Proceedings
of the 4th International Modelica Conference,
Hamburg, March 7-8, 2005,
Gerhard Schmitz (editor)

P. Bunus
Linköping University, Sweden
An Empirical Study on Debugging Equation-Based Simulation Models
pp. 281-288

Paper presented at the 4th International Modelica Conference, March 7-8, 2005,
Hamburg University of Technology, Hamburg-Harburg, Germany,
organized by The Modelica Association and the Department of Thermodynamics, Hamburg University of Technology

All papers of this conference can be downloaded from
http://www.Modelica.org/events/Conference2005/

Program Committee

- Prof. Gerhard Schmitz, Hamburg University of Technology, Germany (Program chair).
- Prof. Bernhard Bachmann, University of Applied Sciences Bielefeld, Germany.
- Dr. Francesco Casella, Politecnico di Milano, Italy.
- Dr. Hilding Elmqvist, Dynasim AB, Sweden.
- Prof. Peter Fritzson, University of Linkping, Sweden
- Prof. Martin Otter, DLR, Germany
- Dr. Michael Tiller, Ford Motor Company, USA
- Dr. Hubertus Tummescheit, Scynamics HB, Sweden

Local Organization: Gerhard Schmitz, Katrin Prölß, Wilson Casas, Henning Knigge, Jens Vasel, Stefan Wischhusen, TuTech Innovation GmbH

An Empirical Study on Debugging Equation-Based Simulation Models

Peter Bunus
Department of Computer and Information Science
Linköping University, Sweden
petbu@ida.liu.se

Abstract

A typical problem which often appears in Modelica models is when too many/few equations are specified. This leads to a situation where the simulation model is inconsistent and therefore cannot be compiled and executed. We propose a methodology for detecting and repairing over- and under-constrained situations based on graph theoretical methods. Components and equations that cause the irregularities are automatically isolated, and meaningful error messages for the user are presented. The potentially large number of error fixing alternatives is reduced by applying filtering rules extracted from the modeling language semantics.

The paper illustrates that it is possible to localize and repair a significant number of errors during static analysis of a Modelica model without having to execute the simulation model. In this way certain numerical failures can be avoided later during the execution process. The paper proves that the result of structural static analysis performed on the underlying system of equations can effectively be used to statically debug real models.

Keywords: Modelica, debugging, structural and static analysis, mathematical modeling, structural validation.

1 Introduction

Mathematical modeling and simulation of complex physical systems is emerging as a key technology in engineering. Modern approaches to physical system simulation allow users to specify simulation models with the help of equation-based languages. Such languages have been designed to allow automatic generation of efficient simulation code from declarative specifications. Complex simulation models are created by combining available model components from user-defined libraries. The resulted models are compiled in a simulation environment for efficient execution.

Unfortunately, errors are made and inconsistencies are easily introduced in the simulation models. A significant part of the model development effort is spent on detecting deviations from specifications and subsequently localizing the sources of such errors. A typical problem which often appears in physical system modeling and simulation is when too many/few equations are specified in a system. This leads to a situation where the simulation model is inconsistent and therefore cannot be compiled and executed. The user should deal with over- and under-constrained situation by identifying the minimal set of equations or variables that should be removed from the system in order to make the remaining set of equations solvable. For example, if there are too many equations in a system of 100 000 equations, which equations should be removed? Currently the only systematic technique is to remove equations one by one until the equation that caused the inconsistency is identified and finally removed from the system. It can easily be imagined that, if a static debugger presents a small subset of over-constraining equations, from which the user can select the equation that needs to be eliminated from the overall model can greatly reduce the amount of time required to get the simulation working.

Currently there are essentially no advanced tools that can handle the debugging of equation-based languages at the source code level and provide useful error fixing solutions. The aim of the research presented in this paper is to considerably improve the situation, especially with respect to debugging the Modelica language. However, powerful graph-theoretic methods can help to pinpoint possible candidates for erroneous equations. A dramatic reduction in the number of erroneous equation candidates can be achieved by applying new methods such as semantic filtering.

In this paper we describe an empirical evaluation of debugging of automated debugging techniques for detecting and repair structural inconsistencies in equation-based simulation models. We focus on performance of debugging tools that use static analysis tools integrated into a Modelica compiler where the main purpose was to reduce the number of debugging alternatives when structural inconsistencies were present in the model. Static analysis techniques only involve statically available information, such as which variables are present in which equations in and equation-based model. No assumptions regarding the inputs and outputs of the simulation models are made. The development of static and dynamic techniques for equation-based languages have been addressed by our previous research (Bunus 2004 [1], Bunus and Fritzson 2003 [2], Bunus and Fritzson 2004 [3]).

