

# Application of Graph Decomposition Techniques to Debugging 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 static analysis for debugging purposes of programs written in declarative equation based modeling languages. At the same time, we examine the particular debugging problems posed by Modelica, a declarative equation based modeling language. We also present our view of the issues and solutions based on a proposed framework for debugging declarative equation based languages. Program analysis solutions for program understanding and for static debugging of declarative equation based languages, based on bipartite graph decomposition, are presented in the paper. We also present an efficient way to annotate the underlying equations in order to help the implemented debugger to eliminate the heuristics involved in choosing the right error fixing solution. This also provides means to report the location of an error caught by the static analyzer or by the numeric solver, consistent with the user's perception of the source code and simulation model. A prototype debugger is implemented.*

## 1. Introduction

Simulation models are increasingly being used in problem solving and in decision making since engineers need to analyze increasingly complex and heterogeneous physical systems. In order to support mathematical modeling and simulation, 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.

Equation based declarative programming presents new challenges in the design of programming environments. In order for declarative equation based modeling languages to achieve widespread acceptance, associated programming environments and development tools must become more accessible to the user. A significant part of the simulation design effort is spent on detecting deviations from the specifications and subsequently localizing the sources of such errors. Employment of debugging environments that control the correctness of the developed source code has been an important factor in reducing the time and cost of software development in classical programming languages. Currently, few or no tools are available to assist developers debugging declarative equation based modeling languages. Since these languages usually are based on object-orientation and acausal physical modeling, traditional approaches to debugging are inadequate and inappropriate to solve the error location problem. To begin to address this need, we propose a methodology for implementing an efficient debugging framework for high level declarative equation based languages, by adapting graph decomposition techniques for reasoning about the underlying systems of equations. Detecting anomalies in the source code without actually solving the underlying system of equations provides a significant advantage: the modeling error can be corrected before embarking on a computationally expensive numerical solution process provided by a numerical solver. Also the errors detected by the numerical solvers are not reported in a way which is consistent with the user perception of the declarative source code.

Traditional static checkers detect type errors and simple anomalies such as obviously uninitialized variables. These static checkers are frequently used and usually integrated in the programming language compiler, the static checking being a distinct phase in the compilation phase. In the case of declarative equation based languages where underlying constraints expressed by equations give the behavior of the simulation model, the functionality of the static checkers need to be extended in order to successfully handle *structural diagnosis* problems of the underlying equations and constraints.

The rest of the paper is organized as follows: Section 2 gives some explanations why is hard to debug declarative equation based languages. In Section 3 a simple simulation model together with the underlying declarative specification and source code written in the Modelica language is presented. Then, we present several graph decomposition techniques and our algorithmic debugging approach based on

those techniques. Section 5 provides some details about the structures used to annotate the underlying equations of the simulation model, in order to help the debugger to eliminate the heuristics when multiple choices are available to fix an error. In Section 6 we explain the debugging of an over-constrained system. Implementation details of the debugger are given in Section 7. Finally, Section 8 concludes and summarizes the work.

## 2. Arising Difficulties When Debugging Declarative Equation Based Languages.

The application of algorithmic debugging techniques [20] and generalized algorithmic debugging techniques [10] to the evaluation of structural procedural languages is an approach which has found increased applicability over the past years. However, traditional approaches to debugging are inadequate and inappropriate to solve the error location problem in declarative equation based languages. The fundamental problem is that conventional debuggers and debugging techniques are based on observation of execution events as they occur.

Even nontraditional declarative debugging techniques such as the above-mentioned algorithmic debugging method, are inadequate for equation-based languages. In order to see this, consider invoking an algorithmic program debugger [20] on a functional/logic program after noticing an external symptom of a bug. The debugger executes the program and builds a trace execution tree at the function level while saving some useful trace information such as function names and input/output parameter values. In other words, the algorithmic debugging method is dependent on the use of functions and function calls in the language. In the case of declarative equation based languages, there is no clear execution tree and the inputs and outputs are not clearly stated inside the model. Therefore using an algorithmic debugging technique is useless.

In conclusion we can take a look on why errors are hard to find in a declarative equation based language:

- There is a cause-effect gap between the time or space when an error occurs and the time or space when the error becomes apparent to the programmer.
- The acausality of the language eliminates the use of program traces as a debugging guide.
- The transformation process from the declarative form to the procedural form looses or obscures a lot of model structure information, which might be useful for debugging purposes.
- Even in a highly structured system, which extensively uses hierarchical and modular simulation models, surprising event may occur because the human mind is not able to fully comprehend the many conditions that can arise mainly because of the interactions of the components.
- Debugging is in some sense harder to perform because much run-time debugging must be replaced by compile-time static checking.
- The static analysis is mostly global and it is necessary to consider the whole program.

## 3. Simulation Model Example

Obviously, each simulation problem is associated with a corresponding mathematical model. In dynamic continuous simulation the mathematical model is usually represented by a mixed set of algebraic equations and ordinary differential equations. For some complicated simulation problem the model can be represented by a mixed set of ordinary differential equations (ODEs), differential algebraic equations (DAEs) and partial differential equations (PDEs). Simulation models can become quite large and very complex in their structure sometimes involving several thousand equations. An efficient way of specifying a simulation model is to employ a modeling language.

