Semantics Guided Filtering of Combinatorial Graph Transformations in Declarative Equation-Based Languages.

Peter Bunus, Peter Fritzson
Department of Computer and Information Science, Linköping University,
SE-581-32 Linköping, Sweden
{petbu,petfr}@ida.liu.se

Abstract

This paper concerns the use of static analysis for debugging purposes of declarative object-oriented equation-based modeling languages. We propose a framework where over- and under-constraining situations present in simulation models specified in such languages are detected by combinatorial graph transformations performed on the flattened intermediate code and filtered by the semantic transformation rules derived from the original language. This is powerful enough to statically detect a broad range of errors without having to execute the simulation model. Debuggers associated with simulation environments for such languages can provide efficient error-fixing strategies based on the graph-based representation of the intermediate code. The emphasis, in this paper, is on detecting and debugging over-constraining equations, which are present in some simulation model specifications. We discuss various ways in which we have extended our approach to allow static global analysis of the original modeling source code.

1. Introduction

Sophisticated engineering systems are inherently complex. In order to support mathematical modeling and simulation of such systems, a number of object-oriented and/or declarative acausal modeling languages have emerged. The advantage of such a modeling language is that the user can concentrate on the logic of the problem rather than on a detailed algorithmic implementation of the simulation model. On the other hand, the high level of abstraction of such models presents new challenges to modeling and simulation tools, since there is a wider gap down to the executable machine code. The process of translating models to efficient code becomes considerably more involved. The large abstraction gap between the high-level models and the executing code also leads to difficulties in finding and correcting model inconsistencies and errors, not uncommon in complex physical system models. Currently there are essentially no tools that can handle debugging of equation-based languages in an efficient manner and provide error-fixing information at the original source code level.

A typical problem that appears in physical system modeling and simulation is when too many equations are specified in the system inevitably leading to an inconsistent state of the simulation model. In such situations numerical solvers fail to find correct solutions to the underlying system of equations. The user should deal with over-determined systems of equations by identifying the minimal set of equations that should be removed from the system in order to make the remaining set of equations solvable. For example, consider a physical system simulation model specified in a declarative object-oriented equation-based modeling language that consist of several hundreds of classes resulting in several thousands of flattened equations. However one of these equations over-constrain the overall system making it impossible to simulate. It can be easily imagined that, for example, a small subset of over-constraining equations provided, by a static debugger, which need to be eliminated from the overall model in order to form a structurally well posed simulation problem can greatly reduce the amount of time required to get the simulation working.

The paper is organized as follows: Section 2 introduces Modelica, a new equation-based declarative language. Some specific Modelica language constructs necessary to understand the paper are given in Section 3. Then Section 4 presents a simple simulation model specified using Modelica. Section 5 gives some preliminary definitions related to the concept of bipartite graphs. In the next section, the simple electrical circuit model from Section 4 is modified by introducing an over-constraining equation. Then, we show how the simulation model is transformed into intermediate code and how this code is mapped into bipartite graphs. In Section 7, the combinatorial graph algorithm is presented for the case of debugging over-constrained systems. Section 8 presents how the combinatorial explosion of error fixing solutions obtained by the debugger are substantially filtered by semantics constraints of the Modelica language. Finally, Section 9 concludes and summarizes the work.

2. Modelica - A Declarative Object Oriented Equation Based Modeling Language.

Modelica is a new multi-paradigm language for hierarchical object-oriented modeling and computational applications, which is developed through an international effort[4][5][3]. The language is one of the rare examples of a programming language that combines declarative functional programming with object-oriented programming. Modelica integrates several programming paradigms:

- Declarative functional programming using equations and functions without side effects.
- Object-oriented programming.
- Constraint programming based on equations.
- Architectural system specification with connectors and components.
- Concurrency and discrete event programming, based on the synchronous data flow principle.
- Visual programming based on connecting icons with ports, and hierarchical decomposition.

Additionally, the multi-domain capability of Modelica gives the user the possibility to combine model components from different application domains within the same application model, e.g. combining electrical, mechanical, hydraulic, thermodynamic, control, algorithmic components. Modelica is primarily a modeling language, sometimes called hardware description language, that allows the user to specify mathematical models of complex systems, e.g. for the purpose of computer simulation of dynamic systems where behavior evolves as a function of time. Modelica is also a declarative object-oriented equation based programming language, oriented towards computational applications with high complexity requiring high performance.

- Modelica is primarily based on equations instead of assignment statements. This permits acausal modeling that gives better reuse of classes since equations do not specify a certain data flow direction. Thus a Modelica class can adapt to more than one data flow context. Algorithmic constructs including assignments are also available in a way that does not destroy the declarative properties of the language.
- Modelica has multi-domain modeling capability, meaning that model components corresponding to physical objects from several different domains such as e.g. electrical, mechanical, thermodynamic, hydraulic, biological and control applications can be described and connected.
- Modelica is an object-oriented language with a general class concept that unifies classes, generics – known as templates in C++, and general subtyping into a single language construct. This facilitates reuse of components and evolution of models.
- Modelica has a strong software component model, with constructs for creating and connecting components. Thus the language is ideally suited as an architectural description language for complex systems.

The reader of the paper is referred to [8][7] and [9][5] for a complete description of the language and its functionality from the perspective of the motivations and design goals of the researchers who developed it.

3. Declarative Object-Oriented Equation Based Language Constructs

In this section, we briefly survey some important language structures and constructs that are present in Modelica. Only those language constructs are presented that are necessary for the understanding of the paper.