The remainder of the paper is organized as follows: Section 2 presents the problem formulation and a motiva-

tional example. Section 3 gives a brief description of the algorithms for detecting and debugging over-constrained situations that arise during the modeling phase with equation-based languages. Section 4 presents an evaluation of our debugging framework based on several benchmarks. Section 5 presents the overall architecture of a prototype debugger developed in the context of a Modelica compiler. Finally Section 6 presents our conclusions and future work.

2 Problem Formulation and Motivational Example

Mathematical modeling proceeds by specifying a set of mathematical equations or functional relations denoted $E = \{e_1, \dots e_n\}$ involving a set of variables denoted $V = \{v_1, \dots v_m\}$. In the general case a system of *n* equation with *m* variables or unknowns can be described by the following equality:

$$e_i(v_1, \dots v_m) = c_i \quad (2.1)$$

where c_i are constants and $i = 1 \dots n$. Solving the system of equations *E* is the problem of finding the set of solutions $S = \{(s_1, \dots s_m) \in T^m \mid e(s_1, \dots s_n)\}$ where *T* is the domain of equations, which fulfill the equality (2.1). The relation (2.1) can be expanded into:

$$\begin{aligned} &a_{11}v_1 + \cdots a_{1m}v_m = c_1 \\ &\vdots \\ &a_{n1}v_1 + \cdots a_{nm}v_m = c_n \end{aligned} \quad (2.2)$$

where $a_{ij}, i = 1 \dots n, j = 1 \dots m$ are real coefficients. In a matrix-vector notation, (2.2) has the form: $\mathbf{Av} = \mathbf{c}$

$$\text{where } \mathbf{A} = \begin{pmatrix} a_{11} & \cdots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nm} \end{pmatrix} \mathbf{v} = \begin{pmatrix} v_1 \\ \vdots \\ v_m \end{pmatrix} \text{ and } \mathbf{c} = \begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix} \quad (2.3)$$

A necessary condition for the existence and uniqueness of a solution *S* is that matrix $\mathbf{A}$ is a square matrix (the number of equations is equal to the number of variables) and there exists permutations $\mathbf{P}_1\mathbf{P}_2$ such that $\mathbf{P}_1\mathbf{A}\mathbf{P}_2$ has a nonzero diagonal. This condition guarantees the structural singularity of the system of equations. The structural singularity checks whether the system of equations is well-posed or not. It is only a necessary but not sufficient condition for the existence and uniqueness of a solution. The more powerful notion of numerical singularity will guarantee the existence and uniqueness of a solution. However the checking the numerical singularity is as expensive as solving the system of equations. Therefore when analyzing the system of equations in this stage we assume that the structural non-singularity is a sufficient abstraction for implying that the equation system has a unique solution. Further analysis based on numerical values and numerical singularities is delayed until the dynamic analysis stage.

If the system of equations is structurally singular we switch from the problem of finding the set of solutions *S* to the problem of finding the maximal subset of equations $E_S = \{e_1, \dots e_t\}$ where $t < n$ and $E_S \subset E$ if $n > m$ (we have more equations than variables) or to the to the problem of finding the maximal subset of variables $V_S = \{v_1, \dots v_k\}$ where $k < m$ and $V_S \subset V$ if $n < m$ (we have more variables than equations).

As an example let us consider a Modelica model consisting of a sinusoidal voltage source and a resistor connected together. This model is trivial, but it serves as a straightforward vehicle for introducing several fundamental debugging concepts.

```
connector Pin
  Voltage v;
  Flow Current i;
end Pin;

model TwoPin
  Pin p, n;
  Voltage v;
  Current i;
equation
  v = p.v - n.v; 0 = p.i + n.i; i = p.i
end TwoPin;

model Resistor
  extends TwoPin;
  parameter Real R;
equation
  R*i = v;
end Resistor;

model VsourceAC
  extends TwoPin;
  parameter Real VA=220;  parameter Real f=50;
  protected constant Real PI=3.141592;
equation
  v=VA*(sin(2*PI*f*time));
end VsourceAC;

model Ground
  Pin p;
equation
  p.v = 0
end Ground;

model Circuit
  Resistor R1(R=10); VsourceAC AC; Ground G;
equation
  connect(AC.p,R1.p); connect(R1.n,AC.n);
  connect(AC.n,G.p);
end Circuit;
```

We introduce an additional equation (`i=23`) inside the `Resistor` component in order to over-constrain the simulation model. The flattened equations corresponding to the `Circuit` model is depicted in Figure 1.

