Automated Static Analysis of Equation-Based Components

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

Mathematical modeling and the simulation of complex physical systems are emerging as key technologies in engineering. The availability of static analyzers and automatic debuggers for detecting structural and numerical inconsistencies in the simulation models is crucial. To address this need, the authors propose a methodology for detecting and repairing overconstrained and underconstrained situations based on graph-theoretical approaches. Components and equations that cause the irregularities are automatically isolated, and meaningful error messages for the user are elaborated. The authors have implemented the AMOEBA (Automatic Modelica Equation-Based Analyzer) environment to support the development and specification of correct equation-based simulation models by applying graph-theoretical approaches and semiautomatic debugging techniques. The implementation architecture and preliminary experiments with a prototype debugger integrated in the symbolic and numeric engine, ModSimPack, of the Modelica language compiler are presented and discussed.

Keywords: Modelica, debugging, overconstrained system, underconstrained system, bipartite graphs

1. Introduction

Recent years have witnessed a significant growth of interest in the modeling and simulation of physical systems. A key factor in this growth has been the development of efficient equation-based simulation languages. Such languages have been designed to allow automatic generation of efficient simulation code from declarative specifications. A major objective was to facilitate the exchange of models, model libraries, and simulation specifications. The Modelica language [1-3] and its associated support technologies have achieved considerable success through the development of domain libraries. By using domain libraries, complex simulation models can be built by aggregating and combining submodels and components from various physical domains.

However, a significant part of the software development effort is spent on detecting deviations from specifications and subsequently localizing the sources of such errors. The real challenge in Modelica-based modeling and simulation has moved now from designing a powerful modeling language and efficient numerical solvers to designing flexible integrated environments and simulators with enhanced debugging capabilities. The high-level abstraction of equation-based models presents new challenges to modeling and simulation tools due to the large gap between the declarative specification and the executable machine code. This abstraction gap leads to difficulties in finding and correcting model inconsistencies and errors, which are not uncommon in the process of developing complex physical system models.

A typical problem that appears in physical system modeling and simulation is when too many or too few equations are specified in the system, thus 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. These situations very often arise during the development of new models and new model libraries. The final end users usually interact with an equation-based modeling and simulation system through a graphical editor, where readily available model components from component libraries are connected together to form a new model. The advantage of using libraries is to reuse as much well-tested code as possible in the models. This will minimize the effort for testing, and the end user will not be exposed to the Modelica code. However, overconstrained and underconstrained situations can still appear due to wrong connections among incompatible components, missing connections, or redundant components. As an example, let us consider a simple electrical circuit model that is structurally sound and that correctly simulates. An end user can easily add a second instance of the ground component to the electrical circuit model. The first ground component, together with the electrical components of the circuit, will form a well-constrained system of equations from where the behavior of the circuit can be computed. However, the second ground component will introduce a redundant

SIMULATION, Vol. 80, Issue 7–8, July–August 2004 321-345
© 2004 The Society for Modeling and Simulation International
DOI: 10.1177/0037549704046340

equation, making the whole system of equations corresponding to the electrical circuit structurally inconsistent. Therefore, a two–ground circuit model cannot be successfully compiled and executed even if the equations associated with each component are correctly stated.

The user should be able to deal with overdetermined and underdetermined systems of equations by identifying the minimal set of equations or variables that should be removed from the system to make the remaining set of equations solvable. For example, let us consider a physical system simulation model specified in a declarative object-oriented, equation-based modeling language that consists of several hundreds of components resulting in several thousands of equations. However, one of these equations overconstrains the overall system, making it impossible to simulate. 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 component. It can easily be imagined that a static debugger that presents a small subset of overconstraining 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. In this way, the component that has caused the failure can be isolated and repaired.

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 for structural inconsistencies. To address this need, we propose a methodology for the declarative debugging of equation-based languages by adapting graph decomposition techniques for reasoning and performing structural analysis of the underlying systems of equations. We propose a verification module tightly integrated with the language compiler in which certain conditions can be checked prior to the quantification of the model and before embarking on a computationally expensive numeric or symbolic solution-finding process. Moreover, the verification tools can provide strong assurance that the simulation model is free of certain types of errors and inconsistencies. We therefore expect static analysis to improve the reliability of equation-based languages used in the development of simulation components, as well as their transparency to developers and end users.

The remainder of this article is organized as follows. Section 2 gives a quick overview of component-based modeling and simulation with the help of the Modelica language. In section 3, the compilation process of the Modelica language is briefly described, and ModSimPack, a symbolic and numeric engine of the Modelica compiler, is introduced. In this context, details about the compilation of equation-based languages into imperative code are also given. Section 4 discusses the basic concepts of structural singularity associated with a system of equations. In the next section, the notion of structural singularity is expressed in terms of bipartite graph matchings associated with the system of equations. Sections 6 and 7 explain the debugging of overconstrained and underconstrained situations by presenting graph-based algorithms that handle those situations. Section 8 presents the architecture of AMOEBA (Automatic Modelica Equation-Based Analyzer). AMOEBA is a static analysis module that has been integrated into the Modelica compiler to support the development and specification of correct equation-based simulation models by applying graph-theoretical approaches and semiautomatic debugging strategies. Finally, section 9 presents the summary and conclusions of the article.

2. Component-Based Modeling and Simulation with Modelica

Modelica is a rather new language for hierarchical object-oriented physical modeling that is being developed through an international effort [1-3]. The language unifies and generalizes previous object-oriented modeling languages. Modelica is intended to become a de facto standard. It allows defining simulation models in a declarative manner, modularly and hierarchically, and combining various formalisms expressible in the more general Modelica formalism. The multidomain capability of Modelica gives the user the possibility to combine electrical, mechanical, hydraulic, and thermodynamic model components within the same application model.

In the context of Modelica *class libraries*, software components are Modelica classes. However, when building particular models, components are *instances* of those Modelica classes. Classes should have well-defined communication interfaces—sometimes called ports (called *connectors* in Modelica)—for communication between a component and the outside world. A component class should be defined *independently of the environment* where it is used, which is essential for its *reusability*. This means that in the definition of the component, including its equations, only local variables and connector variables can be used. No means of communication between a component and the rest of the system, apart from going via a connector, are allowed. A component may internally consist of other connected components (i.e., *hierarchical* modeling).

To grasp this complexity, one must have a pictorial representation of components and connections. Such graphic representation is available as *connection diagrams*, of which a schematic example is shown in Figure 1, where a complex car simulation model is built in a graphical model editor. An integrated graphical user interface (GUI) in the form of a *model editor* (see Fig. 1) allows simulation practitioners and knowledge engineers to express problems in terms with which they are familiar. Therefore, when employing a model editor, minimal knowledge of programming languages is needed, and typing is kept to a minimum. The basic functionality of a model editor includes selecting components from ready-made libraries (shown on the right in Fig. 1), connecting components in model diagrams, and entering parameter values for different components. More complex simulation models can be built by simply

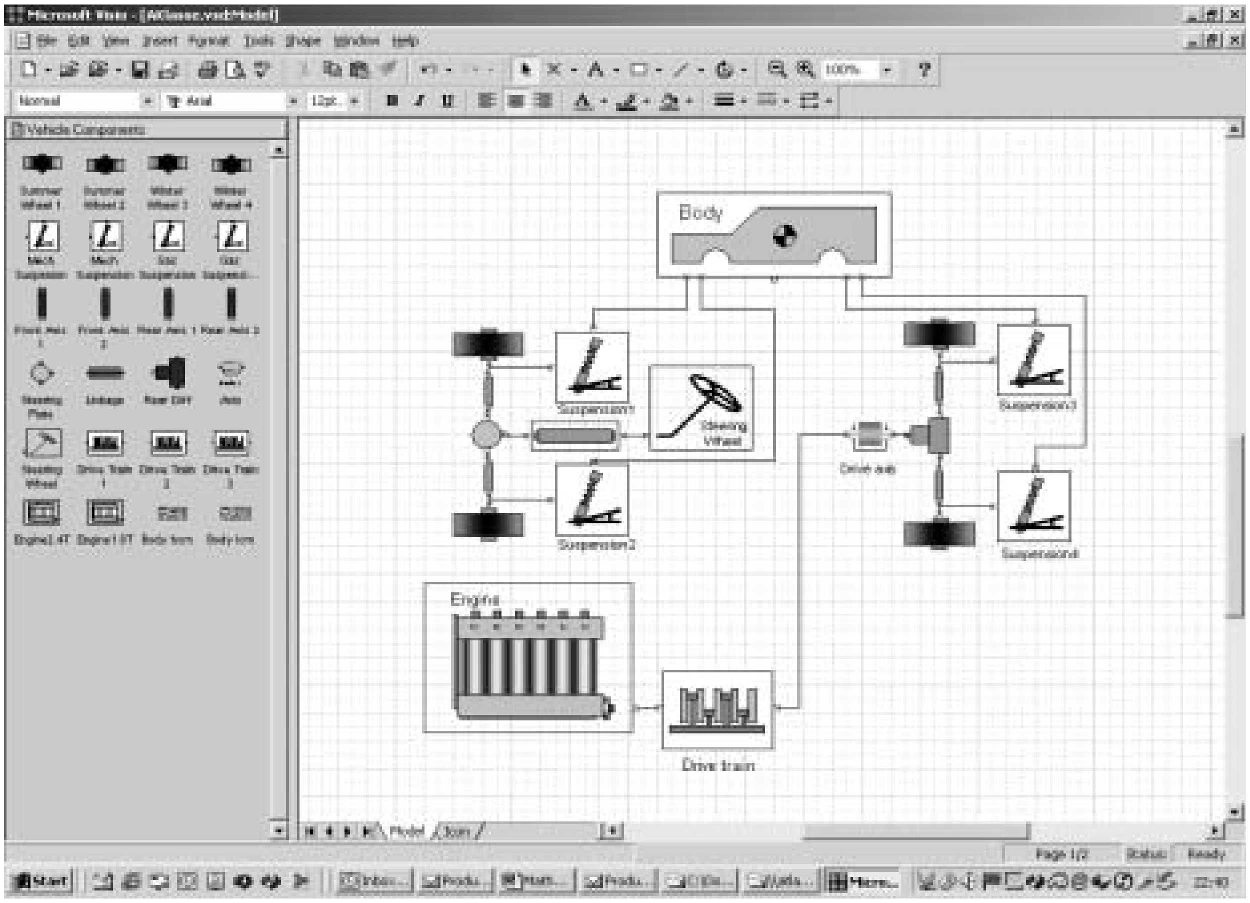


Figure 1. Complex simulation models can be built by combining readily available components from domain libraries

combining available library models. Some of the model libraries cover application areas such as mechanics, electronics, hydraulics, and pneumatics. These libraries are primarily intended to tailor the simulation environment toward a specific domain by giving modelers access to common model elements and terminology from that domain.