Modelica is a new language for hierarchical object-oriented physical modeling which is developed through an international effort [9][6]. The language unifies and generalizes previous object-oriented modeling languages. The reader of the paper is referred to [17][18] and [22] 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. Those interested in shorter overviews of the language may wish to consult [9] or [6].

In Fig.1. the Modelica source code of a simple simulation model consisting of a resistor connected in parallel to sinusoidal voltage is given. The intermediate form is also given for explanatory purposes. The Circuit model is represented as an aggregation of the Resistor, Source and Ground submodels connected together by means of physical ports. The system of equations describing the overall model is obtained by merging the equations of all simple models and all binding equations generated by the connect statements.

<pre> 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; </pre>	<pre> Flat equations 1. R1.v == -R1.n.v + R1.p.v 2. 0 == R1.n.i + R1.p.i 3. R1.i == R1.p.i 4. R1.i*R1.R == R1.v 5. AC.v == -AC.n.v + AC.p.v 5. 0 == AC.n.i + AC.p.i 7. AC.i == AC.p.i 8. AC.v == AC.VA*Sin[2*time*AC.f*AC.PI] 9. G.p.v == 0 10. AC.p.v == R1.p.v 11. AC.p.i + R1.p.i == 0 12. R1.n.v == AC.n.v 13. AC.n.v == G.p.v 14. AC.n.i + G.p.i + R1.n.i == 0  Flat Variables 1. R1.p.v  2. R1.p.i  3. R1.n.v 4. R1.n.i  5. R1.v   6. R1.i 7. AC.p.v  8. AC.p.i  9. AC.n.v 10. AC.n.i 11. AC.v   12. AC.i 13. G.p.v  14. G.p.i  Flat Parameters R1.R -&gt; 10 AC.VA -&gt; 220 AC.f -&gt; 50  Flat Constants AC.PI -&gt; 3.14159 </pre>
--	--

**Fig. 1.** Modelica source code of a simple simulation model and the corresponding flattened systems of equation, variables, parameters and constants.

#### 4. Graph Based Representation of the Underlying Model

Many practical problems form a model of interaction between two different types of objects and can be phrased in terms of problems on bipartite graphs. The expressiveness of the bipartite graphs in concrete practical applications has been demonstrated many times in the literature [4][2]. We will show that the bipartite graph representations are general enough to efficiently accommodate several numeric analysis methods in order to reason about the solvability and unsolvability of the flattened system of equations and implicitly about the simulation model behavior. Another advantage of using the bipartite graphs is that it offers an efficient abstraction necessary for program transformation visualization when the equation based declarative specifications are translated to procedural form.

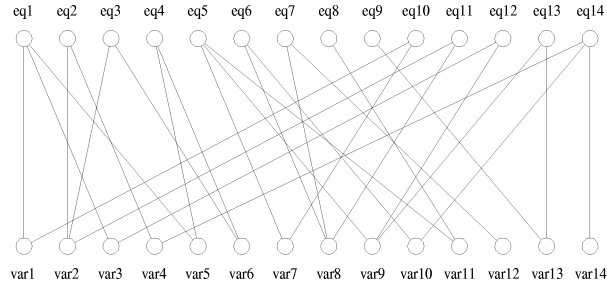
The bipartite graph representation and the associated decomposition techniques are widely used internally by compilers when generating the procedural form from the declarative equation based description of the simulation model [7] [16] but none of the existing simulation systems use them for debugging purposes or expose them visually for program understanding purposes. Our debugging approach follows the same philosophy as does the reduction of constraint systems used for geometric modeling in [1] and [3].

In the remaining of this paragraph it is our intention to give the basic definitions and some of the notation which we shall use throughout the rest of this paper.

**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  $E$ .

**Definition 2:** Let  $G$  be a graph with vertex set  $V(G) = \{v_1, v_2, \dots, v_p\}$  and edge set  $E(G) = \{e_1, e_2, \dots, e_q\}$ . The incidence matrix of  $G$  is the  $p \times q$  matrix  $M(G) = [m_{ij}]$ , where  $m_{ij}$  is 1 if the edge  $e_{ij}$  is incident with vertex  $v_i$  and 0 otherwise.

We consider the bipartite graph associated to a given system of equations resulting from the flattening operation of declarative specification. Let be  $V_1$  the set of equations and  $V_2$  the set of variables representing unknowns. An edge between  $eq \in V_1$  and  $var \in V_2$  means that the variable  $var$  appears in the corresponding equation  $eq$ . Based on this rule the associated bipartite graph of the flattened system of equation from Fig. 1 is presented in Fig. 2.



