
model WrongInitialSolutionSelected
  extends NonlinearSolverFailureInitial(
            y(start=20),
            dp_pump(start=-1000));
end WrongInitialSolutionSelected;
```

The operating point of the pump is determined by a nonlinear system with five unknowns: `w_pump`, `dp_pump`, `dp_valve`, `sqrt_dp`, and `p1`. It is assumed here that `dp_pump` is selected as a tearing variable. At time t=0, this system has two solutions, one with positive `w_pump`, and the other one with negative `w_pump`. If the start value of the tearing variable `dp_pump` is chosen incorrectly, the solver will converge to the negative solution, then lock onto it for the rest of the simulation.

When the user sees the negative `w_pump` in the simulation (which is physically wrong), he/she should be able to analyze how this value was found at time t = 0. The debugger shows that `w_pump` is solved by that nonlinear system, and shows the values of the tearing variables and of the torn variables at each iteration step.

It will then become apparent that the start value of the tearing variable `dp_pump` leads to a negative value of the torn variable `w_pump`, leading to the solution of the problem, i.e., changing the start value of `dp_pump` to a value that allows convergence on the desired solution.

7 Background and Related Work

Modelica is a declarative language that makes writing equations easy while still producing efficient code. However, traditional debugging tools like GDB [12], Valgrind [19], or any of the other tools described in [18] assumes that the program being debugged is statement based. It also assumes that the user knows something about what the program is doing. This is fine if you are a Modelica compiler developer working on fixing some segmentation fault in your own code. A GDB-based approach exists for Modelica [4]; it works fine for debugging algorithms in functions.

But as a Modelica user you know very little about the internals of the run-time system. For example, there is speculative execution while simulating a model making debugging with GDB confusing.

There exists previous work on debugging in Modelica. Bunus [9] proposes a semi-automated dynamic (run-time) debugging of models where 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.

Sjölund [7] is used as the main basis of the equation debugging part of this work. It was mainly focused on tracing operations in the compiler backend. It has been

DOI 10.3384/ECP14096195

extended with structured error messages from the simulation run-time system as well as an actual debugger.

Pop et al [3], [4] describe an integrated debugging approach based on a dependency graph. Edges in that dependency graph can be computed by the transformational tracing mechanism mentioned in Section 4.

8 Current Status

At the time of this writing, the implementation of the debugger framework in the OpenModelica environment is mostly complete but still missing some parts.

This debugger framework has three main parts: the tracing of symbolic operations in the backend of OpenModelica, reporting run-time errors in simulations, and the debugger implemented as an extension of the OMEdit graphical user interface.

The tracing of operations is complete, and the mapping of error positions in the low level generated code to the high-level model from where they originated. However, the reporting of run-time errors only works for a subset of problems at the moment.

The generation of the XML file with the transformation tracing, and its subsequent representation in the OMEdit GUI are fully implemented. Some types of numerical errors (e.g., chattering) can already be debugged as described in the paper.

However, the interface to the numerical solvers (both for the casualization and for the time integration steps) is still incomplete. Also the functionality of analyzing the results of simulation runs (which did not generate errors) at specific points in time is not implemented yet.

It is planned to have the implementation with the abovementioned additional functionality completed by fall 2014.

9 Conclusions and Future Work

We have presented a set of problems of simulating Modelica models that benefits from increased debugging tool support. We have also presented a design and implementation of the first (to our knowledge) documented debugging framework that can handle this set of problems.

The debugger is operational and has been tested on rather large models without noticeable run-time overhead. It is able to map error positions from low-level compiled simulation code to the corresponding source level equations in the Modelica model.

We believe that this kind of debugging support will significantly improve the ease-of-use regarding application modeling with Modelica compared to the current situation typically needing a large amount of trial-and-error and a lot of expertise in the internal mechanisms of Modelica model compilers and simulation run-time systems. This can speed up the acceptance and use of Modelica in the engineering community.

Future work includes creating additional specialized debugging views including a view to display non-convergence of non-linear equation systems.

10 Acknowledgements

This work has been supported by the Swedish Strategic Research Foundation in the EDOp projects and Vinnova in the RTSIM and ITEA2 MODRIO projects, and by VR. 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 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] Adrian Pop, Martin Sjölund, Adeel Asghar, Peter Fritzson, Francesco Casella. Static and Dynamic Debugging of Modelica Models. In *Proceedings of the 9th International Modelica Conference* (Modelica'2012), Munich, Germany, Sept.3-5, 2012.

[5] 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.

[6] 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.

[7] Martin Sjölund. *Tools for Understanding, Debugging, and Simulation Performance Improvement of Equation-based Models.* ISBN 978-91-7519-624-4, Linköping Studies in Science and Technology. Li-

centiate Thesis No. 1592, ISSN 0280-7971, http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-90096, May 27, 2013.

[8] Peter Bunus and Peter Fritzson. 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. 2003.

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

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

[11] 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.

[12] 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].

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

[14] Modelica Association. *The Modelica Language Specification Version 3.2 revision 2*, July 30th 2013. http://www.modelica.org. Modelica Association. *Modelica Standard Library 3.2.1*. Aug. 2013. http://www.modelica.org.

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

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

[17] 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.

[18] Andreas Zeller. Why Programs Fail, Second Edition: A Guide to Systematic Debugging. ISBN: 978-0123745156, 2009

[19] Nicholas Nethercote and Julian Seward. Valgrind: a Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*. PLDI '07. San Diego, California, USA, 2007, pp. 89-100. doi: 10.1145/1250734.1250746