In the following subsection, a short overview of the Modelica language is given. Readers should refer to Fritzson [4], Tiller [5], and the Modelica Association [6, 7] 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.

2.1 A Short Introduction to Modelica

A Modelica program is built from classes, just as in any other traditional object-oriented language. The main difference compared with traditional object-oriented languages is that instead of functions (methods), equations are used to specify the behavior. A class declaration contains a list of variable declarations and a list of equations preceded by the keyword *equation*. We illustrate below a class corresponding to a resistor (`Resistor`) and an alternative voltage source (`VsourceAC`) modeled in Modelica.

```
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 = 3.14;
equation
    v = VA * sin (2*PI*f*time);
end VsourceAC
```

It also should be noted that both classes are specializations through an inheritance mechanism of a special class called TwoPin. The natural inheritance mechanism in the Modelica language works by extending classes with new equations and variables. The TwoPin class that defines electrical components that have two pins is defined as follows:

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

This class instantiates twice the class Pin, which is a special kind of class called the connector class. Such connectors declare variables that are part of the communication interface of an object defined by the connectors of that object. Thus, connectors specify the interface for an interaction between a component and its surroundings. Our connector Pin class uses two Real variables: one for the current (i) and one for the voltage (v). Since in an electrical circuit, the current should always be summed when connecting two components, according to Kirchhoff's law, the variable i defined in the Pin class will have the prefix flow.

```
connector Pin
    Real v;
    flow Real i;
end Pin;
```

Besides the instantiation of two pin interface objects (also called ports or connectors), some extra equations are provided that define the behavior of the objects, such as the voltage drop along the component (v = p.v - n.v) or the current inside the component (0 = p.i + n.i; i = p.i).

Connections between objects can be established between connectors of an equivalent type. Modelica supports equation-based acausal connections, which means that connections are defined as special equation forms. A connection equation form, such as connect(pin1,pin2) with pin1 and pin2 of connector class Pin, connects the two pins so that they form one node. This is equivalent to and is eventually expanded into two equations—namely,

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

The first equation says that the voltages of the connected wire ends are the same. The second equation corresponds to Kirchhoff's current law saying that the currents sum to zero at a node (assuming a positive value while flowing into the component). The sum-to-zero equations are generated when the prefix flow is used. Similar laws apply to flows in piping networks and to forces and torques in mechanical systems.

2.2 Simple Simulation Examples

Now we have all the components that are necessary to define the model of a simple electrical circuit consisting of a sinusoidal voltage source and a resistor connected together, as depicted in Figure 2.

Let us consider more complicated simulation examples of an electrical generator depicted in Figure 3, in which an inertial mass Inertia is rotated with an angular acceleration, given by acceleration component A. An electromechanical component EMF will transform the rotational mechanical energy into electrical energy that is absorbed by a simple electrical circuit consisting of an inductor (Ind) and two resistors (R1 and R2) that are connected in series. We are interested in measuring the voltage between node (1) and node (2) of the electrical circuit with the help of a voltage sensor component, Vsen. The simulation model components come from three distinct component libraries: block, mechanical, and electrical libraries. They are connected together in a model editor similar to the one presented in Figure 1. A simulation environment will derive and execute the simulation code that defines the behavior of the whole model. This illustrates a much more complex modeling example than the one presented in Figure 2. However, in terms of complexity and number of components, it is still a very small modeling problem. A typical modeling problem in Modelica can involve several hundreds of components that can easily result in thousands of flattened equations.

A plot of the input signal Ex.y, the angular acceleration in the inertial component Inertia.w, and the voltage measured by the voltage sensor VSen.v are depicted in Figure 4.

3. The Compilation Process

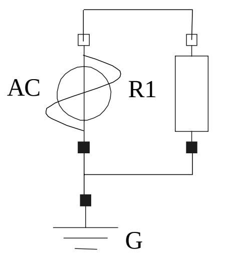
Throughout this article, we draw a parallel between the process of the static analysis of physical system models and the process of compiling equation-based languages in which those models are specified.

3.1 Compiler General Architecture

To gain a better understanding of how an equation-based language compiler works, it is useful to take a look at the compilation process of the Modelica language, which is sketched in Figure 5.

The Modelica source code is first translated into a so-called "flat model." This phase includes type checking and performing all object-oriented operations such as inheritance, modifications, and so forth. The flat model includes a set of equation declarations and functions, with all the object-oriented structure removed, apart from the dot notation within the names. This process is called the *partial instantiation* of the model. More details about this stage in the compilation process can be found in Fritzson et al. [8].

As an example, let us consider the simple electrical circuit model presented in Figure 2. After the partial



```
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
```

Figure 2. Simple electrical circuit model

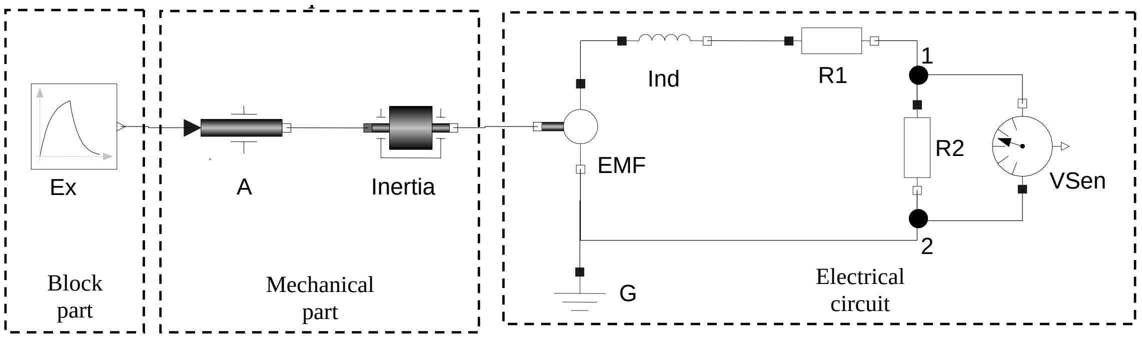


Figure 3. Simulation model corresponding to an electromechanical device

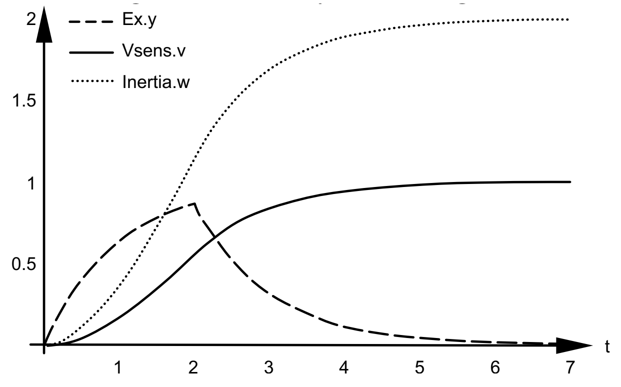


Figure 4. Plots of the simulation results of the electromechanical circuit

instantiation phase, this model will result in the flattened system of equations shown in Table 1.

The next obvious step is to solve the system of equations. First, the equations need to be transformed into a suitable form for the numerical solvers. This is done by the symbolic and the numeric modules of the compiler. We give only a very brief introduction to ModSimPack, our implementation of the symbolic and numeric optimizer module for a Modelica compiler.

ModSimPack takes as input the flattened form of the equations. The equations are mapped into an internal data structure that permits simple symbolic manipulations such as common subexpressions elimination, algebraic simplifications, constant folding, and so on. These symbolic operations decrease the complexity of the system of equations substantially. After this stage, the block lower triangular (BLT) form of the system of equations is computed. Since this stage is extremely important in the context of the structural analysis of the system of equations, we provide in the next subsection a more extensive description.

Finally, in the last phase, the procedural code (in our implementation, C++ code) is generated based on the previously computed BLT blocks. This phase requires a preliminary symbolic transformation phase that will transform the sequential blocks into a form accepted by the numerical solver.