**Fig. 2.** The associated bipartite graph of the simple circuit model from Fig. 1

#### 4.1 Bipartite Matching Algorithms.

**Definition 3:** A **matching** is a set of edges from graph  $G$  where no two edges have a common end vertex.

**Definition 4:** A **maximum matching** is the matching with the largest possible number of edges.

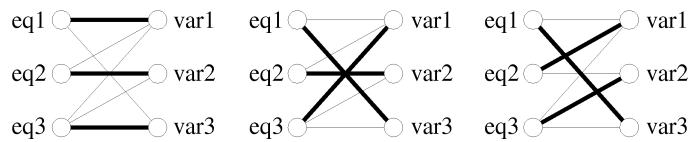
**Definition 5:** A matching  $M$  of a graph  $G$  is **maximal** if it is not properly contained in any other matching.

**Definition 6:** 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 7:** A **perfect matching**  $P$  is a matching in a graph  $G$  that covers all its vertices.

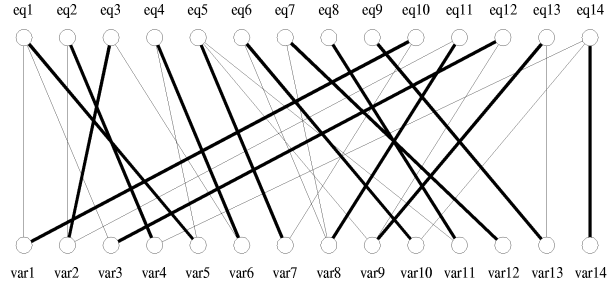
**Definition 8:** A path  $P = \{v_0, v_1, \dots, v_k\}$  in a graph  $G$  is called an **alternating path** of  $M$  if contains alternating free and covered edges.

In Fig. 3 all the possible perfect matchings of a simple bipartite graph are presented. It should be noted that a maximum matching and the perfect matching of a given bipartite graph is not unique.



**Fig. 3.** An example bipartite graph with all the possible perfect matchings marked by thick lines

The associated equation system to a perfect matching is structurally well-constrained and therefore can be further decomposed into smaller blocks and sent to a numerical solver. Fig. 3 illustrates the maximal matching of the associated bipartite graph to the simulation model presented in Fig. 1. It worth noting that in that case the maximal matching is also a perfect matching of the associated bipartite graph.

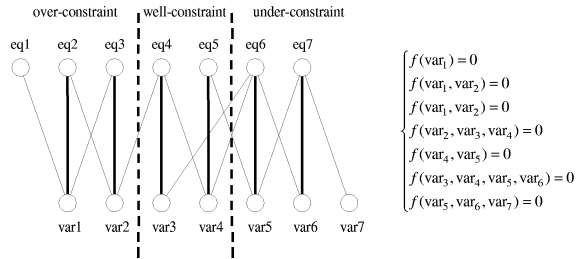


**Fig. 4.** One possible perfect matching of the simulation model associated bipartite graph

From the computational complexity point of view, the best sequential algorithm for finding a maximum matching in bipartite graphs is due to Hopcroft and Karp [13]. The algorithm solves the maximum cardinality matching problem in  $O(n^{5/2})$  time and  $O(nm)$  memory storage where  $n$  is the number of vertices and  $m$  is the number of edges. Algorithms for enumerating all perfect, maximum and maximal matchings in bipartite graphs are also proposed in [11][21]. The enumeration algorithm for all perfect matchings in bipartite graph proposed by Fukuda and Matsui in [11] takes  $O(n^{1/2}m + mN_p)$  time where  $N_p$  is the number of perfect matchings in the given bipartite graph. In [21] improved algorithms for finding and enumerating all perfect, maximal and maximum matchings are presented and it takes only  $O(n)$  time per a matching. We will come back to these algorithms later in the paper when the reduced list of the equivalent over-constraining equations are constructed for a erroneous simulation model.

#### 4.2 Dulmage – Mendelsohn’s Canonical Decomposition.

In this section we shall present a structural decomposition of a bipartite graph associated with a simulation model which relies on the above presented vertex coverings. The algorithm is due to Dulmage and Mendelsohn [5] and canonically decompose any maximum matching of a bipartite graph in three distinct parts: over-constrained, under-constrained, and well-constrained part.



**Fig. 5.** Dulmage Mendelsohn’s canonical decomposition of an associated bipartite graph to an equation system

The algorithm is given as follows:

---

##### **Algorithm 1:** Dulmage and Mendelsohn canonical decomposition

**Input Data:** A bipartite graph  $\mathcal{G}$

**Result:** Three subgraphs: well-constrained  $\mathcal{WG}$ , over-constrained  $\mathcal{OG}$  and under-constrained  $\mathcal{UG}$ .

**begin:**

- Compute the maximum matching  $\mathcal{MG}$  of  $\mathcal{G}$ .
- Compute the directed graph  $\mathcal{G}'$  by replacing each edge in  $\mathcal{MG}$  by two arcs and orienting all other edges from the equations to the variables.
- Let be  $\mathcal{OG}$  the set of all descendants of sources of the directed graph  $\mathcal{G}'$ .
- Let be  $\mathcal{UG}$  the set of all ancestors of sink of the directed graph  $\mathcal{G}'$ .
- Calculate  $\mathcal{WG} = \mathcal{G} - \mathcal{OG} - \mathcal{UG}$ .