eq1	`R1.v = -R1.n.v + R1.p.v`	var1	`R1.p.v`
eq2	`0 = R1.n.i + R1.p.i`	var2	`R1.p.i`
eq3	`R1.i = R1.p.i`	var3	`R1.n.v`
eq4	`R1.i R1.R = R1.v`	var4	`R1.n.i`
eq5	`R1.i = 23`	var5	`R1.v`
eq6	`AC.v = -AC.n.v + AC.p.v`	var6	`R1.i`
eq7	`0 = AC.n.i + AC.p.i`	var7	`AC.p.v`
eq8	`AC.i = AC.p.i`	var8	`AC.p.i`
eq9	`AC.v = AC.VA*sin[2*time*AC.f*AC.PI]`	var9	`AC.n.v`
eq10	`G.p.v = 0`	var10	`AC.n.i`
eq11	`AC.p.v = R1.p.v`	var11	`AC.v`
eq12	`AC.p.i + R1.p.i = 0`	var12	`AC.i`
eq13	`R1.n.v = AC.n.v`	var13	`G.p.v`
eq14	`AC.n.v = G.p.v`	var14	`G.p.i`
eq15	`AC.n.i + G.p.i + R1.n.i = 0`		

Figure 1. Flattened equations and variables corresponding to the `Circuit` model.

It should be noted that the number of equation is greater than the number of variables and therefore we are facing a structurally nonsingular problem.

3 Debugging Over- and Under-constrained Models

The methods proposed in this section present a strategy to deal with overdeterminancy by identifying the minimal set of equations that should be removed from the system in order to make the remaining set of equations solvable. The idea is to isolate the over-constraining part of the bipartite graph associated to the underlying system of equations and to perform reasoning based on specific properties of the specified subgraph. Efficient graph transformations, based on rules derived from the semantics of the modeling language are also performed on the subgraphs. We are going to show how these rules are automatically derived from the modeling language semantics and how the associated annotations to the equations contribute to the filtering of the combinatorial explosion of possible error fixing solutions. Those interested in more details may wish to consult Bunus and Fritzson 2004 [3] or Bunus 2004 [1].

Step 1: Isolating the over-constraining part.

In step 1, from the flattened intermediate form of the equations the associated bipartite graph is derived and a maximum cardinality matching is found. The Dulmage Mendelsohn canonical decomposition (Dulmage and Mendelsohn 1963 [4]) will lead to two different subgraphs: a well-constrained part W_G and an over-constrained part O_G^{1+} as depicted in Figure 2. The maximum cardinality matching is shown in Figure 2 with bold edges.

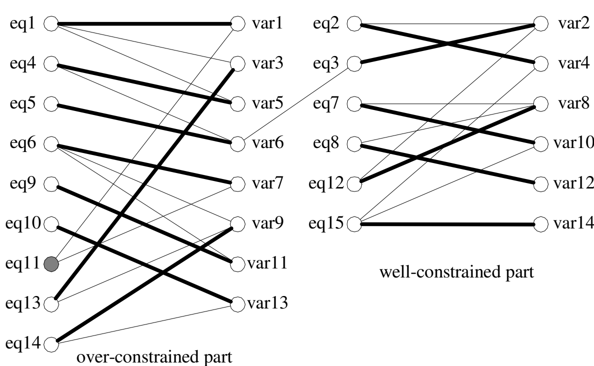

Figure 2. Canonical decomposition of an over-constrained system.

It can be seen that equation *eq11* is not covered by the found maximum cardinality matching. Therefore equation *eq11* is a non-saturated or free vertex of the equation set, therefore it is a source for the over-constrained part O_G^{1+}. Next, starting from *eq11*, the directed graph can be derived from the undirected bipartite graph, as illustrated in Figure 3, by exchanging all the matching edges into bidirectional edges and orienting all other edges from equation to variable nodes. The layout of the directed graphs derived from the undirected bipartite graphs has been rearranged into a tree representation for the purpose of increasing understandability for the reader of the paper.

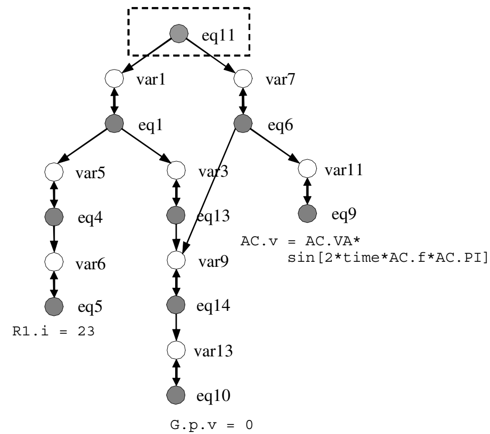

Figure 3. A directed graph associated to the over-constrained part.