3.2 Block Lower Triangular Form

To solve the overall system of equations, we must first determine the correct computation order of the calculations for the model variables. The overall system of equations needs to be ordered to facilitate the use of numerical solvers for improving efficiency. It is desirable to obtain a lower

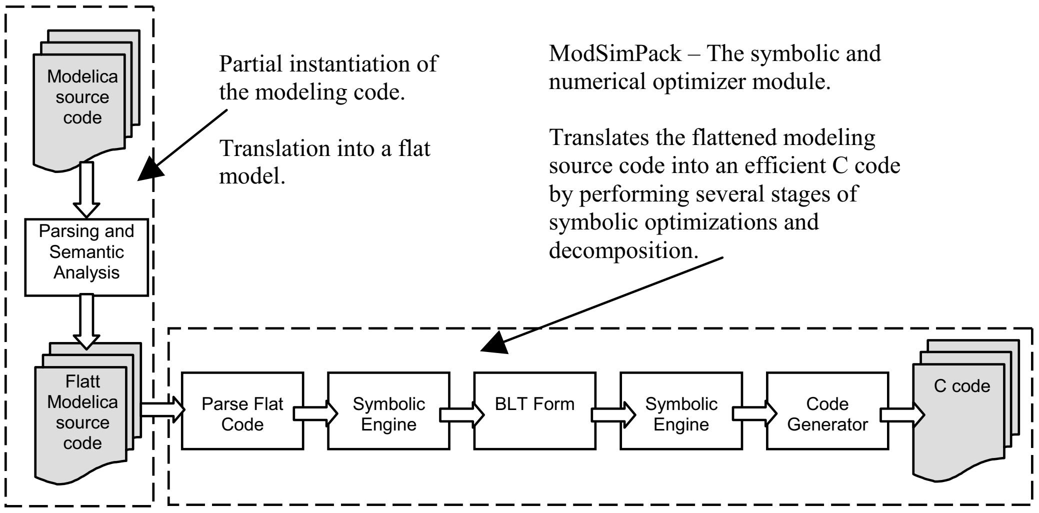

Modelica source code
Partial instantiation of the modeling code.
Translation into a flat model.
ModSimPack – The symbolic and numerical optimizer module.
Translates the flattened modeling source code into an efficient C code by performing several stages of symbolic optimizations and decomposition.
Parsing and Semantic Analysis
Flatt Modelica source code
Parse Flat Code
Symbolic Engine
BLT Form
Symbolic Engine
Code Generator
C code

Figure 5. The main stages in the Modelica language compilation process

Table 1. The flattened set of equations and variables corresponding to the simple electrical circuit from Figure 2

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

triangular form of the permuted incidence matrix. The lower triangular form guarantees that the equations can be sequentially solved one at a time by a forward substitution process. In general, the structure of the incidence matrix extracted from the equations of a physical system simulation model is not lower triangular, and in most cases, it is not possible to find a permutation to transform it into a strictly lower triangular form. However, efficient algorithms exist to transform matrices to the BLT form, as depicted in Figure 6 (by the notation X, we denote the presence of a nonzero element in the matrix). BLT decomposition is a two-stage algorithm. First, a row permutation of the incidence matrix that finds a zero-free diagonal needs to be found (see Fig. 6b). This problem is also known as the problem of finding the maximum transversal. The algorithm is extensively described in Duff and Reid [9], and a Fortran implementation of the algorithm can be found in ACM Algorithm 575, proposed by Duff and Reid [10]. After computing the maximum transversal, the rows and columns of the incidence matrix are permuted again into the BLT form. The algorithm is described by Duff and Reid [11], and a Fortran implementation given by the same authors [12] is known as ACM Algorithm 529.

The advantage of using the BLT decomposition is twofold. First, the overall system of equations is decomposed into smaller blocks that can be solved sequentially by a forward substitution process. Second, solving the blocks sequentially is computationally more efficient in terms of execution time and memory storage than solving the whole system of equations once.

Let us illustrate the above-mentioned advantages with the process of solving a linear equation system $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ is a known matrix corresponding to the incidence matrix, $\mathbf{b}$ is a vector, and $\mathbf{x}$ is a vector corresponding to the unknowns. If $\mathbf{A}$ can be permuted in such a way that it admits a zero-free diagonal, then the linear equation system

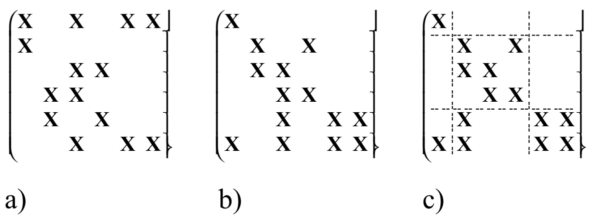


Figure 6. (a) The initial structural incidence matrix, (b) the permuted incidence matrix with a zero-free diagonal, (c) and the block lower triangular (BLT) form of the incidence matrix

$\mathbf{Ax} = \mathbf{b}$ can be transformed into the following form:

$$\begin{bmatrix} \mathbf{A}_{11} & 0 & \cdots & 0 \\ \mathbf{A}_{21} & \mathbf{A}_{22} & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ \mathbf{A}_{m1} & \mathbf{A}_{m2} & \cdots & \mathbf{A}_{mm} \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_m \end{bmatrix} = \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \\ \vdots \\ \mathbf{b}_m \end{bmatrix},$$

where $m \leq n$ is the number of blocks.

The algorithm for solving the original system $\mathbf{Ax} = \mathbf{b}$ rewrites the system as a sequence of subproblems:

$$\mathbf{A}_{11}\mathbf{x}_1 = \mathbf{b}_1$$

$$\mathbf{A}_{22}\mathbf{x}_2 = \mathbf{b}_2 - \mathbf{A}_{21}\mathbf{x}_1$$

$$\mathbf{A}_{33}\mathbf{x}_3 = \mathbf{b}_3 - \mathbf{A}_{31}\mathbf{x}_1 - \mathbf{A}_{32}\mathbf{x}_2$$

$$\cdots$$

$$\mathbf{A}_{mm}\mathbf{x}_m = \mathbf{b}_m - \mathbf{A}_{m1}\mathbf{x}_1 - \mathbf{A}_{m2}\mathbf{x}_2 - \cdots - \mathbf{A}_{m(m-1)}\mathbf{x}_{m-1}$$

Below, we illustrate the sparsity pattern of the incidence matrix associated with the system of flattened equations of the simple electromechanical device described in section 2.2, Figure 3. The flattened set of equations contains a number of 40 equations obtained after the constants and parameter folding and the reduction of single assignments. The corresponding sparsity pattern of the initial incidence matrix is depicted in Figure 7 on the left. The sparsity pattern after the BLT decomposition is illustrated in Figure 7 on the right. After performing the BLT decomposition, 26 sequential blocks were found.

Another advantage is that the BLT form of the equation systems facilitates the debugging of equation systems when too many or too few equations are specified, as demonstrated later.

4. Structural Nonsingularity

Definition 1. We call a square matrix $\mathbf{A}_{m\times m}$ *structurally nonsingular* if and only if there exist permutations $\mathbf{P}_1$, $\mathbf{P}_2$, such that $\mathbf{P}_1\mathbf{AP}_2$ has a nonzero diagonal. Otherwise, the matrix is called *structurally singular*.

The above definition is illustrated by the following example. Let us consider a square matrix $\mathbf{A}_{3\times 3}$ such as the one shown below, where zero elements are present in the diagonal. However, by exchanging the second column with the third column, we obtain a matrix $\mathbf{A} \times \mathbf{P}$, which has a nonzero diagonal. Based on definition 1, the matrix $\mathbf{A}$ presented in the following example is structurally nonsingular:

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{pmatrix} 2 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 2 \end{pmatrix}$$

$$\mathbf{AP} = \begin{pmatrix} 2 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 2 & 1 \end{pmatrix}.$$

It is extremely important to differentiate the notion of *structural singularity* from the notion of *numerical singularity*. Below, we present the definition of *numerical singularity*, as given in Health [13]:

Definition 2. A square matrix $\mathbf{A}_{m\times m}$ is numerically nonsingular if it satisfies one of the equivalent conditions:

1. $\mathbf{A}$ has an inverse (i.e., there is a matrix $\mathbf{A}^{-1}$ such that $\mathbf{AA}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$, where $\mathbf{I}$ is the identity matrix).
2. $\det(\mathbf{A}) \neq 0$ (i.e., the determinant of $\mathbf{A}$ is nonzero).
3. $\operatorname{rank}(\mathbf{A}) = m$ (the rank of a matrix is the maximum number of linearly independent rows or columns it contains).
4. For any vector $\mathbf{z} \neq 0,\ \mathbf{Az} \neq 0$ (i.e., $\mathbf{A}$ annihilates non-trivial vectors).

The concept of matrix nonsingularity can be easily extended to a linear system of equations. Let us consider a linear system of equations with n equations and n unknowns, such as the one shown in (1):

$$\begin{aligned} &a_{11}x_1 + \cdots a_{1n}x_n = b_1 \\ &\vdots \\ &a_{n1}x_1 + \cdots a_{nn}x_n = b_n. \end{aligned} \tag{1}$$

In a matrix-vector notation, a system of linear algebraic equations has the following form:

$$\mathbf{Ax} = \mathbf{b}, \text{ where } \mathbf{A} = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix}$$

$$\mathbf{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \text{ and } \mathbf{b} = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}. \tag{2}$$

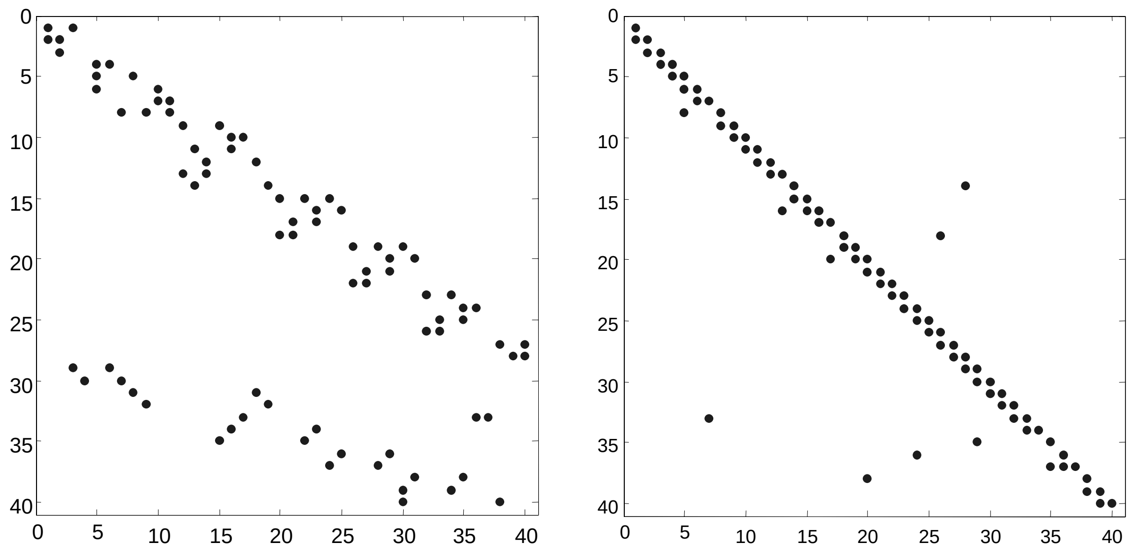


Figure 7. The sparsity pattern associated with the electromechanical device model from Figure 3, before and after the block lower triangular (BLT) transformation

Based on definition 1 and equation (2), the notion of structural nonsingularity for a linear system of equations can be formulated as follows:

Definition 3. A system of linear algebraic equations $\mathbf{Ax} = \mathbf{b}$ is structurally nonsingular if

1. The coefficient matrix $\mathbf{A}$ is a square matrix (the number of equations is equal to the number of variables).
2. $\mathbf{A}$ is structurally nonsingular.