3.1 Modelica Classes

Modelica, like any object-oriented computer language, provides the notions of classes and objects, also called instances, as a tool for solving modeling and programming problems. Every object in Modelica has a class that defines its data and behavior. A class has three kinds of members:

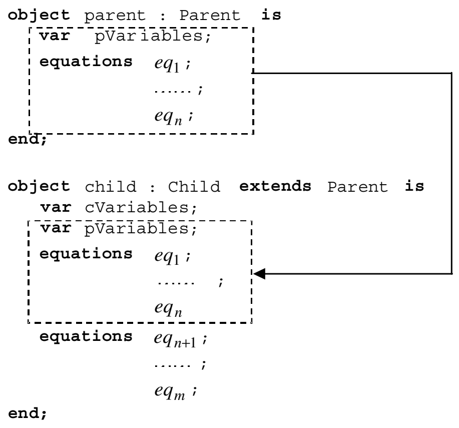
- *Fields* are data variables associated with a class and its instances. Fields store results of computations caused by solving the equations of a class together with equations from other classes.
- *Equations* specify the behavior of a class. The way in which the equations interact with equations from other classes determines the solution process, i.e. program execution.
- *Classes* including functions can be members of other classes.

Here is the declaration of a simple class that might represent a point in a three-dimensional space:

```
class Point "point in a three-dimensional
space"
  public Real x;
  Real y, z;
end Point;
```

3.2 Inheritance

The Modelica view on object-orientation is different since the Modelica language emphasizes *structured* mathematical modeling. Object-orientation is viewed as a structuring concept that is used to handle the complexity of large system descriptions. A Modelica model is primarily a declarative mathematical description, which simplifies further analysis. The natural inheritance mechanism in Modelica and in most of the existing declarative equation based modeling languages is seen as the extension of existing objects with new equations and variables as it is illustrated in Figure 1. Such an inheritance is called embedding inheritance according to [1]. This is in fact the most common inheritance mechanism for object-oriented equation based languages and the one from which we derive, later on, the rules for the graph transformations associated to the intermediate form. The embedding inheritance presents some important benefits, especially from the modeling languages point of view, such as: better support for concrete visual programming systems and any objects can be given individualized behavior.



```
object parent : Parent  is
  var  pVariables;
  equations  eq1 ;
             ...... ;
             eqn ;
end;

object child : Child  extends Parent  is
  var cVariables;
  var  pVariables;
  equations  eq1 ;
             ......  ;
             eqn
  equations  eqn+1 ;
             ...... ;
             eqm ;
end;
```

Figure 1. Embedding Inheritance

3.3 Equations in Modelica

As we already stated, Modelica is primarily an equation-based language in contrast to ordinary programming languages where assignment statements proliferate. Equations are more flexible than assignments since they do not prescribe a certain data flow direction. This is the key to the physical modeling capabilities and increased reuse potential of Modelica classes.

Thinking in equations is a bit unusual for most programmers. In Modelica the following holds:

- Assignment statements in conventional languages are usually represented as equations in Modelica.
- Attribute assignments are represented as equations.
- Connections between objects generate equations.

Equations are more powerful than assignment statements. For example, consider a resistor equation where the resistance `R` multiplied by the current `i` is equal to the voltage `v`:

```
R*i = v;
```

This equation can be used in three ways corresponding to three possible assignment statements: computing the current from the voltage and the resistance, computing the voltage from the resistance and the current, or computing the resistance from the voltage and the current. This is expressed in the following three assignment statements:

```
i := v/R; v := R*i; R := v/i;
```

3.4 Connectors and Connector Classes

Modelica *connectors* are instances of *connector classes*, i.e. classes with the keyword `connector` or classes with the `class` keyword that fulfill the constraints of `connector` restricted classes. Such *connectors* declare variables that are part of the communication *interface* of a component defined by the connectors of that component. Thus, connectors specify the interface for interaction between a component and its surroundings.

For example, class `Pin` is a connector class that can be used to specify the external interface for electrical components that have pins as interaction points.

```
connector Pin
  Voltage       v;
  flow Current  i;
end Pin;
Pin pin; // An instance pin of class Pin
```

3.5 Connections

Connections between components can be established between connectors of equivalent type. Modelica supports equation-based acausal connections, which means that connections are realized as equations. For acausal connections, the direction of data flow in the connection need not be known. Additionally, causal connections can be established by connecting a connector with an `input` attribute to a connector declared as `output`.

Two types of coupling can be established by connections depending on whether the variables in the connected connectors are non-flow (default), or declared using the prefix `flow`:

- Equality coupling, for non-flow variables,
- Sum-to-zero coupling, for flow variables.

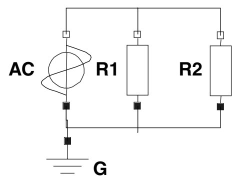
For example, the keyword `flow` for the variable `i` of type `Current` in the `Pin` connector class indicates that all currents in connected pins are summed to zero, according to Kirchhoff's current law.