**end.**

---

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. A possible error fixing strategy is to remove the additional over-constraining equations from the system in order to make the system well-constrained. Even if the additional equations are *soft constraints* which means that they verify the solution of the equation system and are just redundant equations, they are reported as errors by the debugger because there is no way to verify the equation solution during static analysis without explicitly solving them.

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.

Over and under-constrained situations can coexist in the same model. In the case of over-constrained model, the user would like to remove the over-constraining equations in a manner which is consistent to the original source code specifications, in order to alleviate the model definition.

The *well-constrained* part: the number of equations in the system is equal to the number of variables and therefore the mathematical system of equations is structurally sound having a finite number of solutions. This part can be further decomposed into smaller solution subsets. A failure in decomposing the well-constrained part into 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.

The decomposition captures one of the many possible solutions in which the model can be made consistent. The direct solution proposed by the decomposition sometimes cannot be acceptable from the restriction imposed by the modeling language or by the modeling methodology by itself. Therefore a search through the equivalent solution space needs to be done and, check whether the equivalent solutions are acceptable.

## 5. Equation Annotations

For annotating the equations we use a structure which resembles the one developed in [8]. We define an annotated equation as a record with the following structure: <Equations, Name, Description, No. of associated eqs., Class name, Flexibility level, Connector generated>. The values defined by annotations are later incorporated in the error repair strategies, when heuristics involved in choosing the right option from a series of repair strategies need to be eliminated.

**Table 1.** The structure of the annotated equation

Attribute	Value
Equation	$R1.i * R1.R == R1.v$
Name	"eq4"
Description	"Ohm's Law for the resistor component"
Nr. of associated eq	1
Class Name	"Resistor"
Flexibility Level	3
Connector generated	no

The *Class Name* tells from which class the equation is coming. This annotation is extremely useful in exactly locating the associated class of the equation and therefore providing concise error messages to the user.

The *No. of associated eqs.* parameter specify the number of equations which are specified together with the annotated equation. In the above example the *No. of associated eqs.* is equal to one since there are no additional equations specified in the *Resistor* component. In the case of the *TwoPin* component the number of associated equations is equal to 3. If one associated equation of the component need to be eliminated the value is decremented by 1. If, during debugging, the equation  $R1.i * R1.R == R1.v$  is diagnosed to be an over-constraining equation and therefore need to be eliminated, the elimination is not possible because the model will be invalidated in that way (the *No. of associated eqs.* cannot be equal to 0) and therefore other solutions need to be taken into account.

The *flexibility level*, in a similar way as it is defined in [8], allows the ranking of the relative importance of the constraint in the overall flattened system of equations. The value can be in the range of

1 to 3, with 1 representing the most rigid equation and 3 being the most flexible equation. Equations, which are coming from a partial model and therefore are inherited by the final model, have a greater rigidity compared to the equations defined in the final model. For example, 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. Taking into account these formal rules, for an equation defined inside a Modelica class, a maximal flexibility level will be assigned. In conclusion a maximum flexibility level will be defined for the equations in the final model, followed by equations defined in partial classes and equations generated by the `connect` statements.

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

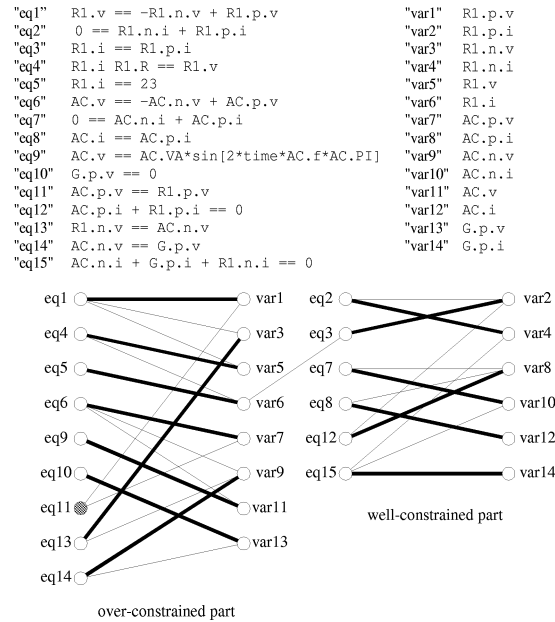
It is worth noting that the annotation attributes are automatically initialized by the static analyzer, incorporated in the front end of the compiler, by using several graph representations [12] of the declarative program code.

## 6. Debugging of an Over-Constrained System

Let us again examine the simple simulation example presented in Fig.1. where an additional equation ( $i=23$ ) was intentionally introduced inside the `Resistor` component in order to obtain a generally over-constrained system. The D&M canonical decomposition will lead to two parts: a well-constrained part and an over-constrained part (see Fig. 6.). Equation “*eq11*” is a non-saturated vertex of the equation set so it is a source for the over-constrained part. Starting from “*eq11*” the directed graph can be redrawn as is illustrated in Fig. 7a. An immediate solution of fixing the over-constrained part is to eliminate “*eq11*” which will lead to a well constrained part and therefore the equation system becomes structurally sound. However, examining the associated annotations to the “*eq11*” :

```
<AC.p.v == R1.pv, "eq11", " ", 2, "Circuit", 1, yes>
```

one can note that the equation is generated by a `connect` statement from the `Circuit` model and the only way to remove the equation is to remove the `connect (AC.p, R1.p)` statement. But removing the above-mentioned statement will remove two equations from the flattened model, which is indicated by the *No. of associated eqs.* = 2 parameter. One should also note the *flexibility level* of the equation is equal to 1, which is extremely low, indicating that the equation is extremely rigid. Therefore another solution need to be found, namely another equation need to be eliminated from the equation system instead of removing the equation `AC.p.v == R1.pv`.