The existence and uniqueness of a solution to a system of linear algebraic equations $\mathbf{Ax} = \mathbf{b}$ depend on whether the matrix $\mathbf{A}$ is numerically singular. The structural singularity checks whether the system of equations is well posed but cannot guarantee anything about the existence and uniqueness of the solution. For this reason, checking the structural singularity is considered a preprocessing phase to the more powerful notion of the numerical singularity. However, an equation system that is structurally nonsingular may still be numerically nonsingular. During static analysis, it is enough to limit ourselves to the notion of structural singularity. According to definition 2, the numerical singularity implies that the determinant of $\mathbf{A}$ is different from zero or that $\mathbf{A}$ has an inverse. Computing the determinant $\det(\mathbf{A})$ or the inverse $\mathbf{A}^{-1}$ is as expensive as solving $\mathbf{Ax} = \mathbf{b}$. During static analysis, all numerical computations should be avoided to provide prompt and quick feedback to the user. Therefore, when analyzing the system of equations in this stage, we make the assumption that the structural nonsingularity 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.

To be useful for our purposes, the structural nonsingularity needs to be further extended to cover systems of ordinary differential equations (ODEs) and systems of differential algebraic equations (DAEs). ODEs and DAEs are the most frequent kinds of systems of equations that arise in physical system simulation when modeling with equation-based languages. We should start by giving the most general representation of a DAE system in fully implicit form [14]:

$$F(\dot{x}, x, t) = 0, \tag{3}$$

where F and x are vector valued, and t is the time. A special case is when we can solve for $\dot{x}$, if we can transform equation (3) in the explicit or normal form:

$$\dot{x} = f(x, t). \tag{4}$$

The condition that needs to be fulfilled for this transformation is that the Jacobian of F with respect to $\dot{x}$ (denoted by $\partial F/\partial \dot{x}$) is nonsingular. In this case, the system described in (3) is an ODE. If $\partial F/\partial \dot{x}$ is singular, the system is called a DAE. In a DAE, there are algebraic constraints on the variables. The algebraic constraints can be explicitly represented as in the following equation:

$$\begin{aligned} F(\dot{x}, x, y, t) &= 0, \\ G(x, y, t) &= 0, \end{aligned} \tag{5}$$

where F and x are vector valued, t is the time, and the algebraic variables are constrained by the function G. For a DAE represented as in (5), the Jacobian of F with respect to $\dot{x}$ (denoted by $\partial F/\partial \dot{x}$) is nonsingular.

The DAE system from (5) can be rewritten in a semi-explicit form as follows:

$$\begin{aligned} \dot{x} &= f(x, y, t), \\ 0 &= g(x, y, t). \end{aligned} \tag{6}$$

The structural representation of the system of differential equations (6) is given by the *index matrix representation* [15]. The index matrix representation for a DAE system such as (6) is a $n \times n$ matrix $\mathbf{M}$ (as shown in (7)) consisting of leading derivative indices, where each element $\mathbf{M}_{i,j}$ corresponds to the leading derivative index of variables x and y in equations f and g.

$$\mathbf{M} = \begin{pmatrix} \frac{\partial f}{\partial \dot{x}} & \frac{\partial f}{\partial y} \\ \frac{\partial g}{\partial x} & \frac{\partial g}{\partial y} \end{pmatrix} = \begin{pmatrix} \mathbf{I}_d & \frac{\partial f}{\partial y} \\ \frac{\partial g}{\partial x} & \frac{\partial g}{\partial y} \end{pmatrix}, \tag{7}$$

where $\mathbf{I}_d$ stands for the $d \times d$ identity matrix, and d is the number of variables that are differentiated in the system. The structural singularity of the DAE system can be verified on the sparsity pattern associated with $\mathbf{M}$.

As an example, we are considering a differential algebraic system representing a chemical reaction that takes place in a chemical reactor. The example is taken from Brenan, Campbell, and Petzold [14]:

$$\begin{aligned} \dot{C} &= K_1(C_0 - C) - R, \\ \dot{T} &= K_1(T_0 - T) + K_2 R - K_3(T - T_C), \\ 0 &= R - K_3 e^{-\frac{K_4}{T}} C, \\ 0 &= C - u. \end{aligned} \tag{8}$$

The unknowns in the DAE system corresponding to (8) are C, T, R, T_C. It is important to note that in definition 1, the notion of structural nonsingularity is defined by the existence of a nonzero diagonal of the associated matrix. The numerical values of the elements have no influence on the notion of structural singularity. Therefore, when the structural nonsingularity of a system of equations is checked, it is enough to consider the sparsity pattern of the index matrix representation, denoted by $\mathbf{M}_S$. The sparsity matrix $\mathbf{M}_S$ shown in (9) is structurally nonsingular since, in accordance with definition 1, there exists a permutation $\mathbf{P}$ such that $\mathbf{M}_S \times \mathbf{P}$ has a nonzero free diagonal, as shown in the following equation (by the notation *, we denote the presence of a nonzero element):

$$\mathbf{M}_S = \begin{array}{c} \begin{array}{cccc} C & T & R & T_C \end{array} \\ \left(\begin{array}{cccc} * & 0 & * & 0 \\ 0 & * & * & * \\ * & * & * & 0 \\ * & 0 & 0 & 0 \end{array}\right) \end{array} \begin{array}{l} \\ eq1 \\ eq2 \\ eq3 \\ eq4 \end{array}, \tag{9}$$

$$\mathbf{M}_S\mathbf{P} = \begin{array}{c} \begin{array}{cccc} R & T_C & T & C \end{array} \\ \left(\begin{array}{cccc} * & 0 & 0 & * \\ * & * & * & 0 \\ * & 0 & * & * \\ 0 & 0 & 0 & * \end{array}\right) \end{array} \begin{array}{l} \\ eq1 \\ eq2 \\ eq3 \\ eq4 \end{array}. \tag{10}$$

The nonsingularity of the index matrix associated with a system of equations is a necessary and sufficient condition for the local uniqueness of the solution. To check the uniqueness of the solution of the overall simulation model, the nonsingularity of each index matrix associated with each BLT subsystem needs to be verified. Checking the nonsingularity of the index matrix is as expensive as solving the system of equations. However, the notion of structural singularity is a necessary condition for the uniqueness of the solution. The structural singularity can be more efficiently investigated by means of a graph-theoretical approach, as described in the following sections. The chosen graph-theoretical approach should accommodate the notion of structural nonsingularity as well as the BLT form of the overall system of equations.

For a more detailed discussion of the numerical properties of ODEs and DAEs, the reader is referred to Brenan, Campbell, and Petzold [14]; Deuflhard and Bornemann [16]; and Bendtsen and Thompsen [17].

5. The Bipartite Graph Representation

The next obvious step in our analysis is to establish a similar notion to the structural nonsingularity on the graph-based representation of the incidence matrix corresponding to the system of equations. First, we recall some necessary notions from graph theory.

Definition 4. A bipartite graph is an ordered triple $G = (V_1, V_2, E)$ such that $V_1 = \{v_1, \ldots, v_k\}$ and $V_2 = \{u_1, \ldots, u_k\}$ are sets of vertices, $V_1 \cap V_2 = \oslash$, and $E \subseteq \{\{x, y\}; x \in V_1, y \in V_2\}$ is the set of edges. The vertices of G are elements of $V_1 \cup V_2$. The edges of G are elements of E.

We consider the bipartite graph associated with a given system of equations resulting from flattening an object-oriented hierarchical model. Let V_1 be the set of equations and V_2 the set of variables in the flattened model. An edge between $eq \in V_1$ and $var \in V_2$ means that the variable var appears in the equation eq. Based on this rule, the associated bipartite graph of the flattened system of equations of the simple electrical circuit model from Figure 2 is shown in Figure 8.

We introduce the following notations and definitions:

Definition 5. A *matching*, denoted M_G^k, is a set of k edges from a graph G, where no two edges have a common end vertex. If $G = (V_1, V_2, E)$ is a bipartite graph with bipartition $V_1 = \{v_1, \ldots, v_k\}$ and $V_2 = \{u_1, \ldots, u_k\}$, we denote by $\partial^1 M_G$ and $\partial^2 M_G$ the sets of vertices in V_1 and V_2, respectively, incident to arcs in M_G.

