

Towards Run-time Debugging of Equation-based Object-oriented Languages

Adrian Pop and Peter Fritzson
Programming Environments Laboratory
Department of Computer and Information Science
Linköping University
{adrpo, petfr}@ida.liu.se

Abstract

The development of today's complex products requires advanced integrated environments and modeling languages for modeling and simulation. Equation-based object-oriented declarative (EOO) languages are emerging as the key approach to physical system modeling and simulation. The increased ease of use, the high abstraction and the expressivity of EOO languages are very attractive properties. However, these attractive properties come with the drawback that programming and modeling errors are often hard to find. In this paper we propose an integrated framework for run-time debugging of equation-based modeling languages. The framework integrates classical debugging techniques with special techniques for debugging EOO languages and is based on graph visualization and interaction. The debugging framework targets the Modelica language.

Keywords: Run-time Debugging, Modeling and Simulation, Graph Visualization and Interaction, Eclipse.

1. Introduction

The development in system modeling has come to the point where complete modeling of systems is possible, e.g. the complete propulsion system, fuel system, hydraulic actuation system, etc., including embedded software can be modeled and simulated concurrently. This does not mean that all components are dealt with down to the very smallest details of their behavior. It does, however, mean that all functionality is modeled, at least qualitatively.

Such advanced development of today's complex products requires integrated environments and modeling languages for modeling and simulation. Equation-based object-oriented declarative (EOO) languages are the key approach to physical system modeling and simulation. The increased ease of use, the high abstraction and the expressivity of EOO languages are very attractive properties. However, these attractive properties come with the drawback that programming and model-

ing errors are often hard to find. This paper proposes an integrated framework for run-time debugging of equation-based modeling languages. The framework integrates classical debugging techniques with special techniques for debugging EOO languages and is based on graph visualization and interaction.

The paper is structured as follows: Section 2 presents the background of EOO languages and their current debugging techniques. Section 3 presents our debugging idea, whereas Section 4 presents the debugging framework. Conclusions and future work are presented in Section 5.

2. Background and Related Work

In this section we briefly introduce the Modelica language which is the target language for our debugging framework and describe the current debugging methods for EOO languages.

2.1 Modelica

Modelica [4][5] is a modern language for equation-based object-oriented mathematical modeling of primarily physical systems. A non-profit organization named "Modelica Association" was founded the year 2000 for further development and promotion of Modelica. Several tools implement the Modelica specification, ranging from open-source such as OpenModelica [3], to commercial implementations like Dymola [8] and MathModelica [7].

The language allows defining models in a declarative manner, modularly and hierarchically. The multi-domain capability of Modelica allows combining electrical, mechanical, hydraulic, thermodynamic, etc., model components within the same application model. In short, Modelica has improvements in several important areas:

- *Object-oriented mathematical modeling.* This technique makes it possible to create model components, which are employed to support hierarchical structuring, reuse, and evolution of large and complex models covering multiple technology domains.

- *Acausal modeling.* Modeling is based on equations instead of assignment statements as in traditional input/output block abstractions. Direct use of equations significantly increases re-usability of model components, since components adapt to the data flow context in which they are used. Interfacing with traditional software is also available in Modelica.
- *Physical modeling of multiple application domains.* Model components can correspond to physical objects in the real world, in contrast to established techniques that require conversion to “signal” blocks with fixed input/output causality. In Modelica the structure of the model naturally correspond to the structure of the physical system in contrast to block-oriented modeling tools such as Simulink. For application engineers, such “physical” components are particularly easy to combine into simulation models using a graphical editor.

Hierarchical system architectures can easily be described with Modelica thanks to its powerful component model. The Modelica component model includes the following three items: a) components, b) a connection mechanism, and c) a component framework. Components are connected via the connection mechanism realized by the Modelica system, which can be visualized in connection diagrams. The component framework realizes components and connections, and ensures that communication works over the connections.

For systems composed of acausal components with behavior specified by equations, the direction of data flow, i.e., the causality is initially unspecified for connections between those components. Instead the causality is automatically deduced by the compiler when needed. Components have well-defined interfaces consisting of ports, also known as connectors, to the external world. A component may internally consist of other connected components, i.e., hierarchical modeling.. Figure 1 shows hierarchical component-based modeling of an industry robot.