Step 2: Reducing the over-constraining equations by using structural information.

The general error fixing strategy in the case of over-constrained equation subsystems is to remove the extra equations. An immediate fix to the over-constrained part is to remove one of the equation nodes, which will lead to a well-constrained part. However, as it can be seen from Figure 4, not all the equation edges can be safely removed.

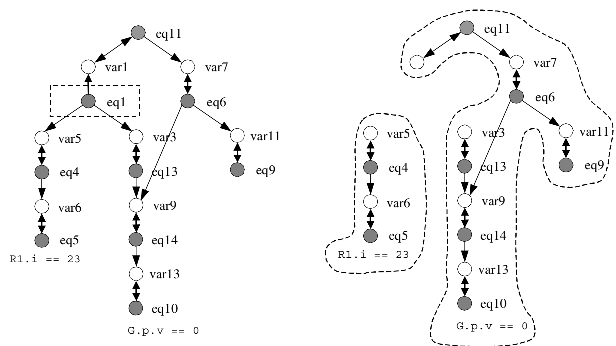

Figure 4. The elimination of an unsafe equation node (*eq1*) from the over-constrained subgraph (on the left) leads to two disconnected components (on the right).

By removing an equation node and the corresponding incident edges from the bipartite graph the remaining undirected graph must remain connected. In our particular example the set of over-constraining equations that satisfy this condition is {*eq11, eq13, eq10, eq5, eq9*}. It should be noted that the safe removal of equation nodes only refers to the bipartite graph representation of the intermediate code of the flattened set of equations, and it is influenced by only structural properties of the bipartite graph. If we would like to further reduce this set of equations,

removal criteria derived from the semantics of the modeling language would need to be developed and included in the debugging strategy.

Step 3: Reducing the over-constraining equations by using semantic information

As we have seen in the previous example not all the over-constraining equations are possible to remove without causing further structural failures in the model description. By taking into account simple rules derived from the language semantics we can safely discard some other elimination alternatives as well.

We note that equation *eq11* (`AC.p.v = R1.p.v`) is generated by a `connect` equation from the `Circuit` model and the only way to remove the equation *eq11* is to remove the original `connect(AC.p, R1.p)` equation. However, removing the above-mentioned equation will remove two equations from the flattened model since the `connect` equation expands into two equations. It is obvious that this modification cannot be performed by the user at the original source code level.

In order to provide a mechanism to reason about the erroneous model under consideration based on language semantics rules the equations need to be annotated. We define an annotated equation as a record with the following structure:

```
< Equation,
  Name,
  Description,
  No. of associated equations,
  Class name,
  Flexibility level,
  Connector generated,
  No. of linked equations
>
```

The *Class Name* indicates which class the equation comes from. This annotation is extremely useful in exactly locating the associated class of the equation and therefore providing concise error messages to the user in terms of original source code statements.

The *No. of associated eqs.* field defines the number of equations which are specified together with the annotated equation inside the same model. For an equation that belongs to the `TwoPin` class the number of associated equations is equal to 3. If one associated equation of the class needs to be eliminated the value is decremented by 1. During debugging, if the equation `R1.i * R1.R = R1.v` is diagnosed to be an over-constraining equation and therefore needs to be eliminated, then the elimination is not possible because the model will be invalidated (the *No. of associated eqs.* cannot be equal to 0) and therefore other solutions need to be investigated.

The *flexibility level*, in a similar way as defined in Flannery and Gonzalez 1997 [5], allows the ranking of the relative importance of the equation in the overall flattened system of equations. The value can be in the range of 0 to 3, with 0 representing the most rigid equation and 3 being the most flexible equation. In practice, it turns out that the equations generated by connections are more rigid from the constraint relaxation point of view than the equations specified inside the model. This means that preference is given to repair strategies that involve the removal of equations which defines the behavior of a particular component and not to topology changes of the circuit given by the connection equations. We set the flexibility value to 0 for those equations that should not be removed or modified. These equations are *locked* for editing which means that an automatic debugger should not consider any repair strategy that would involve the modification or the removal of the equations associated to such a component. For example the equations of components that come from well tested and trusted libraries can have this value set to zero.

The *Connector generated* is a `Boolean` attribute which tells whether the equation is generated or not by a `connect` equation. Usually these equations have a very low flexibility level.

The *No. of linked equations* attribute specifies how many other equations are linked with the current equations. Equations that come from `connect` equations or from parent objects (such as the `TwoPin` partial component) have this attribute greater than zero. Removing an intermediate equation that has this attribute greater than zero will trigger the removal of other intermediate additional equations equal to the number of linked equations. This is due to the fact that the removal of an intermediate equation is only possible by removing the original source code that generated that equation. By doing this all the generated intermediate equations by the original equation will be removed.