Connection statements are used to connect instances of connection classes. A connection statement `connect(pin1,pin2)` with `pin1` and `pin2` of connector class `Pin`, connects the two pins so that they form one node. This produces two equations, namely:

```
pin1.v = pin2.v
pin1.i + pin2.i = 0
```

4. Simple Electrical Circuit Model

As a motivational example, we can easily construct a simple electrical circuit model by connecting two resistors components in parallel with a voltage source as is shown on the left part of Figure 2.



```
model TwoPin
   Pin p,n;
   Real v,i;
equation
   v=p.v-n.v;
   0=p.i+n.i;
   i=p.i;
   i=10;
end TwoPin
```

Figure 2. Electrical circuit with two resistors in parallel with an additional equation introduced in the `TwoPin` component

The corresponding Modelica source code is given below:

```
conector Pin
   Real v;
   flow Real i;
end Pin.

model TwoPin
   Pin p,n;
   Real v,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 VcourceAC extends TwoPin;
   parameter Real VA=220;
   parameter Real f=50;
   protected constant Real PI=3.14;
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),R2(R=20);
   VsourceAC AC; Ground G;
equation
   connect(AC.p,R1.p);connect(R1.n,AC.n);
   connect(R1.p,R2.p);connect(R1.n,R2.n);
   connect(AC.n ,G.p);
end Circuit;
```

The model described above constitutes a valid simulation model, which can be compiled, linked together to a numerical solver, and executed. Now we are introducing an error in the source code by adding an additional over-constraining equation (`i=10`) in the model definition of the `TwoPin` component as is shown on the right part of Figure 2. This extra equation will be inherited by all the components that extend the `TwoPin` component. Therefore each instance of the `Resistor` and `VsourceAC` models will contribute to one extra over-constraining equation to the final flattened system of equations.

5. Bipartite Graph Representation of the Intermediate Code

Definition 1: A bipartite graph is an ordered triple $G = (V_1, V_2, E)$ such that V_1 and V_2 are sets, $V_1 \cap V_2 = \emptyset$ and $E \subseteq \{\{x, y\}; x \in V_1, y \in V_2\}$. The vertices of G are elements of $V_1 \cup V_2$. The edges of G are elements of E.

The associated bipartite graph to the flattened intermediate code is obtained by associating to V_1 the set of equations and to V_2 the sets of unknown variables. An edge between $eq \in V_1$ and $var \in V_2$ means that variable var appears in the corresponding equation eq.

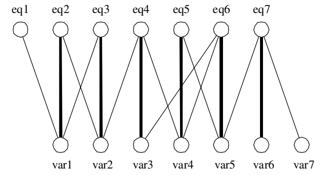
Definition 2: We call a *matching* a set of edges from graph G if no two edges have a common end vertex. A matching M of a graph G is called *maximum matching* if it is a matching with the largest possible number of edges.

Definition 3: A vertex *v* is *saturated* or *covered* by a matching M if some edge of M is incident with *v*. An unsaturated vertex is called a *free* vertex.

Definition 4: A *perfect matching* P is a matching in graph G that covers all its vertices.

Definition 5: A path $P = \{v_0, v_1, \cdots, v_k\}$ in a graph G is called an *alternating path* of M if contains alternating free and covered edges. We use the following notation $u \xrightarrow{=} v$ for an alternating path from vertex *u* to vertex *v*. The notation used to represent an alternating path can be extended to the following notation $P = \{(u_1, v_1), (v_1, u_2), (u_2, v_3) \cdots (v_k, u_{k+1})\}$. A simple path from vertex *u* to vertex *v* is denoted with the help of the following notation $u \xrightarrow{*} v$.

Let us consider a system of linear equations and the associated bipartite graph presented in Figure 3. A possible maximum matching M is represented by the thicker edges.



$$\begin{cases} f(\text{var}_1) = 0 \\ f(\text{var}_1, \text{var}_2) = 0 \\ f(\text{var}_1, \text{var}_2) = 0 \\ f(\text{var}_2, \text{var}_3, \text{var}_4) = 0 \\ f(\text{var}_4, \text{var}_5) = 0 \\ f(\text{var}_3, \text{var}_4, \text{var}_5) = 0 \\ f(\text{var}_5, \text{var}_6, \text{var}_7) = 0 \end{cases}$$

Figure 3. A simple equation system and the associated bipartite graph.

At the next step in our analysis, we are exchanging all the edges that are included in the matching M with bidirectional edges and orienting all other edges from equation nodes to the variable nodes. The following graph, depicted in Figure 4, is obtained:

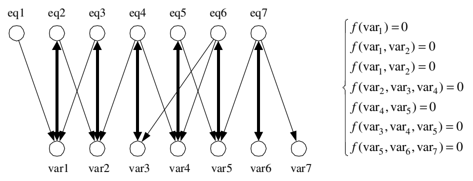


Figure 4. Oriented bipartite graph

Starting from the equation nodes that are not covered by the matching we compute the set of all nodes that are reachable from the free nodes and isolate the obtained subgraph. For the free variable nodes we compute the set of all ancestors that sinks into the free node and isolate the graph. Performing these steps we obtain the graph decomposition shown in Figure 5.

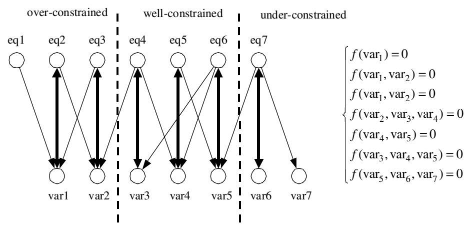


Figure 5. Canonical bipartite graph decomposition

The algorithm is due to Dulmage and Mendelsohn [2] and canonically decompose any maximum matching of a bipartite graph in three distinct parts: over-constrained, under-constrained and well-constrained part.

The *over-constrained* part: the number of equations in the system is greater than the number of variables. The additional equations are either redundant or contradictory and thus yield no solution. We refer to the over-constrained graph using the notation O_G^{k+} where k represents the number of free vertices in the graph

The *under-constrained part*: the number of variables in the system is greater than the number of equations. A possible error fixing strategy would be to initialize some of the variables in order to obtain a well-constrained part or add additional equations to the system.