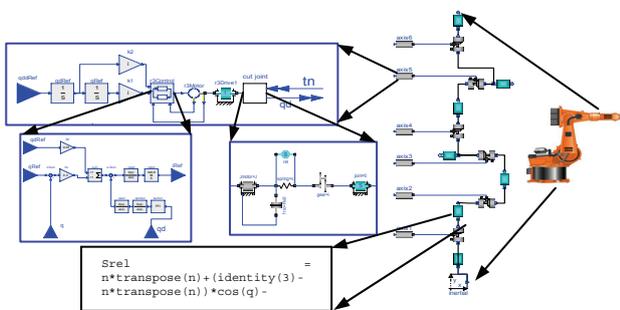


Figure 1. Hierarchical model of an industrial robot, including components such as motors, bearings, control software, etc. At the lowest (class) level, equations are typically found.

2.2 Debugging techniques for EOO languages

In the context of debugging declarative equation-based object-oriented languages both the static 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. *Overconstrained system:* the number of variables is smaller than the number of equations, which means that some equations have to be removed when solving the system of equations.
2. *Underconstrained system:* the number of variables is larger than the number of equations, which means that more equations have to be added in order to solve the system of equations.

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 parameters values for the model simulation are incorrect.
2. *model specification:* when the equations 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 the EOO languages have been proposed earlier [1][2]. With the new Modelica 3.0 language specification, the static 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.

In the context of dynamic (run-time) aspect of debugging of EOO languages, [1] proposes an automated algorithmic 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.

In this paper we present a different approach that does not require the user to write diagnostic specifications of the model. Our method is based the integration between graph visualization/interaction and execution-based debugging of algorithmic code.

3. Proposed Debugging Method

In this section we present our run-time debugging method. The proposed integration within a general debugging framework for EOO languages is presented in the next section.