Definition 6. A matching M_G^{max} of a graph G is called a *maximum cardinality matching* or *maximum matching* if it is a matching with the largest possible number of edges.

Definition 7. A *perfect matching* M_G^p is a matching in a graph G that covers all vertices of G. For a perfect matching

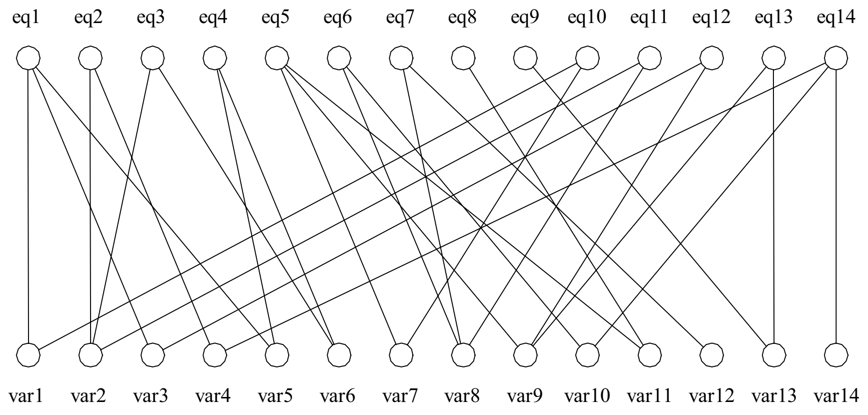


Figure 8. The associated bipartite graph of the simple electrical circuit model from Figure 2

corresponding with a bipartite graph $G = (V_1, V_2, E)$, the following relation holds: $| G |=| M |=| \partial^1 M | + | \partial^2 M |$, where the notation $| G |$ stands for the cardinality of the graph G.

Figure 9 illustrates one possible maximum matching of the bipartite graph associated with the flat equations of the simple electrical circuit presented in Figure 2. It is worth noting that in this particular case, the maximum matching is also a perfect matching of the associated bipartite graph.

It should also be noted that a maximum matching or a perfect matching of a given bipartite graph is not unique. In Figure 10, all the possible perfect matchings of a simple bipartite graph are presented.

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 [18]. 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. Efficient algorithms for enumerating all perfect and maximum matchings in bipartite graphs are proposed in Fukuda and Matsui [19] and Uno [20, 21]. The enumeration algorithm for all perfect matchings in bipartite graphs, proposed in Fukuda and Matsui [19], takes $O(n^{1/2}m + mN_p)$ time, where N_p is the number of perfect matchings in the given bipartite graph. Uno proposes two improved algorithms for finding and enumerating all perfect and maximum matchings, which take only $O(n)$ [20] and $O(\log n)$ [21] time, respectively, by perfect matching.

It should be noted that the structural singularity of a system of equations is the same as the finding of a perfect matching in the associated bipartite graphs. The existence of a nonzero diagonal of the incidence matrix guarantees the existence of a maximum transversal in the corresponding bipartite graph that is equivalent to the existence of a perfect matching.

We shall now present a structural decomposition algorithm that relies on maximum matchings and vertex coverings for a bipartite graph associated with a simulation model. The algorithm is due to Dulmage and Mendelsohn [22] and canonically decomposes any maximum matching of a bipartite graph into three distinct parts: overconstrained, underconstrained, and well constrained. Let us consider a system of linear equations and its associated bipartite graph, as presented in Figure 11. One possible maximum matching M is represented by the thick edges.

In the next step of our analysis, we exchange all the edges that are included in the matching M with bidirectional edges and orient all other edges from equation nodes to variable nodes. The following graph depicted in Figure 12 is obtained.

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. In a similar way, we compute for the free variable nodes the set of all ancestors that sink into the free node and isolate the graph. The well-constrained subgraph can further be decomposed by isolating and defining a partial-order relation among the subgraphs induced by its strongly connected components. In this way, the Dulmage and Mendelsohn decomposition results in an ordering of variables and equations that permits the sequential solving of the diagonal blocks obtained after permutation. Performing these steps, we obtain the graph decomposition shown in Figure 13.

The perfect matching associated with a bipartite graph is not unique. However, the final decomposition into irreducible blocks is unique.

The Dulmage and Mendelsohn algorithm for the canonical decomposition of bipartite graphs is given below and results in three distinct parts of the graph: the overconstrained, underconstrained, and well-constrained parts. Furthermore, the algorithm decomposes the well-constrained part into irreducible components and establishes a partial-order relation among them. These irreducible components are similar to the equation blocks

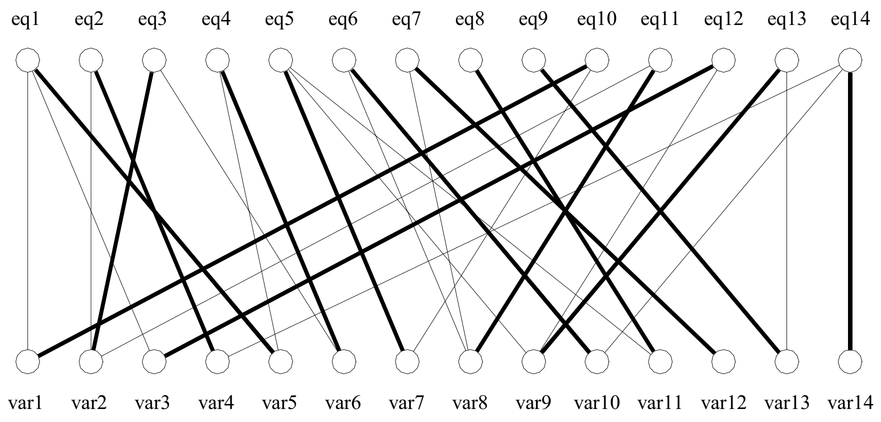


Figure 9. One possible perfect matching (marked by thick lines) of the bipartite graph associated with the electrical circuit model

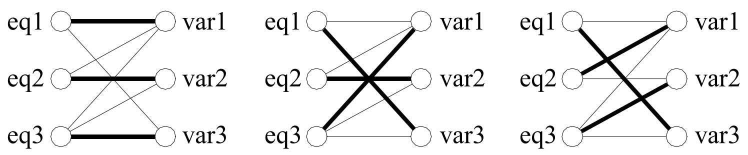
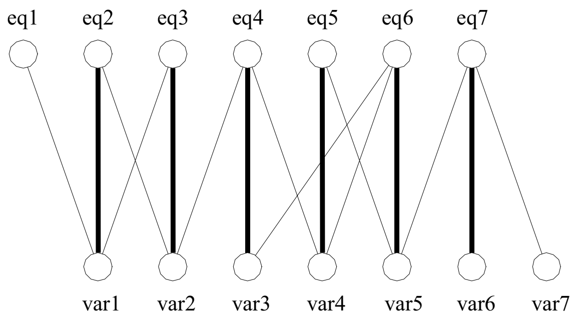


Figure 10. An example of a simple bipartite graph with all possible perfect matchings marked by thick lines



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

Figure 11. Dulmage-Mendelsohn's canonical decomposition of a bipartite graph

computed by the block triangular decomposition performed on the incidence matrix.

Algorithm 1: Dulmage and Mendelsohn canonical decomposition

Input: A bipartite graph $G = (V_1, V_2, E)$.

Output: Three subgraphs: well-constrained W_G, overconstrained O_G^{k+}, and underconstrained U_G^{k-}.

Compute the maximum matching M_G^{max} of $G = (V_1, V_2, E)$.

Compute the directed graph $\bar{G} = (V_1, V_2, \bar{E})$, where $\bar{E}$ is obtained by replacing each edge that is included in M_G^{max} by two directed edges oriented from V_1 to V_2 and from V_2 to V_1, respectively, and orienting all other edges from V_1 to V_2.

$\bar{E} = \{(\overleftrightarrow{u, v}) \mid (u, v) \in \epsilon(M_G^{max})$ and $(\overrightarrow{u, v}) \mid (u, v) \in E - \epsilon(M_G^{max})\}$.

Let O_G^{k+} be the set of all descendants of the k sources of the directed graph $\bar{G}$. O_G^{k+} is the overconstrained subgraph of G induced on $\{v \in V_1 \cup V_2 \mid u \longrightarrow v$ on $\bar{G}$ for some $u \in V_1 - \partial^1 M\}$ where $u \longrightarrow v$ is the notation for a simple path from vs tex u to vertex v.

Let U_G^{k-} be the set of all ancestors of k sink of the

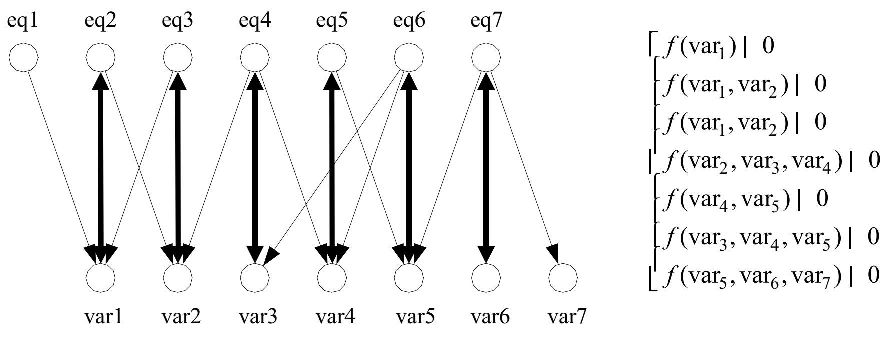


Figure 12. Oriented bipartite graph

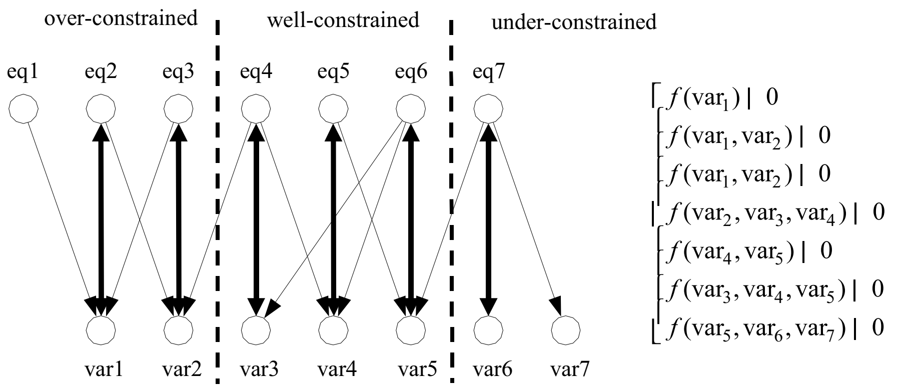


Figure 13. Canonical bipartite graph decomposition

directed graph $\bar{G}$. U_G^{k-} is the overconstrained subgraph of G induced on $\{v \in V_1 \cup V_2 \mid v \longrightarrow u$ on $\bar{G}$ for some $u \in V_2 - \partial^2 M\}$.