It is worth noting that the annotation attributes are automatically initialized by the static analyzer. These are incorporated in the front-end of the compiler, by using several graph representations of the declarative object-oriented program code. Therefore the user does not need to manually annotate the source code. A debugger preprocessor takes care of the automatic generation and initialization of the annotating code. In this way a mapping between the intermediate code and original declarative code is kept during the translation phases.

The annotations associated to the set of equivalent over-constraining equations {*eq11, eq13, eq10, eq5, eq9*} are shown in Table 1.

Table 1. The associated annotations of the remaining over-constraining equation set

Name	Equation	No. of assoc. eqs.	Class name	Flex. level	Con. gen.	No.of linked eqs.
eq11	`AC.p.v=R1.p.v`	3	`Circuit`	1	Yes	1
eq13	`R1.n.v= AC.n.v`	3	`Circuit`	1	Yes	1
eq10	`G.p.v=0`	1	`Ground`	2	No	0
eq5	`R1.i=23`	2	`Resistor`	2	No	0
eq9	`AC.v=AC*VA*sin..`	1	`VsourceAC`	2	No	0

The equation node *eq11* was already analyzed and can therefore be removed from the set. Equation node *eq13* is

removed as well, for the same reasons as equation *eq11*. By analyzing the remaining equations {*eq10, eq5, eq9*}, one should note that they have the same flexibility level and therefore candidates for elimination with equal probability. However, by analyzing the value of the *No. of associated eqs.* annotation, equation *eq10* and *eq9* have this attribute equal to one, which means that they are the only equations that define the behavior of the model. Removing one of these equations will invalidate the corresponding model component, which is probably not the intention of the modeler and therefore not acceptable as an error fixing solution.

By examining the annotations corresponding to equation *eq5* one can see that it can safely be removed because its flexibility level is high. The removal of *eq5* will not trigger the removal of any other equation since it has no linked equations (indicated by the value of *No. of linked eqs.* annotation which is equal to 0). Moreover, removing equation *eq5* will not invalidate the model since there is another equation defined inside the `Resistor` model (`R1.i * R1.R = R1.v`) denoted by the value of *No. of associated eqs.* annotation which is equal to 2.

Step 3: Outputting the debugging alternatives.

After selecting the right equation for elimination the debugger tries to identify the associated class of that equation based on the *Class name* parameter defined in the annotation structure. Having the class name and the intermediate equation form (`R1.i=23`), the original equation can be reconstructed (`i=23`) to exactly indicate to the user the equation that needs to be removed in order to make the simulation model well-constrained. In this case the debugger correctly located the faulty equation previously introduced by us in the simulation model.

When multiple valid error fixing solutions are possible and the debugger cannot decide which one to choose, a ranked list of error fixes is presented to the user for further analysis and decision. In those cases, the user must take the final decision, as the debugger cannot know or does not have enough information to decide which equation is over-constraining. The advantage of this approach is that the debugger automatically identifies and solves several anomalies in the declarative simulation model specification without having to execute the system.

When debugging under-constrained systems (more variables than equations are present in the system) two distinct strategies can be considered. The first strategy considers the removal of the free variables while the second strategy considers the addition of new equations to the overall system of equations, which must contain the free variables. Additionally, the second strategy takes into account extra variables that can be added to the introduced new equation. New equations can be introduced at different levels in the object hierarchy.

4 Experimental Validation

In this paper we are interested in the quality of structural and semantics filtering rules employed in the proposed static debugging algorithm for correcting over- and under-constrained system of equations extracted from simulation models expressed in the Modelica language.

Firstly, we have modified several working simulation models by inserting additional equations in the model definitions at various places, thereby over-constraining the whole system models. In this first set of experiments we were interested if over-constraining situations are detected and how many repair possibilities are reported by the debugger.

A short description of the benchmark programs and the over-constraining nature for each example is given in Table 2. The measurements in Table 2. were performed as follows. We built several Modelica simulation models that were structurally correct. Then we have modified each example by inserting an extra equation in different components of the simulation model. In this way the models became over-constrained. During the translation phase the system of flattened equation and each equation was annotated. In the next step a canonical decomposition was performed on the structurally singular system of flat equations and the over-constraining graph was isolated. Based on the over-constraining graph the reduced set of

Table 2. Benchmark program description for over constrained systems.