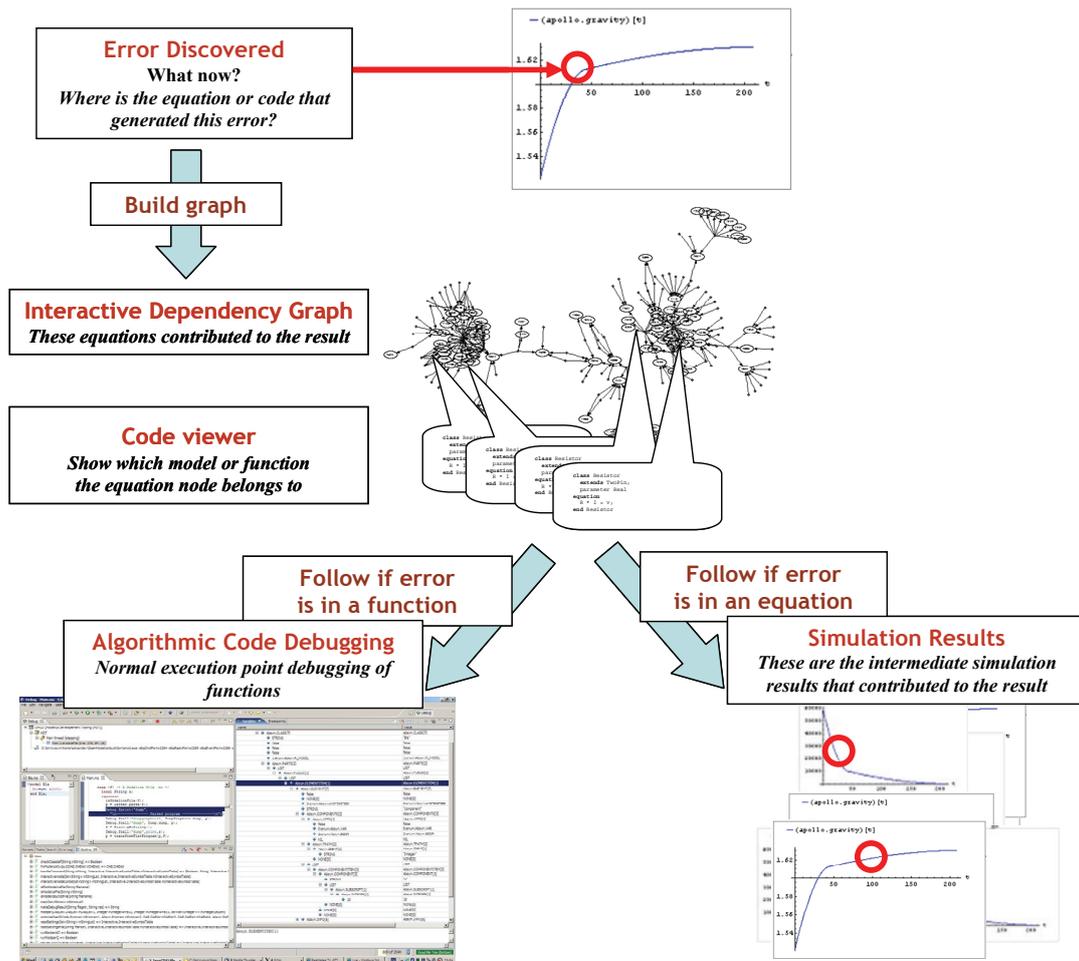


Figure 2. Debugging approach overview.

3.1 Run-time Debugging Method

Our method partly follows the approach proposed in [1]. 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 [6].

The overview of our debugging strategy is presented in Figure 2. In short, our run-time debugging method is based on the integration of the following:

1. 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.

In the following we present a possible debugging session.

During the simulation phase, the user discovers an error in the plotted results. 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

(the dynamic program slice with respect to the variable with the wrong value) where nodes consist of all the equations, functions, parameter value definitions, and inputs that were used to calculate the wrong variable value. The variable with the erroneous value is displayed in a special node which is the root of the graph. The interactive dependency graph 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.

- *Classifying a variable as having wrong values*: addition of the variable to the set of variables with wrong values.
- *Classifying an equation as correct* eliminates the equation node from the graph and builds a new graph based on the inputs of the correct equation node.
- *Building a new dependency graph* based on the new set of variables with wrong values (classified variables) or by modifying the equations or parameter values nodes.
- *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 manifestation to origin.

Our 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.

4. Run-time Debugging Framework

In this section we present the first prototype of the debugging framework based on the proposed method from the previous section. In this paper the debugging framework is limited to error tracking of a single variable with wrong results.

4.1 Translation in the Debugging Framework

The debugging framework is closely related to the translation process. The translation process from the modeling language down to simulation code is presented in the following. The Modelica translation process has several stages (Figure 3):

- *Parser* – breaks the model down into tokens and builds the abstract syntax tree. (not in Figure 3)
- *Translator (Flattening and elaboration)* – reports the errors and flatten the model hierarchy and applying modification.
- *Analyzer* – analyses the system of equations and sorts the equations in the order they need to be solved
- *Optimizer* – optimizes the sorted system of equations
- *Code Generator* – generates C code linked with the simulation runtime and solvers.
- *C Compiler* – compiles the generated C code to an executable
- *Simulation* – the executable is executed to generate the simulation results.

As one can see, the translation process is complex and most of the transformations performed on the models are destructive. For debugging purposes all the transformations performed in each stage needs to be recorded to be able to point the errors to the user using the high level Modelica code.

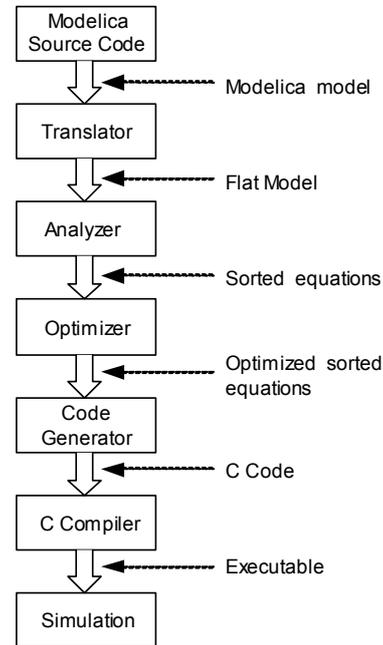


Figure 3. Translation stages from Modelica code to executing simulation.

The debugging framework alters the Modelica translation stages by introducing means to map (and save such mapping) each transformed model element back to its origin as presented in Figure 4.