Calculate $W_G = \bar{G} - O_G^{k+} - U_G^{k-}$. W_G is obtained by deleting from $\bar{G}$ all the vertices and edges of O_G^{k+} and U_G^{k-}.

Compute the strongly connected components $S_{Gs}(s = 1 \dots n)$ of W_G.

Compute the subgraphs W_{Gs} of W_G induced on $S_{Gs}(s = 1 \dots n)$.

Define the partial order on $W_{Gs}(s = 1 \dots n)$.

Define the partial order $O_G^{k+} \prec W_{G_s} \prec U_G^{k-}$ for any s.

The *overconstrained* 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 overconstraining equations from the system to make the remaining system well constrained. Even if the additional equations would be *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 *underconstrained* 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 to obtain a well-constrained part or add additional equations to the system.

Overconstrained and underconstrained situations can coexist in the same model. In the case of an overconstrained model, the user would like to remove the overconstraining equations in a manner that is consistent with the original source code specifications to alleviate the model definition.

The *well-constrained* part: the number of equations in the system is equal to the number of variables, and all the equation and variable nodes are covered by a perfect matching. Therefore, the underlying mathematical system of equations is structurally nonsingular. This part can further be 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 solving the well-constrained

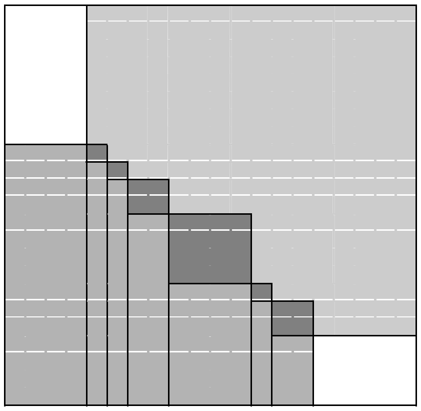


Figure 14. Dulmage and Mendelsohn (D&M) decomposition with one overconstrained and one underconstrained block at the beginning and at the end of the incidence matrix

part numerically means that no valid solution exists, and somewhere there is numerical redundancy in the system.

The concept of Dulmage and Mendelsohn (D&M) decomposition can be extended to the incidence matrix corresponding to the bipartite graph. In this way, we obtain the BLT form of the matrix, as already mentioned in section 3.2. If the model is not correctly formulated and presents overconstrained and underconstrained components, these components will appear at the beginning and at the end of the sparsity pattern, respectively, as shown in Figure 14. The square blocks in the middle represent the irreducible blocks of the well-constrained part obtained after the canonical decomposition. Before embarking on a numerical solution, the overconstrained and underconstrained blocks need to be made into square blocks by eliminating extra equations and eliminating extra variables, respectively.[1]

Our structural analysis algorithms, employed in the following sections, use the bipartite graph-based representation of the system of equations instead of the incidence matrix and the sparsity pattern representation. Even if they represent the same abstraction, we believe that the graph representation is more expressive and useful for generating explanations of possible bug sources and locations than the incidence matrix representation.

6. Detecting and Repairing Overconstrained Blocks

Let us again examine the simple simulation example presented in Figure 2, where an additional equation (`i = 23`) was intentionally introduced inside the `Resistor` component to obtain a generally overconstrained system. The D&M canonical decomposition will lead to two different subgraphs: a well-constrained part W_G and an overconstrained part O_G^{1+}, as shown in Figure 15.

Starting from $eq11$, the directed graph can be derived from the undirected bipartite graph, as illustrated in Figure 16, by exchanging all the matching edges into bidirectional edges and orienting all other edges from equation to variable nodes. The general error-fixing strategy in the case of overconstrained equation subsystems is to remove the extra equations. An immediate fix to the overconstrained part is to remove $eq11$, which will lead to a well-constrained part. 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.

Therefore, several important criteria need to be developed based on the combinatorial properties of the overconstraining subgraph and the filtering rules imposed by the modeling language semantics before proceeding to the elimination of extra equations. We give the following definition regarding the safe elimination of an equation node.

Definition 8. Any node v of the overconstraining subgraph O_G^{k+} corresponding to the bipartite graph G is safe for removal if, by removing that node and the corresponding incident edges $inc_E(v)$, the remaining undirected graph is connected.

The elimination of an "unsafe" node is depicted in Figure 17. Based on definition 8, we call the *equivalent overconstraining equation set* associated with an overconstrained part of a system of equations the set of equation nodes $\{eq_1, eq_2, \ldots, eq_n\}$ that are safe for removal.

From the overconstrained part O_G^{1+} resulting from the D&M decomposition, depicted in Figure 16, we can construct an algorithm to find the equivalent overconstraining set based on the associated directed graph of the overconstrained part.

We describe the algorithm as follows:

Algorithm 2: Finding the equivalent overconstraining equations set

Input: The directed overconstrained graph $\overline{O_G^{k+}}$ resulting after D&M decomposition applied to G.

Output: The reduced equivalent overconstraining equation set L.

```
begin
    Initialize $L = \{\}$;
    for each free node $v_k \in v(\overline{O_G^{k+}})$ do
```

1. This constitutes the naive approach to the overconstrained and underconstrained problem. The following two sections provide additional details on how to systematically debug such components.

"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`		

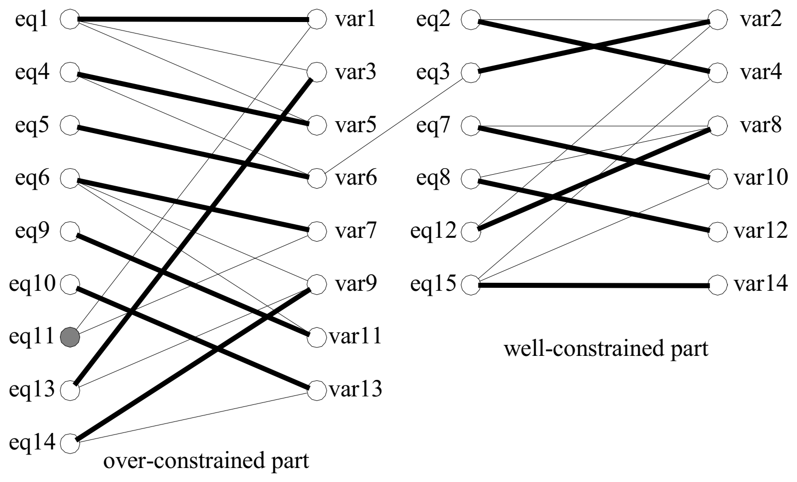


Figure 15. Canonical decomposition of an overconstrained system

```
        Construct a depth-first search tree $T$ in $\overline{O_G^{k+}}$ starting
        with the root vertex $v_k$
        for each node $n \in T$ do
            if $n \in V_1$ ($n$ is an equation node) then
                - Hide $n$ from $\overline{O_G^{k+}}$ and all the adjacent edges
                - Compute the number of strongly connected
                  components $no_{str}$ of $\overline{O_G^{k+}}$.
                if $no_{str} == 1$ then
                    - add $n$ to $L$
                end if
                - Restore $n$ and all the adjacent edges
            end if
        end for
    end for
    Return the equivalent overconstraining equation
     set $L$.
end
```

In our particular example, the set of equivalent overconstraining equations calculated by the above algorithm is $\{eq11, eq13, eq10, eq5, eq9\}$. It should be noted that the notion of 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 combinatorial 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 need to developed and included in the debugging strategy.