Test model	Description	No. of var.	No of eq.	Over contr. part	Red. over constr. part	Semantic filtering	Debugging alt.
scircuitR[1+]	A simple electrical circuit model consisting of a resistor connected in parallel with a continuous voltage source. The `Resistor` component is over-constrained by an extra equation.	14	15	9	5	1	1
scircuitPin[3+]	A simple electrical circuit model consisting two resistors connected in parallel with a direct current source. The `TwoPin` component is over-constrained by one extra equation.	20	23	19	15	3	1
generatorR[2+]	A generator circuit model similar where the `Resistor` component is over-constrained by one an extra equation.	49	51	35	22	6	3
dcmotorR[1+]	A direct current motor circuit model where the `Resistor` component is over-constrained by one an extra equation.	36	37	31	29	7	7

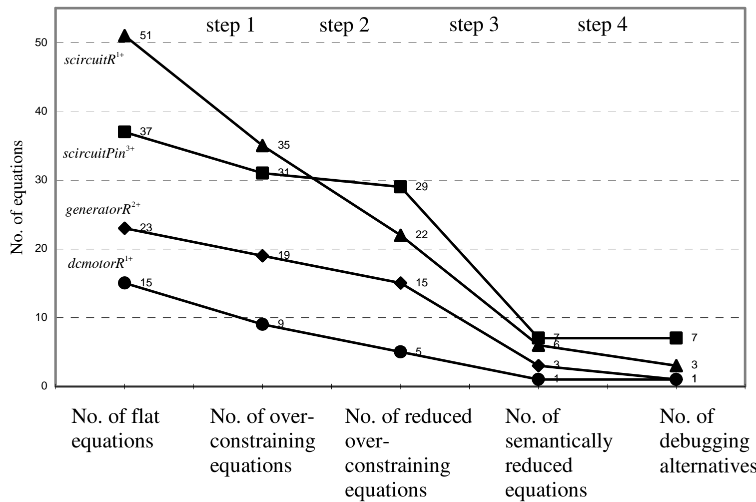

Figure 5. Number of over-constraining equations obtained after each reduction step during structural debugging.

over-constraining equations was computed. This set of equations was further reduced by using semantic filtering rules. Based on this final set of equation the error messages are output to the user. The numbers of debugging alternatives are shown in the last column of Table 2. Figure 5 depicts the number of over-constraining equations obtained after each reduction step.

Secondly, we investigated the detection capabilities of the static debugger when under-constrained situation were purposely introduced in the simulation model by deleting equations or adding extra variables in the system. The modifications performed on each model are described Table 3. The debugging of the under-constrained system was performed by considering only those corrections that imply the removal of a free variable from the system. Figure 6 displays the number of under-constraining variables after each reduction step. After each step during the structural debugging the number of free variables that can be removed from the system is dramatically reduced. It should be noticed in Figure 6 that the largest reduction in the number of free variables and implicitly in the number of debugging alternatives presented to the user is achieved by the semantic filtering phase.

We are interested in the quality of structural and semantics filtering rules employed in the proposed static debugging algorithm for correcting over- and under-constrained system of equations extracted from simulation models ex-

Table 3. Benchmark program description for under-constrained systems

Test model	Description	No of eq.	No. of var.	Under contr. part	Red. over constr. part	Semantic filtering	Debugging alt.
scircuitR^{1-}	A simple electrical circuit model consisting of a resistor connected in parallel with a continuous voltage source. In the `Resistor` component an extra variable has been declared (`Real s`) and the equation `R*i=v*s` was introduced instead of the correct equation `R*i=v`.	14	15	8	5	1	1
scircuitPin^{3-}	A simple electrical circuit model consisting two resistors connected in parallel with a direct current source. The `TwoPin` component is under-constrained by introducing an extra variable (`Real s`) and by exchanging equations `0 = p.i + n.i` with `s = p.i + n.i`.	20	23	7	4	3	1
generatorR^{2-}	A generator circuit model. In the Resistor component an extra variable has been declared (`Real s`) and the equation `R*i=v*s` was introduced instead of the correct equation `R*i=v`.	49	51	28	21	3	1
dcmotorR^{1-}	A direct current motor circuit model. In the `Resistor` component an extra variable has been declared (`Real s`) and the equation `R*i=v*s` was introduced instead of the correct equation `R*i=v`.	37	36	30	28	4	4