**Fig. 6.** The decomposition of an over-constrained system

## 6.1 Calculating the Set of Over-Constraining Equations.

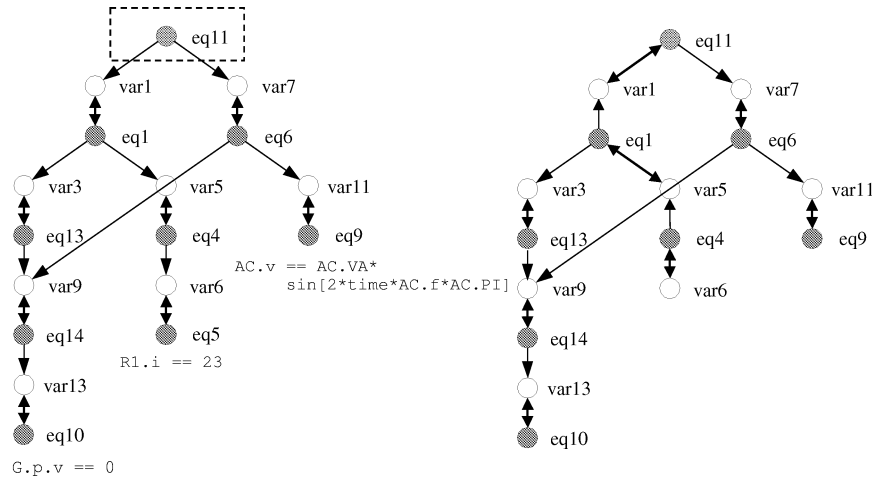
In the next step of the debugging procedure for the over-constrained system of equations we need to introduce several definitions regarding the some particular equation subsets which have special properties from the structural analysis point of view.

**Definition 9:** We call the *equivalent over-constraining equation list* associated to a system of equations the list of equations  $\{eq_1, eq_2, \dots, eq_n\}$  from where eliminating any of the component equations will lead to a well constrained system of equations.

**Definition 10:** We call the *reduced equivalent over-constraining equation list* the subset of equations obtained from the equivalent over-constraining equations after the language constraints have been applied.

When the size of the reduced equivalent over-constraining equation list exceeds 1, the automatic debugging is no longer available, and then the list should be presented by the debugger to the user in order to solve the conflicting situation.

From the over-constrained part resulting from the D&M decomposition we can construct an algorithm to find the equivalent over-constraining list based on the associated directed graph of the over-constrained part:



**Fig. 7. a)** An associated directed graph to the over-constrained part starting from “eq11”. **b)** The fixed well-constrained directed graph by eliminating equation “eq5”.

We describe the algorithm as follows:

---

### **Algorithm 2:** Finding the equivalent over-constraining equations list

---

**Input Data:** An over-constrained graph  $OG$  resulting after D&M decomposition applied to  $G$ .

**Result:** the reduced equivalent over-constraining equation list

**begin:**

- Compute the directed graph  $OG'$  of  $OG$  by replacing each edge in  $MG$  by two arcs and orienting all other edges from the equations to the variables.
- Find a depth-first search tree  $T$  in  $OG'$  with the root vertex being one of the sources of the directed graph  $OG'$ .
- Apply a strongly connected component decomposition algorithm for the graph obtained by removing the last visited equation vertex.
- **If** the number of strongly connected components is equal to 1 **then**  
     add the last visited equation vertex to the reduced list.
- Output the equivalent over-constraining equations list.

**end.**

---

The basic idea behind the algorithms for enumerating all the perfect matchings in a bipartite graph, presented in [11] and [21], is the exchange of matching edges with other edges along an alternating cycle or a feasible path. Enumerating all of the maximal matching which include the source vertex (in our case “eq11”) is not interesting, since they are equivalent from the solution relaxation point of view. All the



obtained maximal matching will include a perfect matching plus the source node. Therefore we are interested in finding other maximal matchings which include the source vertex leaving other equation vertexes free. One important property of the over-constrained bipartite graph is that it only contains alternating paths because it is constructed from perfect matching and a supplementary free edge. We can easily obtain all the maximal matchings in the over-constrained graph by exchanging matching edges with other edges along an alternating path. An algorithm for computing the reduced equivalent over-constraining equation list is given below:

---

**Algorithm 3:** Annotation based equation set reduction

---

**Input Data:** A reduced equation set taken from the output of **Algorithm 2**.

**Result:** the final reduced equivalent over-constraining equation list

**begin:**

- Eliminate from the list all of the equations generated by a `connect` statement and for which the *No. of associated eqs.* parameter exceeds the system degree of freedom.
- Eliminate all the equations for which the *No. of associated eqs.* parameter is equal to 1. Add that equation to the history list.
- Sort the remaining equations in decreasing order of flexibility level
- Output the sorted list of equations.