6.1 Equation Annotations

The previously computed reduced equivalent equation set can further be refined by taking into account simple rules derived from the language semantics. To provide a mechanism to reason about the erroneous model under consideration based on language semantics rules, we need to annotate the equations. For annotating the equations, we use a structure that resembles the one developed by Flannery and Gonzalez [23]. We define an annotated equation as a record with the following structure: <`equation, name, description,`

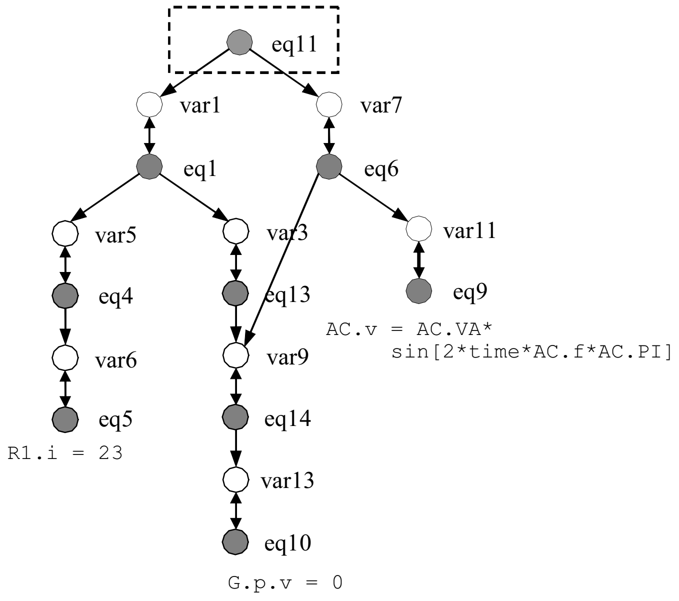


Figure 16. A directed graph associated with the overconstrained part starting from $eq11$

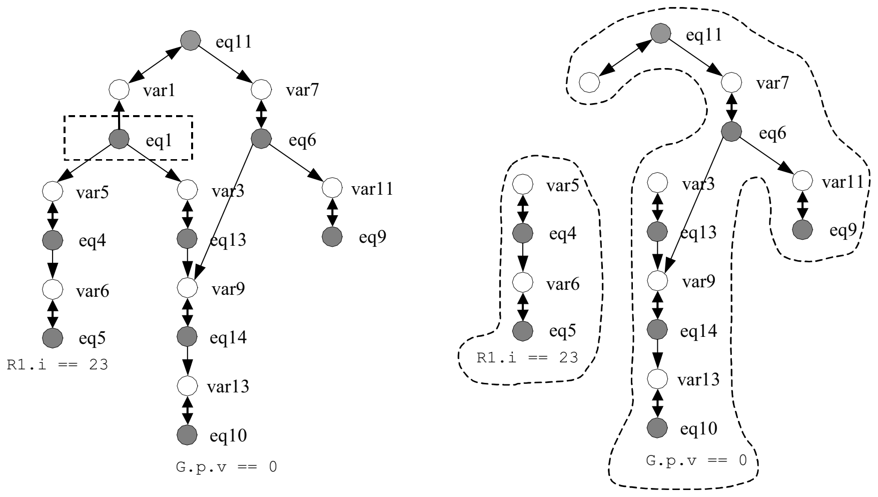


Figure 17. The elimination of an unsafe equation node ($eq1$) from the overconstrained subgraph (on the left) leads to two disconnected components (on the right)

`number of associated equations, class name, flexibility level, connector generated, number of linked equations`>.

An example including the annotations associated with an equation extracted from the `Resistor` model described in section 2.1 is given in Table 2. The values defined by annotations are later incorporated in the error repair strategies. These values are used to choose the right error-fixing solution from a series of possible repair strategies.

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 *number of associated equations* field defines the number of equations that are specified together with the

Table 2. 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*
Number of associated equations	1
Class name	*Resistor*
Flexibility level	3
Connector generated	No
Number of linked equations	0

annotated equation inside the same model. In the example given in Table 2, the number of associated equations is equal to 1 since there are no additional equations specified in the `Resistor` class. 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 overconstraining equation and therefore needs to be eliminated, then the elimination is not possible because the model will be invalidated (the *number of associated equations* 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 [23], 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 that define 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 with 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 that tells whether the equation is generated by a `connect` equation. Usually, these equations have a very low flexibility level.

The *number 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 the original declarative code is kept during the translation phases. However, "trusted" components can be manually marked by the user. This operation will set the value of the flexibility level annotation to zero for each equation associated with a trusted component.

The annotations associated with the set of equivalent overconstraining equations $\{eq11, eq13, eq10, eq5, eq9\}$ are shown in Table 3. 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 are candidates for elimination with equal probability. However, by analyzing the value of the *number of associated equations* annotation, equations $eq10$ and $eq9$ have this attribute equal to 1, 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 with 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 the *number of linked equations* 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 the *number of associated equations* annotation, which is equal to 2. 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 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.

Table 3. The associated annotations of the equivalent overconstraining equation set

Name	Equation	Number of Associated Equations	Class Name	Flexibility Level	Connector Generated	Number of Linked Equations
$eq11$	`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.pv = 0`	1	`Ground`	2	No	0
$eq5$	`R1.i = 23`	2	`Resistor`	2	No	0
$eq9$	`AC.v = AC*VA* sin(2t*AC.f*AC.PI)`	1	`VsourceAC`	2	No	0

By examining the annotations corresponding to the set of equations that need to be eliminated, the debugger can automatically determine the possible error-fixing solutions and 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 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 overconstraining. 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.

6.2 Higher Degree of Overconstraining

This section describes how the algorithms from the previous sections are modified when multiple free nodes are present in the overconstrained subgraph.

We construct a simple electrical circuit model by connecting two resistors in parallel with a voltage source, as shown in Figure 18. The Modelica definition of the `Ground`, `VsourceAC`, and `Resistor` classes are reused from the previous example. The `TwoPin` class is modified by introducing an additional overconstraining equation (`i = 10`) in the model definition. This extra equation will be inherited by all the classes that extend the `TwoPin` class. Therefore, each instance of the `Resistor` and `VsourceAC` models will contribute to one extra overconstraining equation to the final flattened system of equations.

During model translation, the flattened set of equations corresponding to the simulation model is derived, and the associated bipartite graph G is constructed. The flattened model corresponding to the simple electrical circuit now contains three extra overconstraining equations. Therefore, three equation vertices are not covered by a maximum matching. By choosing an arbitrary maximum cardinality matching and performing a D&M canonical decomposition, the overconstrained subgraph O_G^{3+}, as depicted in Figure 19, is obtained.

The maximum cardinality matching for the corresponding bipartite graph leaves three vertices corresponding to the equations $eq9$, $eq18$, and $eq17$ uncovered. Three equations need to be eliminated from the overconstrained subgraph to make the system well constrained. One equation needs to be eliminated from each overconstrained part O_{1G}^{1+}, O_{2G}^{1+}, O_{3G}^{1+} (see Fig. 20).

The set of safe overconstraining equations associated with each overconstrained subgraph can be computed using algorithm 2. We obtain the following reduced set of equation nodes:

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

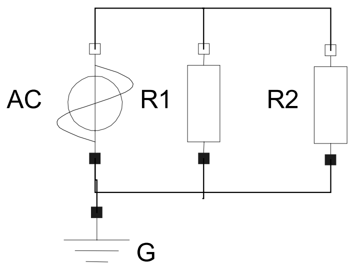
$$\{eq18, eq3, eq14, eq20, eq21, eq16, eq4\} \in v(O_{2G}^{1+}),$$

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

The safe equation nodes are highlighted in Figures 19 and 20 by dashed ellipses.

At this stage, there are multiple choices for elimination. As we already mentioned, one equation node needs to be eliminated for each overconstraining subgraph. Presenting this information to the user with the list of safe equations is unfeasible at this stage due to the large number of combinations.