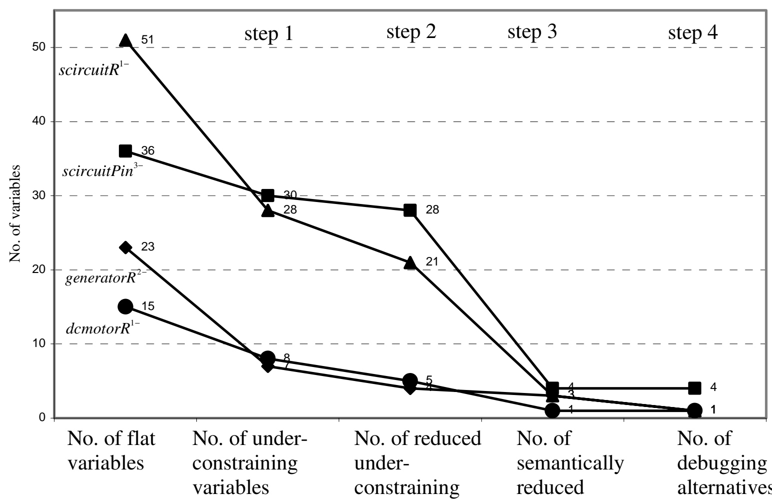

Figure 6. Number of under-constraining variables obtained after each reduction step during structural debugging.

pressed in the Modelica language. Table 4 shows the percentage reduction in the number of equations/variables that need to be examined by user after each step in the debugging process.

Table 4. Percentage reduction of the number of equation/variables that need to be examined by the user after each reduction step.

Test model	No. of flat eq./.var	Step1	Step 2	Step3	Step 4
scircuitR^{1+}	15	40.0%	66.7%	93.3%	93.3%
scircuitPin^{3+}	23	17.4%	34.8%	87.0%	95.7%
generatorR^{2+}	51	31.4%	56.9%	88.2%	94.1%
dcmotorR^{1+}	37	16.2%	21.6%	81.1%	81.1%
scircuitR^{1-}	15	46.7%	66.7%	93.3%	93.3%
scircuitPin^{3-}	23	69.6%	82.6%	87.0%	95.7%
generatorR^{2-}	51	45.1%	58.8%	94.1%	98.0%
dcmotorR^{1-}	36	16.7%	22.2%	88.9%	88.9%

As can be seen in Figure 5, Figure 6 and from the percentage reduction Table 4, the proposed algorithm for debugging over- and under-constrained systems is very efficient in reducing the number of debugging alternative shown to the user. On the average, 91% of the irrelevant candidates were eliminated, which allows the user to look for the bug among the few remaining candidates, thus dramatically improving bug localization effectiveness.

5 Implementation

For the previously presented graph decomposition techniques to be useful in practice, we must be able to construct and manage the graph representation of equation-based specifications efficiently and integrate them into an automatic or semi-automatic debugging tool. The use of graph-based tools in structural analysis is of great interest both in displaying properties of systems of equations and also in following and performing symbolic manipulations of variables and equations when modeling with equation-based languages (Harman 2005 [6]). We show how existing graph theoretical decomposition techniques can be adapted and integrated into debugging tools integrated into simulation environments that employs such languages.

At this stage we are able to provide an overview of the proposed framework developed for the Modelica language and Modelica-based simulation environments. Even if we have limited our prototype implementation to the Modelica language, the developed debugging kernels can easily be adapted to handle other object-oriented equation-based languages as well. It is important to note that the proposed debugging framework can easily be integrated into the existing Modelica compilers.

AMOEBA (Automatic MOdelica Equation-Based Analyzer) is the static analysis module that we have designed and implemented in order to attach it to a Modelica compiler. The tool is able to successfully detect and provide error-fixing solutions for typical over and under-constrained situations, which might appear during the modeling stage using Modelica. Figure 7 show the general architecture of our static debugger.

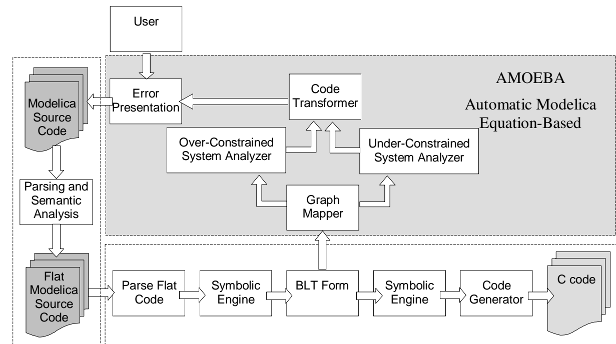

Figure 7. AMOEBA integration into the compilation framework.

Below we present each phase of the static analysis with the corresponding module:

The flattened equations are transformed into the bipartite graph representation by a *Graph Mapping* module. The canonical decomposition algorithm applied by the *BLT* module in the compiler splits the graph into three distinct subgraphs corresponding to an over-constrained system of equations (too many equations are present), an under-constrained system (too few equations or too many variables are present in the system) and a well-constrained system of equations (the number of variables is equal to the number of equations). A simple heuristic filtering rule assumes that the well-constrained part obtained after decomposition will lead to a solvable system of equations and therefore need not be included in any repair strategy. If under- or over-constrained situations are detected, this means that there are some inconsistencies in the model.