The *well-constrained part*: the number of equations in the system is equal to the number of variables and therefore the mathematical system is structurally sound having a finite number of solutions. This part can be further decomposed in smaller solution subsets. A failure in decomposing the well-constrained part in smaller subsets means that this part cannot be decomposed and has to be solved as it is. A failure in numerically solving the well-constrained part means that no valid solution exists and there is somewhere a numerical redundancy in the system.

6. Over-Constrained Subgraph

During the model translation the corresponding flattened set of equations to the simulation model from Figure 2 is derived from the original source code (shown in Table 1) and the associated bipartite graph G is constructed.

Table 1. The flat form of the equations corresponding to the over-constrained electrical circuit model.

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==10`*	***var4***	`R1.n.i`
eq5	*`R1.i R1.R==R1.v`*	***var5***	`R1.v`
eq6	*`R2.v==-R2.n.v+R2.p.v`*	***var6***	`R1.i`
eq7	*`0==R2.n.i+R2.p.i`*	***var7***	`R2.p.v`
eq8	*`R2.i==R2.p.i`*	***var8***	`R2.p.i`
eq9	*`R2.i==10`*	***var9***	`R2.n.v`
eq10	*`R2.i*R2.R==R2.v`*	***var10***	`R2.n.i`
eq11	*`AC.v==-AC.n.v+AC.p.v`*	***var11***	`R2.v`
eq12	*`0==AC.n.i+AC.p.i`*	***var12***	`R2.i`
eq13	*`AC.i==AC.p.i`*	***var13***	`AC.p.v`
eq14	*`AC.i==10`*	***var14***	`AC.p.i`
eq15	*`AC.v==AC.VA* sin[2*time*AC.f*AC.PI]`*	***var15***	`AC.n.v`
eq16	*`G.p.v==0`*	***var16***	`AC.n.i`
eq17	*`AC.p.v==R1.p.v`*	***var17***	`AC.v`
eq18	*`R1.p.v==R2.p.v`*	***var18***	`AC.i`
eq19	*`AC.p.i+R1.p.i+R2.p.i==0`*	***var19***	`G.p.v`
eq20	*`R1.n.v==R2.n.v`*	***var20***	`G.p.i`
eq21	*`R2.n.v==AC.n.v`*		
eq22	*`AC.n.v==G.p.v`*		
eq23	*`AC.n.i+G.p.i+ R1.n.i+R2.n.i==0`*		

Choosing an arbitrary maximum cardinality matching and performing a D&M canonical decomposition the over-constrained subgraph is isolated and represented graphically in Figure 6. The maximum cardinality matching for the corresponding bipartite graph will let three vertices uncovered corresponding to the equations *eq9, eq18* and *eq17.*

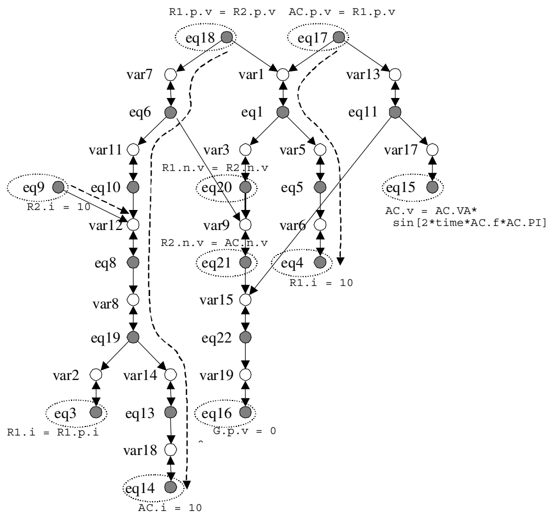


Figure 6. The over constrained directed graph O_G^{3+}.

Following the sets of the D&M canonical decomposition, by starting from the free vertices, all of the descendants are computed. Each free vertex induces an over-constrained subgraph. Three equations need to be eliminated from the over-constrained subgraph in order to make the systems well-constrained. One equation needs to be eliminated from each over-constraining component $O_{1G}^{1+}, O_{2G}^{1+}, O_{2G}^{1+}$ (see Figure 7). Only those equations can be removed from O_G^{3+} that are not covered by an associated maximum matching. But the maximum matching associated to the graph G is not unique and a new matching with the same cardinality can be obtained by exchanging matching edges with non-matching edges along alternating paths. An alternating path is a path where matching edges and non-matching edges are alternating. For example, we can obtain a new maximum cardinality matching exchanging the edges along the path $\{eq9, var12, eq8, var8, eq19, var2, eq3\}$ or denoted shortly $eq9 \xrightarrow{=} eq3$. We use the notation $\xrightarrow{=}$ to denote an alternating path. In that way *eq3* can be made into a free vertex and can be considered for elimination.

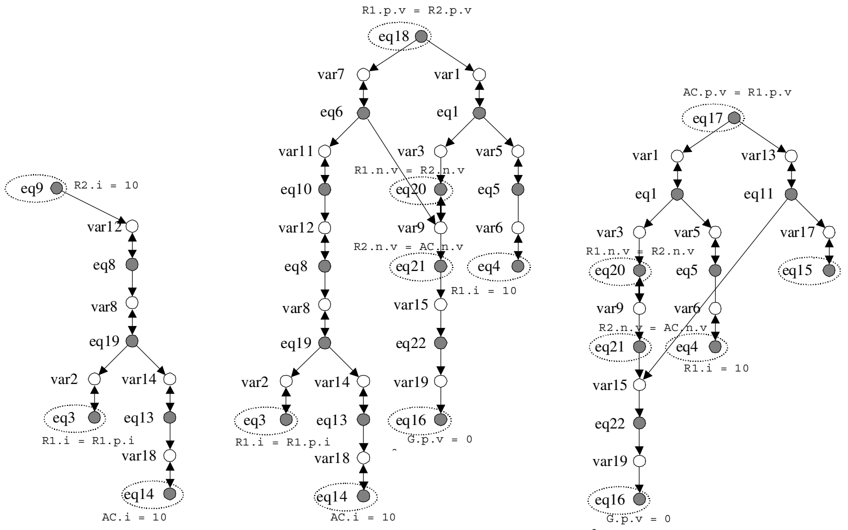


Figure 7. The $O_{1G}^{1+}, O_{2G}^{1+}, O_{2G}^{1+}$ components of the O_G^{3+} over-constrained subgraph