**end.**

---

If the length of the reduced equivalent over-constraining list is equal to 1 automatic debugging of the model is possible by eliminating the equation from the simulation model without any supplementary user-intervention. Of course the history list together with the elimination is output to the user. If the length of the list is greater than 1, this means that several error fixing strategies are possible and therefore user intervention is required. The reduced list is output to the user starting with the equation which has the higher flexibility level.

In our case the set of equivalent over-constraining equations is {"eq11", "eq13", "eq10", "eq5", "eq9"}. "Eq11" was already analyzed and therefore can be eliminated from the set. "Eq13" is eliminated too for the same reasons as equation "eq11". Analyzing the remaining equations {"eq10", "eq5", "eq9"} one should note that they have the same flexibility level and therefore they are candidates for elimination with an equal chance. But analyzing the value of the *No. of associated eqs.* parameter, equation "eq10" and "eq9" have that attribute equal to one, which means that they are singular equations defined inside the model. Eliminating one of these equations will invalidate the corresponding model, which is probably not the intention of the modeler. Examining the annotations corresponding to equation "eq5" one can see that it can be safely eliminated because the flexibility level is high and eliminating the equation will not invalidate the model since there is another equation defined inside the model. After choosing 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 ( $i=23$ ) indicating exactly to the user which equation need to be removed in order to make the simulation model mathematically sound. In that case the debugger correctly located the faulty equation previously introduced by us in the simulation model.

By examining the annotations corresponding to the set of equations which need to be eliminated, the implemented debugger can automatically determine the possible error fixing solutions and of course prioritize them. For example, by examining the flexibility level of the associated equation compared to the flexibility level of another equation the debugger can prioritize the proposed error fixing schemes. When multiple valid error fixing solutions are possible and the debugger cannot decide which one to chose, a prioritized list of error fixes is presented to the user for further analysis and decision. In those cases, the final decision must be taken by the user, as the debugger cannot know or doesn't have sufficient 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.

## 7. Prototype Debugger Implementation Details

For the above presented graph decomposition techniques to be useful in practice, we must be able to construct and manage the graph representation of the declarative specification efficiently. Another important factor that must be taken into account is the incrementality of the approach in order to

accommodate incremental static analyzers to be added to the existing simulation environment of the declarative equation based language. In this section, we outline the architecture and organization of the implemented debugger attached to the simulation environment.

A prototype debugger was built and attached to the *MathModelica* simulation environment as a testbed for evaluating the usability of the above presented graph decomposition techniques for debugging declarative equation based languages. *MathModelica* is an integrated problem-solving environment (PSE) for full system modeling and simulation [14][15]. The environment integrates Modelica-based modeling and simulation with graphic design, advanced scripting facilities, integration of code and documentation, and symbolic formula manipulation provided via *Mathematica*[23]. Import and export of Modelica code between internal structured and external textual representation is supported by *MathModelica*. The environment extensively supports the principles of literate programming and integrates most activities needed in simulation design: modeling, documentation, symbolic processing, transformation and formula manipulation, input and output data visualization.

The information output by the debugger will of course lead to a mathematically sound system of equations, however some of the solutions might not be acceptable from the modeling language point of view or from the physical system model perspective. The debugger focuses on those errors whose identification would not require the solution of the underlying system of equations.

As indicated previously, it is necessary for the compiler to annotate the underlying equations to help identify the equations and to help eliminating the heuristic involved in choosing the right solution. Accordingly, we modified the front end of the compiler to annotate the intermediate representation of the source code where equations are involved. The annotations are propagated appropriately through the various phases of the compiler, and, when an error is detected, the debugger uses them to eliminate some of the heuristics involved in the error solving process and, of course, to exactly identify the problematic equations and to generate error messages consistent with the user perception of the source code of the simulation model.

The implemented debugger was successfully tested on declarative models involving several hundreds of algebraic equations and differential algebraic equations.

The general architecture of the implemented debugger is presented in Fig. 8 and the debugging algorithm proceeds as follows: Based on the original declarative source code the intermediate representation is generated. From the intermediate representation the overall system of equations is extracted and transformed into bipartite graph form. The associated bipartite graph is canonically decomposed and error-fixing strategies are applied if the decomposition lead to over- or under-constrained components. The debugger will try to solve the errors automatically without the explicit intervention of the user. If automatic error solving is not possible due to missing information the user will be consulted regarding the repair strategy. When the user is interrogated, all valid options that will lead to a numerically sound underlying system of equations are presented.