The *Over-* and *Under-Constrained System Analyzers* applies the algorithms presented in previous sections, in order to transform these graphs into a well-constrained graph and elaborate the necessary program modifications.

The *Code Transformer* module needs to validate the program correction: it must assure that there exists a semantically correct source code program that can be translated into the intermediate program correction. The source code transformations must be performed only by using atomic changes at the original source code level. Finally, the error fixing solution is output by the debugger in terms of atomic changes that need to be performed on the original source code in order to obtain a valid original source code program that will generate the corresponding program modifications at the intermediate code level. When multiple error fixing solutions exist, the annotations attached to the flattened equations are used in the process of eliminating some of the modifications and prioritizing the remaining ones.

The *Error Presentation* module is responsible for presenting error messages to the user based on the previously

obtained valid source code modifications. Before being presented to the user, the output is filtered. For example, all the modifications that would involve atomic changes on locked components are eliminated and the remaining corrections are ranked based on equations annotations. This module handles most of the user interaction necessary for the debugger to complete the missing formal specification of the program. At this level the user can be confronted with several error fixing corrections that will eliminate the symptom of the detected inconsistency at the intermediate code level. The corrections that most closely correspond to the programmer's view of the model structure should be selected.

6 Conclusions

Structural analysis techniques are widely used for assessing the correctness and the credibility of mathematical models expressed with the help of equations. Experience has taught us that pre-processing a system of equations pays high dividends by reducing the time for finding inconsistencies and efficiently correcting them. From the user point of view, such techniques are extremely beneficial because they provide guidance during early stages of the simulation model building process and do not require solving the equations system.

The paper illustrates that it is possible to localize and repair a significant number of errors during static analysis of object-oriented equation-based modeling languages without having to execute the simulation model. In this way certain numerical failures can be avoided later during the execution process. The paper proves that the result of structural static analysis performed on the underlying system of equations can effectively be used to statically debug Modelica simulation models.

This paper describes one of the first experimental studies on how these new static debugging techniques perform on erroneous model examples. We have presented an empirical evaluation of proposed static analysis based debugging paradigm for equation-based languages. Our studies demonstrated that static analysis can dramatically reduce debugging time, suggesting the potential of structural analysis as a highly effective approach.

Currently, the debugger’s functionality is limited mostly due to our inability to compile the full Modelica language. Therefore only a limited number of real world examples with limited size and complexity have been tested. The integration of the presented debugging techniques into the Open Source Modelica framework is underway. In order to provide a complete debugging framework for the Modelica language we intent to integrate the proposed structural analysis techniques with the existing debugger for the algorithmic subset of the Modelica language proposed by Pop and Fritzson 2005 [7].

We claim that the techniques developed and proposed in this paper are suitable for a wide range of equation-based languages and not only for the Modelica language. These techniques can be easily adapted to the specifics of a particular simulation environment. Our claim is based on the *close integration of the developed debugging techniques and the compilation process*. Most of the existing compilers for equation based languages share the same principles.

Acknowledgements

This research was supported by Center for Industrial Information Technology (grant CENIIT 05.02) at Linköping University Sweden.

REFERENCES

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

[2] Bunus Peter and Peter Fritzson. (2003). "Semi-automatic Fault Localization and Behaviour Verification for Physical System Simulation Models." In Proceedings of the 18th IEEE International Conference on Automated Software Engineering. (Montreal, Canada, October 6-10, 2003).

[3] Bunus Peter and Peter Fritzson. (2004) "Automated Static Analysis of Equation-Based Components." Simulation: Transactions of the Society for Modeling and Simulation International. Special Issue on Component Based Modeling and Simulation., vol. 80: 8, August 2004.

[4] Dulmage A.L. and N.S. Mendelsohn. (1963) "Coverings of bipartite graphs." Canadian J. Math, vol. 10, pp. 517-534.

[5] Flannery L. M. and A. J. Gonzalez. (1997) "Detecting Anomalies in Constraint-based Systems." Engineering Applications of Artificial Intelligence, vol. 10: 3, pp. 257-268.

[6] Harman Peter. (2005). " Visualisation of Model Transformation Algorithms for a Modelica Translator." In Proceedings of the 4th International Modelica Conference. (Hamburg, Germany, 7-8 March, 2005).

[7] Pop Adrian and Peter Fritzson. (2005). "A Portable Debugger for Algorithmic Modelica Code." In Proceedings of the 4th International Modelica Conference. (Hamburg, Germany, 7-8 March, 2005).