The following equations are included in each over-constrained subcomponent:

$\{eq9, eq8, eq19, eq3, eq13, eq14\} \in \mathcal{V}(O_{1G}^{1+})$

$\{eq18, eq6, eq10, eq9, eq8, eq19, eq3, eq13, eq14, eq1, eq20,$

$eq21, eq22, eq16, eq5, eq4\} \in \mathcal{V}(O_{2G}^{1+})$

$\{eq17, eq1, eq20, eq21, eq22, eq16, eq5, eq4, eq11, eq15\} \in \mathcal{V}(O_{3G}^{1+})$

Analyzing some of the equation nodes from the previous lists, we can see that by eliminating those, the remaining graph becomes disconnected, containing two independent components, which are not linked together by edges. This situation is not very common from the modeling point of view. Those nodes, which by elimination disconnect the underlying bipartite graph, can be safely removed from the lists. We obtain the following reduced equation lists called the *safe equations* lists:

$\{eq9, eq3, eq14\} \in \mathcal{V}(O_{1G}^{1+})$

$\{eq18, eq9, eq3, eq14, eq20, eq21, eq16, eq4\} \in \mathcal{V}(O_{2G}^{1+})$

$\{eq17, eq20, eq21, eq16, eq4, eq15\} \in \mathcal{V}(O_{3G}^{1+})$

Without any premises, all the possible combinations that can be scheduled for elimination are represented by the following graph (see Figure 8) where on the top we have the equations which are included in O_{1G}^{1+}, in the middle equation nodes from O_{2G}^{1+}, and at the bottom equations from O_{3G}^{1+}. If three nodes (one from each subset) are linked together with edges, this means that it constitutes a valid set that can be considered for elimination.

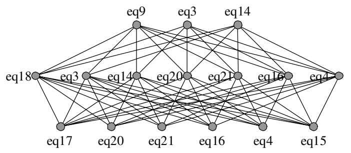


Figure 8. The possible elimination combinations

It should be noted that some equations appear in more than one equation list. For example, *eq3* appears in the safe equation list associated to the subcomponent O_{1G}^{1+} and O_{2G}^{1+}. This means that *eq3* can be made a free vertex by exchanging matching edges with non-matching edges along the path $eq9 \xrightarrow{=} eq3 \in O_{1G}^{1+}$ or $eq18 \xrightarrow{=} eq3 \in O_{2G}^{1+}$. If *eq3* is scheduled for elimination in the subgraph O_{1G}^{1+} it cannot be scheduled again for elimination in the subgraph O_{2G}^{1+}, even it is present in the list of equations associated to the subgraph. Moreover, if *eq13* is scheduled for elimination in subgraph O_{1G}^{1+} it will also affect the scheduling for elimination of the node *eq14* in subgraph O_{2G}^{1+}. The operation of exchanging the non-matching edges with matching edges along the path $eq9 \xrightarrow{=} eq3 \in O_{1G}^{1+}$ will affect the path $eq18 \xrightarrow{=} eq14 \in O_{2G}^{1+}$ isolating *eq14.* For this reason *eq14* cannot be selected any more for elimination in O_{2G}^{1+} even if previously had a valid path to the free node *eq18*.

Therefore a mechanism to quickly check if certain equation subsets can constitute a safe removal set is needed. The next section introduces a special graph structure, which captures the dependencies among the equations and also gives an algorithm that quickly checks the validity of certain equation subsets

7. The Alternating Paths Dependency Graphs

In order to illustrate the path selection algorithm and to show the dependencies among the over-constraining variables the list of safe over-constraining equations associated to each over-constrained subgraph is expanded to list of paths where each equation element is replaced by the path from the free variable to itself. The safe over-constraining equation lists are transformed into:

$$\{eq9 \xrightarrow{*} eq9, eq9 \xrightarrow{=} eq3, eq9 \xrightarrow{=} eq14\} \qquad \text{for } O_{1G}^{1+}$$

$$\{eq18 \xrightarrow{*} eq18, eq18 \xrightarrow{=} eq3, eq18 \xrightarrow{=} eq14, eq18 \xrightarrow{=} eq20, eq18 \xrightarrow{=} eq21, eq18 \xrightarrow{=} eq16, eq18 \xrightarrow{=} eq4\} \qquad \text{for } O_{2G}^{1+}$$

$$\{eq17 \xrightarrow{*} eq17, eq17 \xrightarrow{=} eq20, eq17 \xrightarrow{=} eq21, eq17 \xrightarrow{=} eq216, eq17 \xrightarrow{=} eq4, eq17 \xrightarrow{=} eq15\} \qquad \text{for } O_{3G}^{1+}$$

Each path can be further extended by including the cut variable node in the path. A cut variable node is the first shared variable node along two considered paths. For example, considering the paths $eq9 \xrightarrow{=} eq3 \in O_{1G}^{1+}$ and $eq18 \xrightarrow{=} eq14 \in O_{2G}^{1+}$ the first common variable node which is included in both paths is *var12*. Including the cut variable node the path $eq9 \xrightarrow{=} eq3$ becomes $eq9 \xrightarrow{=} \text{var}\,12 \xrightarrow{=} eq13$. The list of all cut variables nodes $\{var12, var1, var9, var5\}$ associated to O_G^{3+} can be easily computed and used to expand the path lists.