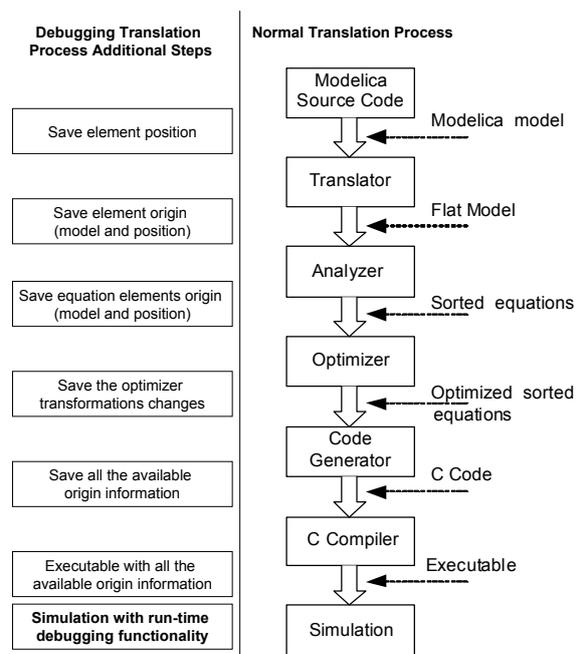


Figure 4. Translation stages from Modelica code to executing simulation with additional debugging steps.

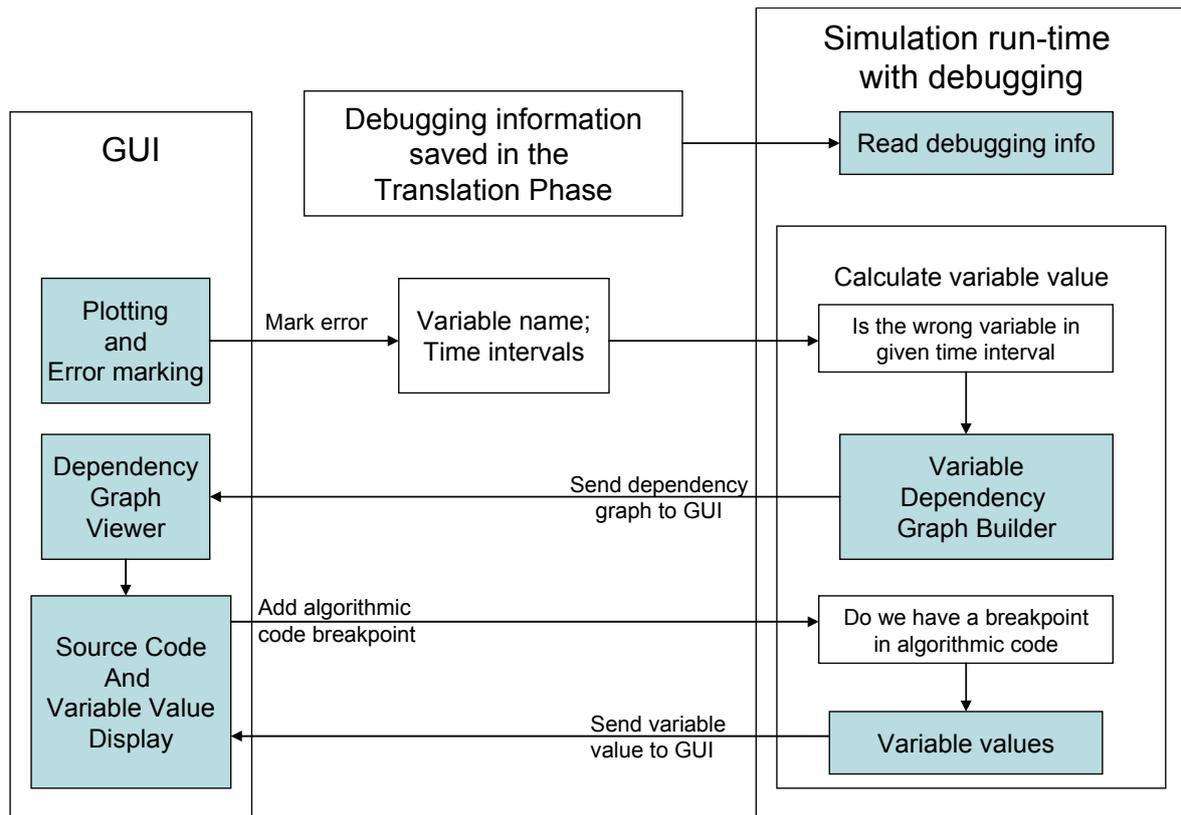


Figure 5. Run-time debugging framework overview.

The additional origin information needed by the debugging framework is saved by the debugging translation process within a file: `debug-info.xml`. The debug file is read by the simulation run-time only when needed.

If an error appears in the simulation results, the user can mark the variable with the wrong value and the error time interval(s) on the simulation plot. The simulation with run-time debugging functionality is then invoked with the error information.