---

**Algorithm 4:** Static Debugging of the declarative equation based simulation model

**Input Data:** The Modelica source code of the simulation Model

**Result:** Error fixing information consistent with the user perception of the original source code

**begin:**

- Each model  $\mathcal{M}$  specified in the Modelica modeling language is translated to an intermediate flattened form  $\mathcal{JM}$ .
- From  $\mathcal{JM}$  the flattened equations  $\mathcal{FEM}$ , variables  $\mathcal{FVM}$  and constant  $\mathcal{FCM}$  list are extracted.
- Annotate the  $\mathcal{FEM}$  with additional debugging information.
- From  $\mathcal{FEM}$  and  $\mathcal{FVM}$  obtain the bipartite graph representations  $\mathcal{G}$  of the model.
- Apply **Algorithm 1** in order to decompose  $\mathcal{G}$  into well-constrained  $\mathcal{WG}$ , over-constrained  $\mathcal{OG}$  and under-constrained  $\mathcal{UG}$  components.

**while**  $\mathcal{OG} \neq \emptyset$  and  $\mathcal{UG} \neq \emptyset$  **do**

- Apply error fixing strategies to the  $\mathcal{OG}$  by eliminating the additional constraining equations.
- Apply error-fixing strategies to  $\mathcal{UG}$  by adding additional equations to the model or initializing some of the variables.

**end.**

**end.**

---

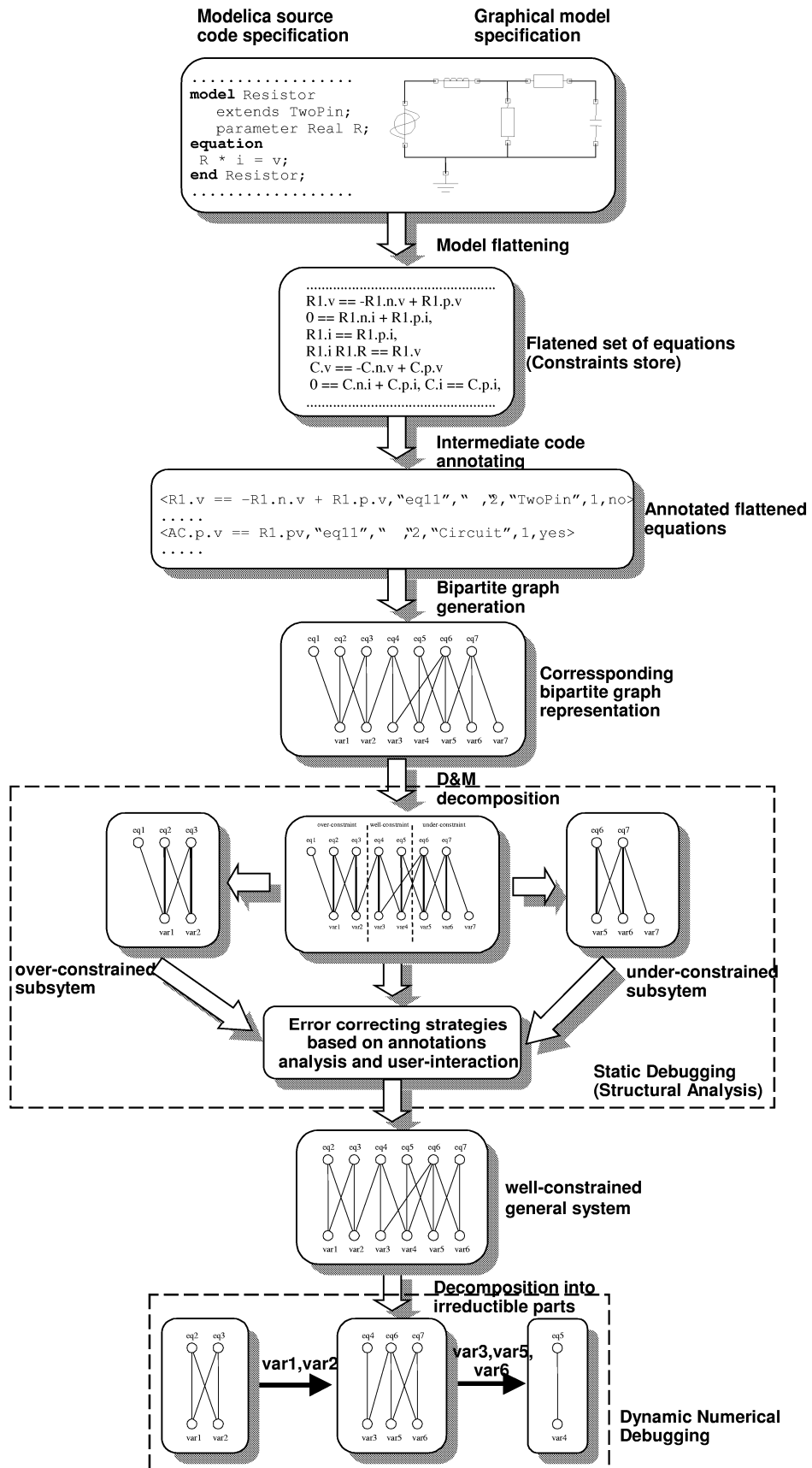


Fig. 8. General architecture of the implemented debugger

## 8. Summary and 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 a usage example for debugging an over-constrained simulation model. 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 understanding complicated simulation models.