The graphical representation of the paths with the cut variable nodes is given in Figure 9 where we have inverted all the edges. An edge starting from a free equation and pointing to itself denotes a free variable.

Any further computations involving the equation node elimination can now be performed on the shortened representation of the alternating paths. Let us illustrate the reasoning performed on the above mentioned graph representation by choosing, for example, equation node *eq3* from the set of safe nodes included in O_{1G}^{1+}. In order to safely eliminate *eq3*, the matching edges need to be exchanged with non-matching edges along the path $eq9 \xrightarrow{=} eq3 \in O_{1G}^{1+}$.

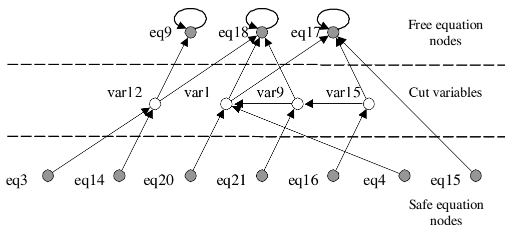


Figure 9. Shortened representation of the alternating paths.

By performing the exchange of edges, all the paths that have common edges with the modified path are affected, making the safe nodes unreachable from the free equation node. Therefore all the edges adjacent to *eq3* can be eliminated. Then, we follow the chosen path and reach the cutting variable *var12.* All the associated edges adjacent to node *var12* will also be eliminated from the path graph. We are moving forward on the chosen path, one more time, and reach the free equation node. All the adjacent edges of the free equation node are also removed. An edge starting from *eq3* and pointing to itself can be drawn, indicating in that way that equation node *eq3* has been made into a free node instead of equation node *eq9,* as is shown in Figure 10 a). It should be noted that the safe equation node *eq14* becomes isolated and cannot reach a free nodes.

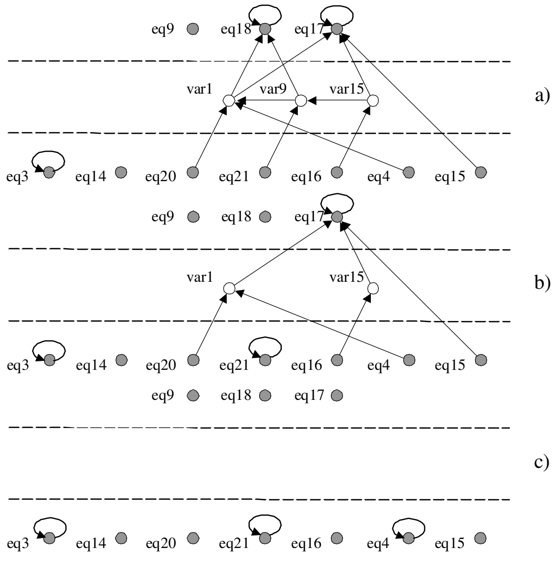


Figure 10. Transformation of the shortened alternating path graph by choosing **a)** *eq3* **b)** *eq3* and *eq21* **c)** *eq3*, *eq21* and *eq4* for elimination

In the next step another equation node need to be selected from the safe equation lists associated to the second over constrained subgraph O_{2G}^{1+}. We can pick any of the equations present in the list $\{eq18, eq9, eq3, eq14, eq20, eq21, eq16\} \in \mathcal{V}(O_{1G}^{1+})$, with a few restrictions introduced by the previous equation node choice. Equation node *eq13* has already been selected for elimination in subgraph O_{1G}^{1+} and *eq14* cannot reach a free node anymore. Let us choose *eq21* along the path $eq18 \xrightarrow{=} \text{var}\,9 \xrightarrow{=} eq21$. All the adjacent edges to equation *eq21 var9* and *eq18* are now removed from the graph together with the cut variable node *var9,* and *eq21* is marked as a free node.

Moving to the next over-constrained subgraph O_{3G}^{1+} we may choose *eq4* for elimination which is available along the path $eq17 \xrightarrow{=} var1 \xrightarrow{=} eq4$ as is depicted in Figure 10 b). Following the alternate path and eliminating the extra edges and the cut variable node, we obtain finally the graph shown in Figure 10 c). The above chosen equation nodes can be safely removed from the underlying system of equations because any of them have a valid path to a free equation node. By exchanging the matching edges with non-matching edges along those paths each of the chosen equation can be made into a free equation node.

The shortened alternating path graph is useful to quickly check if a given subset of equations, chosen by the user or automatically chosen by the debugger, is eliminated from the overall system of equation will lead to a remaining well-constrained system of equations. In conclusion the recursive algorithm Algorithm 1 (CHKOC) can be given to automatically check the validity of an equation subset.

The *optimize* function called in the *chhoc* procedure will eliminate all the paths from the graph that are not terminated with a free equation node. The *DFSm* function performs a depth first search on an input graph starting from a given node and return the list of traversed paths.

The lists of equations are traversed and each equation is checked if has a valid path to a free node. If a valid path for an equation is found the belonging nodes of the path are eliminated from the graph and the procedure *chhoc* is called recursively having as parameters the reduced graph and the list free equations from where the already checked equation was eliminated. If the next equation node does not have a valid path to a free node, then the search returns to the equation node checked just before and a new path is considered. The general step is repeated until every path associated to that equation node has been checked. At each general step, if a valid path is found the equation node is automatically added to a global list *L*. At the end of the procedure the list *L* contains the maximum number of equation nodes that have a valid path to a free equation node. If the list *L* is the same as the input list then the combination of equations nodes from the input list constitutes a valid set and can be safely removed from the over-constrained system of equations in order to make the system consistent.