4.2 Debugging Framework Overview

The run-time debugging framework overview is presented in Figure 5. The figure presents the interaction between the components of the graphical user interface (GUI) and the components of the simulation run-time with debugging. Typically, the user debugging starts at the end of a simulation when the user observes the erroneous behavior of a plotted variable value. The user marks the variable name and the time interval and invokes the debugging functionality. The simulation run-time with debugging is then invoked with the user selection as input.

In the next section we detail the debugging framework components.

4.3 Debugging Framework Components

The debugging framework has several components which deal with the user interaction (GUI part) and the

handling of the debugging information (simulation run-time part). The information saved during the translation process also plays an important role in the debugging framework.

4.3.1 Plotting and Error Marking

This GUI component shows the values of a variable during simulation time. The component has special functionality which helps the user to mark an error on the plot using the mouse. The user markings are encoded as a variable name and time intervals. After marking the error, the user invokes the debugging functionality with this marking.

4.3.2 Dependency Graph Viewer

The dependency graph viewer is a GUI component that displays an interactive graph. The graph is given by the dependency graph builder component. The graph shows the calculated variable name and value, the equation in which this value was calculated and all the additional data (parameters, equation blocks, etc) which was used to calculate this value.

In this implementation the user has limited graph interaction possibilities. When the user double clicks on a graph node or edge, the origin of the selected element (variable, equation or parameter) is computed from the debugging information and the Source Code and Variable Value Display component is shown presenting the original source code element.

4.3.3 Source Code and Variable Value Display

The source code display is handled by this component. Also, the user can set breakpoints on the algorithmic code within this view. If the runtime reaches a breakpoint, the execution breaks and the variable values from this model can be examined.

4.3.4 Dependency Graph Builder

The most complex component of the debugging framework is the dependency graph builder. This component starts from a variable name and builds the dependency graph for that variable based on the debugging information saved in the translation phase.

The constructed graph is based on the Block Lower Triangular Dependency Graph (BLTDG) which is computed from the Block Lower Triangular form by considering the data dependencies. The calculation of the BLTDG is presented in detail in [2]. The constructed graph contains also additional information regarding the origin of each involved element.

4.4 Implementation status

Currently we are working at the integration of the debugging framework components.

The debugging framework is developed in Eclipse as a set of plugins that integrate our existing OpenModelica Modelica Development Tooling (MDT) [6] (for code browsing and algorithmic code debugging) with graph visualization and interaction libraries. The OpenModelica Compiler [3] has been adapted to produce the additional debugging information, the dependency graph and the simulation results.

5. Conclusions and Future Work

In this paper we present an integrated run-time debugging framework for EOO languages based on graph visualization and interaction.

We argue that such debugging framework will ease both the run-time debugging and the understanding of EOO languages.

We are aware that the scalability of our method might be an issue and we plan to research different filtering techniques for pruning the dependency graph.

Our short term goal is to finalize the prototype implementation of the proposed debugging framework, evaluate it and report experience on debugging a set of selected models, and release it as part of the OpenModelica Development Environment.

6. Acknowledgements

This work was supported by Vinnova in the Safe & Secure Modeling and Simulation project, by SSF in the

RISE project and by the CUGS and Proviking graduate schools at Linköping University.

References

- [1] Bunus Peter and Peter Frizson. (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, October 6-10, 2003)
- [2] Bunus Peter. Debugging Techniques for Equation-Based Languages. PhD Thesis. Department of Computer and Information Science, Linköping University, 2004.
- [3] Peter Fritzson, Peter Aronsson, Håkan Lundvall, Kaj Nyström, Adrian Pop, Levon Saldamli, and David Broman. *The OpenModelica Modeling, Simulation, and Software Development Environment*. In Simulation News Europe, 44/45, Dec 2005. <http://www.ida.liu.se/projects/OpenModelica>.
- [4] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*, 940 pp., ISBN 0-471-471631, Wiley-IEEE Press, 2004. See also: <http://www.mathcore.com/drmmodelica/>
- [5] The Modelica Association. The Modelica Language Specification Version 2.2. <http://www.modelica.org>
- [6] Adrian Pop, Peter Fritzson, Andreas Remar, Elmir Jagudin, and David Akhvlediani. *OpenModelica Development Environment with Eclipse Integration for Browsing, Modeling, and Debugging*. In Proc of the Modelica'2006, Vienna, Austria, Sept. 4-5, 2006.
- [7] MathCore, *MathModelica* <http://www.mathcore.com>
- [8] Dynasim AB, *Dymola*, <http://www.dynasim.com>