The contributions of this paper are twofold: the proposal of integrating graph decomposition techniques for debugging declarative equation based languages and an efficient equation annotation structure which helps the debugger to eliminate some of the heuristics involved in the error solving process. The annotations also provides an efficient way of identifying the equations and therefore helps the debugger in providing error messages consistent with the user's perception of the original source and simulation model. The implemented debugger helps to statically detect a broad range of errors without having to execute the simulation model. Since the simulation system execution is expensive the implemented debugger helps to greatly reduce the number of test cases used to validate the simulation model.

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.
- when a simulation model fails to compile we report the error consistent with the user's perception of the simulation model or source code, and also several possible repair strategies are presented. The error fixing strategies are also prioritized by the debugger, which benefits the user in choosing the right error fixing solution.

## Acknowledgements

We thank the entire MathModelica team from MathCore AB, without which this work would not have been possible. The work has been supported by MathCore AB, by KK Industrial Research School and by the ECSEL Graduate School supported by the Swedish Strategic Research Foundation.

## References

- [1] Ait-Aoudia, S.; Jegou, R. and Michelucci, D. "Reduction of Constraint Systems." In *Compugraphic*, pages 83--92, Alvor, Portugal, 1993.
- [2] Asratian A.S.; Denley T. and Häggkvist R. *Bipartite Graphs and their Applications*. Cambridge University Press 1998.
- [3] Blik, C.; Neveu, B. and Trombettoni G. "Using Graph Decomposition for Solving Continuous CSPs", *Principles and Practice of Constraint Programming*, CP'98, Springer LNCS 1520, pages 102-116, Pisa, Italy, November 1998.
- [4] Dolan A. and Aldous J. *Networks and algorithms – An introductory approach*. John Wiley & Sons 1993 England.
- [5] Dulmage, A.L., Mendelsohn, N.S. *Coverings of bipartite graphs*, Canadian J. Math., 10, 517-534.
- [6] Elmqvist, H.; Mattsson S.E and Otter, M. "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) 1999.
- [7] Elmqvist, H. *A Structured Model Language for Large Continuous Systems*. PhD thesis TFRT-1015, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden. 1978.
- [8] Flannery, L. M. and Gonzalez, A. J. *Detecting Anomalies in Constraint-based Systems*, Engineering Applications of Artificial Intelligence, Vol. 10, No. 3, June 1997, pages. 257-268.
- [9] Fritzson, P. and Engelson, V. "Modelica - A Unified Object-Oriented Language for System Modeling and Simulation." In *Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP'98*, Brussels, Belgium, Jul. 20-24), 1998.
- [10] Fritzson, P.; Shahmehri, N.; Kamkar, M. and Gyimothy, T. *Generalized algorithmic debugging and testing*. ACM Letters on Programming Languages and Systems, 1(4):303--322, December 1992.

- [11] Fukuda, K.; Matsui, T. "Finding All The Perfect Matchings in Bipartite", *Appl. Math. Lett.* Vol. 7, No. 1 (1994), pages 15-18.
- [12] Harrold, M.J and Rothermel, G. "A Coherent Family of Analyzable Graphical Representations for Object-Oriented Software." *Technical Report* OSU-CISRC-11/96-TR60, November, 1996, Department of Computer and Information Science Ohio State University.
- [13] Hopcroft, J.E. and Karp, R.M. *An  $n^{5/2}$  algorithm for maximum matchings in bipartite graphs.* SIAM Journal of Computing, 2(4):225--231, December 1973.
- [14] Jirstrand, M. "MathModelica – A Full System Simulation Tool". In Proceedings of *Modelica Workshop 2000* (Lund, Sweden, Oct. 23-24), 2000.
- [15] Jirstrand, M.; Gunnarsson J. and Fritzson P. "MathModelica – a new modeling and simulation environment for Modelica. " In *Proceedings of the Third International Mathematica Symposium (IMS'99, Linz, Austria, Aug), 1999.*
- [16] Maffezzoni C.; Girelli R. and Lluca P. *Generating efficient computational procedures from declarative models.* Simulation Practice and Theory 4 (1996) pages 303-317.
- [17] Modelica Association. *Modelica – A Unified Object-Oriented Language for Physical Systems Modeling - Tutorial and Design Rationale Version 1.4* (December 15, 2000).
- [18] Modelica Association. *Modelica – A Unified Object-Oriented Language for Physical Systems Modeling – Language Specification Version 1.4.* (December 15, 2000).
- [19] Saldamli, L.; Fritzson, P. "Object-oriented Modeling With Partial Differential Equations". In Proceedings of *Modelica Workshop 2000* (Lund, Sweden, Oct. 23-24), 2000.
- [20] Shapiro Ehud Y. *Algorithmic Program Debugging.* MIT Press (May). 1982.
- [21] Takeaki Uno. "Algorithms for Enumerating All Perfect, Maximum and Maximal Matchings in Bipartite Graphs." In *Proceedings of Eighth Annual International Symposium on Algorithms and Computation (ISAAC '97)*, Singapore, December 17--19, 1997.
- [22] Tiller M. Michael. *Introduction to Physical Modeling with Modelica.* Kluwer Academic Publisher 2001.
- [23] Wolfram S.; *The Mathematica Book.* Wolfram Media Inc. (February). 1996.