Algorithm 1 CHKOC(SO_G^{k+} , $\{eq_1, eq_2 \ldots eq_k\}$)

Checks if a subset of equation constitutes a valid elimination set from the flattened set of equations

Input Data: The shortened alternating path graph SO_G^{k+} and subset of equations $\{eq_1, eq_2 \ldots eq_k\}$

Result: a Boolean value: true if the subset of equations constitutes a valid elimination set, false otherwise.

```
begin:
  Procedure chkoc(G, eqList)
  begin
      if (size(eqList) != 0) then
         $eq_k$ = pop_last(eqList);
         pathList := DFSm(G, $eq_k$);
         optimize(pathList);
         for all $p' \in pathList$ do
            for all nodes $v \in p'$ do
               delete_node_and_adj_edges$(G, v)$;
            end for;
            if $eq_k \not\subset L$ then L := append(L, { $eq_k$ });
            chkoc (G,eqList);
            restore_nodes_and_adj_edges$(G)$;
         end for;
         push_back(eqList ,eqk);
      endif;
   end procedure.
  $L = \varnothing$;
  chkoc( $SO_G^{k+}$ , $\{eq_1, eq_2 \ldots eq_k\}$ )
  if (L == $\{eq_1, eq_2 \ldots eq_k\}$ ) then return true;
  else return false;
end.
```

By checking each possible combination using Algorithm 1 the graph from Figure 8 can be reduced to the graph depicted in Figure 11 that still has a high complexity.

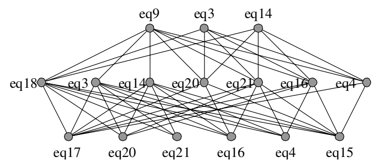


Figure 11. Reduced safe nodes combinations graph

8. Guided Filtering by Semantic Rules

Applying Algorithm 1 and taking into account the structural information regarding the over-constrained subgraph the number of possible combinations of safe nodes have been reduced. There are still too many combinations of safe equations. Presenting them to the user at this stage is not very useful. These combinations represent the error fixing solutions at the flattened intermediate code level. However the user has only the possibility of making modifications at the source code level. Therefore a mechanism to validate the combinatorial solutions from the modeling language point of view is needed.

Let us consider the mapping between the original Modelica source code of the simple electrical circuit model and the generated intermediate form of the flattened equation shown in Figure 12. Only those statements, from the original Modelica code, that directly generate the intermediate form are shown. Model definitions, variable declaration and other irrelevant language construct were intentionally eliminated. The correspondence graph between the original code statement and the flattened intermediate form can be constructed during the translation phase.

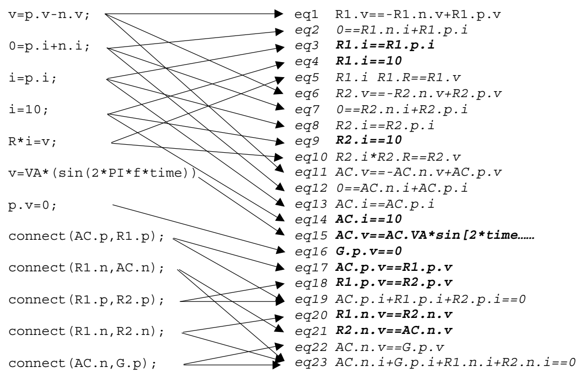


Figure 12. Correspondence between the Modelica source code statements and the corresponding flattened equations

The safe equations which are also present in the shortened alternating path graph are indicated in Figure 12 by bold letters. The correspondence graph from Figure 12 can be simplified by removing those original code statements that generates intermediate equations that are outside the set of safe equations and we obtain the correspondence graph from Figure 13. We have kept only those original source code statements that generate at least one equation that can be found among the component equation nodes of the shortened alternating graph.

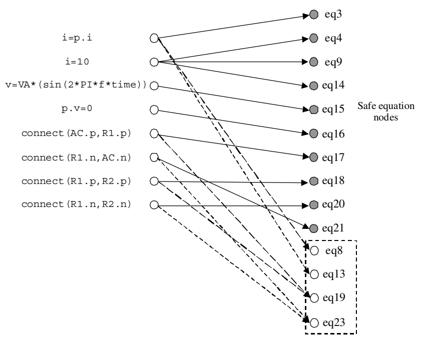


Figure 13. Reduced correspondence graph between the Modelica code and the corresponding flattened equations.

Let us analyze the equations: *eq4, eq9*, and *eq14*. They were generated by inheritance from the original equation `i=10,` in the `TwoPin` component. By eliminating the original source code statement all the three equations will be removed from the flattened intermediate form. Any attempt at eliminating only one of the equations from the intermediate form by removing a statement from the original source code, will fail. In conclusion, all the adjacent edges of nodes that represent *eq3, eq9*, or *eq14* in the graph from Figure 11 that includes a node, which is not among these nodes, can be safely removed.

Equation `i=p.i` will generate three equations *eq3, eq8* and *eq13* in the intermediate code. Only *eq13* is among the equation nodes that need to be eliminated. It should be noted that the elimination of *eq3* is only possible by removing `i=p.i`. However, the removal of `i=p.i` will trigger the elimination of two additional equations which are not members of the safe equation list. Therefore *eq3* cannot be considered for elimination and all the adjacent edges representing *eq3* in the graph from Figure 11 can be removed. For the same reason all the adjacent edges to *eq17*, *eq18, eq20, eq21* can also be removed from the graph.

After performing all the simplifications we obtain the graph from Figure 14 that contains only edges linking the *eq4, eq9*, and *eq14* equation nodes. It means that the user can eliminate the statement `i=10` from the `TwoPin` component in the original source code. This statement was exactly the additional statement introduced at beginning of this analysis in order to over-constrain the simulation model.

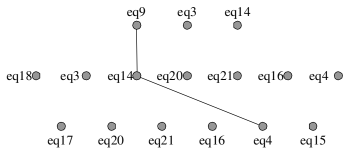


Figure 14. The simplified graph denoting the possible equation node combinations that can be scheduled for elimination.

The same over-constraining final effect, and the same form of the flattened intermediate code can be achieved by over-constraining each class derived from the `TwoPin` instead of over-constraining the parent component itself with an extra equation. In conclusion the classes `Resistor, VsourceAC` will get an extra equation `i=10,` each. The generated flattened form of the equations will be the same. Reasoning based on the semantic transformation rules, we obtain the following combination graph, depicted in Figure 15, with the additional constraint that *eq9* and *eq4* should be selected together (they cannot appear independently in the selection set).

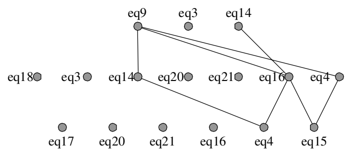


Figure 15. Possible combinations of equation nodes scheduled for elimination when the `Resistor` and `VsourceAC` component are over constrained

The following equation sets can be selected for elimination and presented to the user for validation:

- *{eq9,eq14,eq4}* by eliminating `i=10` from the `Resistor` component and `i=10` from the `VsourceAC` component.
- *{eq9,eq16,eq4}* by eliminating `i=10` from the `Resistor` component and `p.v=0` from the `Ground` component.
- *{eq9,eq4,eq15}* by eliminating `i=10` from the `Resitor` component and `v=VA*sin(2*PI*f*time)` from the `VsourceAC` component.
- *{eq14,eq16,eq15}* by eliminating `i=10` and `v=VA*sin(2*PI*f*time)` from the `VsourceAC` component, `p.v=0` from the `Ground` component.

9. Conclusions

Determining the cause of errors in models of physical systems is hampered by the limitations of the current techniques of debugging declarative equation based languages. We have presented a new approach for debugging declarative equation based languages by employing graph decomposition techniques and have given several usage examples for debugging over-constrained models. It has also been demonstrated that it is possible to create a tool with an enhanced user interaction capability that explicitly can be used in debugging and understanding complicated simulation models.

AMOEBA (Automatic MOdelica Equation Based Analyzer) is a debugging tool that we have designed, implemented and attached to existing Modelica based simulation environment. The tool is able to successfully detect and provide error-fixing solutions for typical over and under-constrained situations, which appear during the modeling stage using Modelica. These kinds of bugs are usually the most frequent error situations encountered when programming with declarative equation based language. Most of the wrong specification at the equations level will fall into one of these categories. The debugging process concentrates on finding structural inconsistencies at the intermediate code level by using graph decomposition algorithms, similar to those presented in the paper, and then mapping the error back to the original source code in order to provide meaningful error messages to the user. Whenever is possible the original source code is manipulated in such a way that the simulation model becomes consistent. Our prototype debugger shows that static analysis can considerably enhance the error finding process when modeling with equation based languages as well as improving the designer's capability to deal with the increasing complexity of today's system simulation models described by such languages.

The merits of the proposed debugging technique are as follows:

- The user is exposed to the original source code of the program and is therefore not burdened with understanding the intermediate code or the numerical artifacts for solving the underlying system of equations.
- The user has a greater confidence in the correctness of the simulation model.
- It statically detects a broad range of errors without having to execute the simulation model.

References

[1] Abadi M. and L. Cardelli, *A Theory of Objects*, Springer Verlag, ISBN 0-387-94775-2, 1996.

[2] Dulmage, A.L., Mendelsohn, N.S. *Coverings of bipartite graphs,* Canadian J. Math. 10, 517-534.

[3] Elmqvist, H.; S. E. Mattsson and M. Otter. 1999. “Modelica - A Language for Physical System Modeling, Visualization and Interaction.” In *Proceedings of the 1999 IEEE Symposium on Computer-Aided Control System Design* (Hawaii, Aug. 22-27).

[4] Fritzson, P.; P. Bunus ”Modelica, a general Object-Oriented Language for Continuous and Discrete-Event System Modeling and Simulation." In *Proceedings of the 35th Annual Simulation Symposium* (San Diego, California, April 14-18, 2002)

[5] Fritzson, P. "Introduction to Modelica". First chapter of book draft. www.ida.liu.se/~pelab/modelica

[6] Fritzson P and V. Engelson. “Modelica, A Unified Object-Oriented Language for System Modeling and Simulation.” In *Proceedings the 12th European Conference on Object-Oriented Programming).* (Brussels, Jul. 20-24,1998).

[7] Modelica Association. *Modelica – A Unified Object-Oriented Language for Physical Systems Modeling - Tutorial and Design Rationale Version 1.4* (December 15, 2000). http://www.modelica.org

[8] Modelica Association. Modelica – *A Unified Object-Oriented Language for Physical Systems Modeling – Language Specification Version 2.0.* (July 10, 2002). http://www.modelica.org

[9] Tiller M. *Introduction to Physical Modeling with Modelica*. Kluwer Academic Publishers, 2001.