It should be noted that some equations appear in more than one safe equation set. For example, $eq3$ appears in the safe equation sets associated with the subgraphs O_{1G}^{1+} and O_{2G}^{1+}. This means that $eq3$ can be made a free vertex by exchanging matching edges with nonmatching 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 from the subgraph O_{1G}^{1+}, it cannot be scheduled again for elimination from the subgraph O_{2G}^{1+}, even though it is present in the set of safe equations associated with the subgraph. Moreover, if $eq3$ is scheduled for elimination from subgraph O_{1G}^{1+}, this will also affect the elimination of node $eq14$ from subgraph O_{2G}^{1+}. The operation of exchanging the nonmatching 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 $eq14$ from $eq18$. Therefore, $eq14$ cannot be selected any more for elimination from O_{2G}^{3+}, even if it previously had a valid alternating path to the free node $eq18$.



```
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 18. Electrical circuit with two resistors connected in parallel with an additional equation introduced in the `TwoPin` class

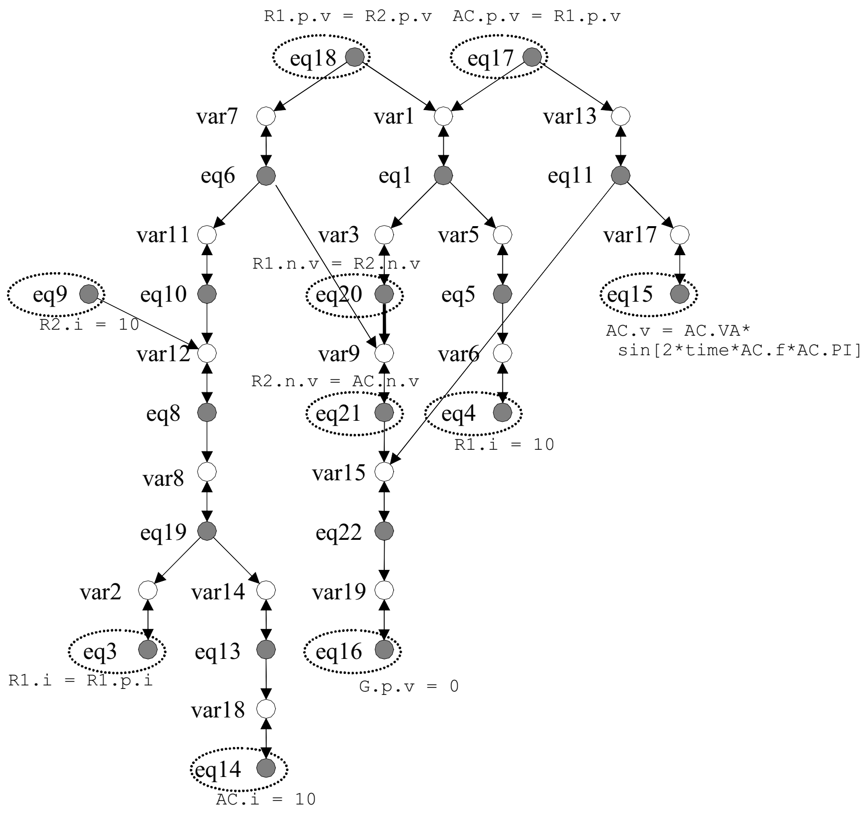


Figure 19. The overconstrained directed graph

By taking into account these additional structural constraints together with the constraints imposed by the language semantics extracted from the annotation associated with each equation, only one possible combination of three equation nodes that can be safely removed is found. Removing *eq*4 (`R1.i = 10`), *eq*9 (`R2.i = 10`), and *eq*14 (`AC.i = 10`) equation nodes from the overconstraining graph will make the graph well constrained without violating any structural and semantics rule. All three equations were obtained by inheritance from the original equation `i = 10` in the `TwoPin` class. Therefore, the overconstraining equation `i = 10` from `TwoPin` class has been correctly identified.

7. Detecting and Repairing Underconstrained Blocks

In the following sections, we illustrate the possible error-fixing solutions for a typical underconstrained situation and the reasoning involved in the graph transformation

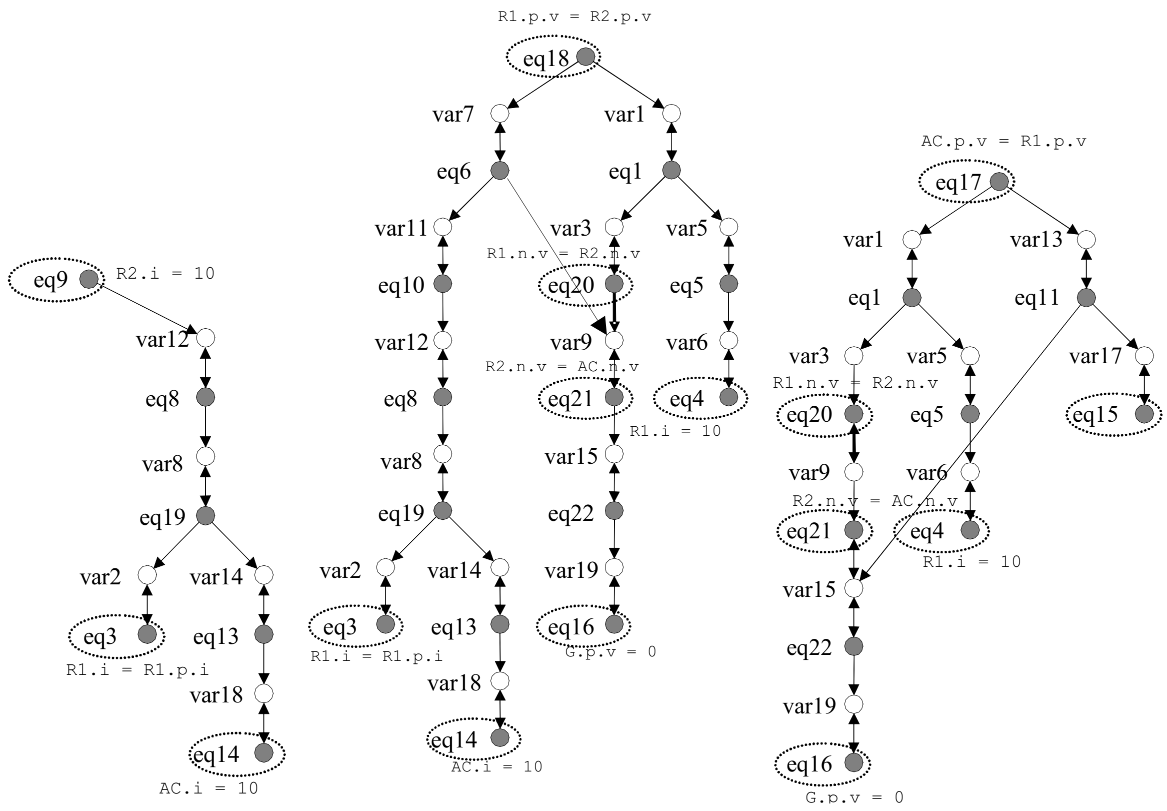


Figure 20. The O^{1+}_{1G}, O^{1+}_{2G}, O^{1+}_{3G} components of the O^{3+}_{G} overconstrained subgraph

system. Let us consider the following system of equations presented in Figure 21 on the right. The corresponding bipartite graph to the system of equations is also given in Figure 21 on the left.

One possible maximal matching (represented by thicker edges) of the bipartite graph and the D&M canonical decomposition is given in Figure 22. The first step when performing the canonical decomposition algorithm is to transform the undirected bipartite graph G into a directed graph $\bar{G}$ by exchanging all the edges that are part of the maximal matching for bidirectional edges and by orienting all other edges from equations to variable nodes. The corresponding directed graph $\bar{G}$ is shown in Figure 23.

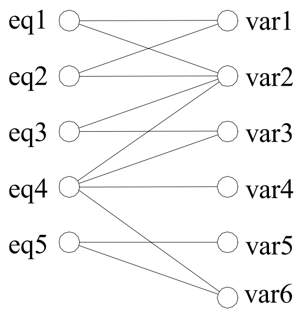
Based on the D&M canonical decomposition algorithm, the underconstrained part U^{1-}_{G} contains all the equation and variable nodes that sink into the free variable node. The variables contained in an underconstrained part constitute an *eligibility set*. In our small example, the eligibility set is $\{var4, var5, var6\}$. Any of the variables from the eligibility set can be taken away, and the remaining associated graph will be well constrained. Variable $var6$ is not covered by the maximal matching and therefore is a free vertex. In the directed graph $\bar{G}$, it can be noticed that there are two alternating paths that sink into the free vertex $var6$ (as indicated by the dashed arrows in Figure 23):

$$\{(\overrightarrow{var4, eq4}), (\overrightarrow{eq4, var6})\}\text{and}\{(\overrightarrow{var5, eq5}), (\overrightarrow{eq5, var6})\}.$$

By exchanging the matching edges with nonmatching edges and the nonmatching edges with matching edges along an alternating path, a new matching that covers the free vertex $var6$ can be obtained, but it will uncover another vertex from the eligibility set. Therefore, an error-fixing strategy must consider all the possible combinations that remove one variable node from the eligibility set.

During the first stage of the error-fixing process, only those solutions that involve the elimination of a variable from the eligibility set are taken into account. We have the following possible solutions illustrated in Figure 24.

By removing $var6$ from the underconstrained subsystem U^{1-}_{G}, the considered maximum matching becomes a perfect matching of the remaining bipartite graph. Therefore, the associated system of equations can be considered structurally sound. By removing $var6$, the resulting bipartite graph will, however, be disconnected, and an independent edge (*eq5,var5*) that is not connected to the main



$$\begin{cases} f(\mathrm{var}_1, \mathrm{var}_2) \mid 0 \\ f(\mathrm{var}_1, \mathrm{var}_2) \mid 0 \\ f(\mathrm{var}_2, \mathrm{var}_3) \mid 0 \\ f(\mathrm{var}_2, \mathrm{var}_3, \mathrm{var}_4, \mathrm{var}_6) \mid 0 \\ f(\mathrm{var}_5, \mathrm{var}_6) \mid 0 \end{cases}$$

Figure 21. A simple system of equations with the associated bipartite graph

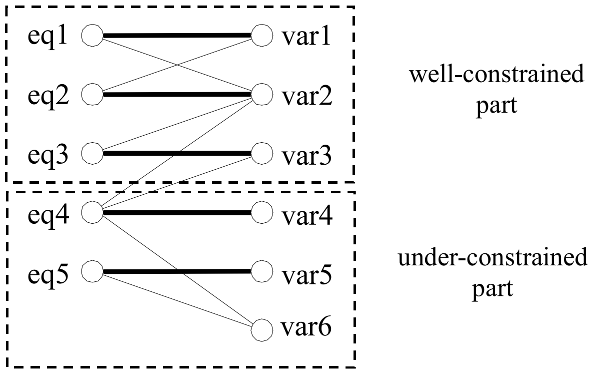


Figure 22. Maximum matching and canonical decomposition of the bipartite graph

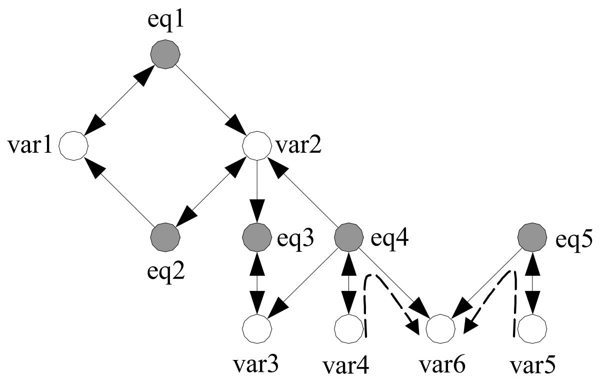


Figure 23. Directed graph associated with the system of equations

bipartite graph appears in the system. This situation is unusual in physical system modeling, and therefore it can be safely discarded from the error-fixing solutions.

It should also be noted that multiple error-fixing strategies are possible in the case of underconstrained subsystems. Another error-fixing strategy for an underconstrained system is to add one extra equation to the system and link the free variable to the added equation instead of eliminating it. This strategy, applied for the free variable $var6$, is presented in Figure 25b, in which an extra equation $eq6$ is added to the overall system of equations.

This strategy involves two steps: at the first step, an extra equation is added and linked to the free variable. Then, at the second step, additional checking is performed to see if other variables from the system might be present in the added equation (Fig. 25c).

Let us again analyze the simple electrical circuit model from Figure 2, in which the `Resistor` component is changed again by declaring an extra variable (`Real s`) and introducing this variable into the following `Resistor` model equations:

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

Obviously, this modification will introduce one extra variable without increasing the number of equations in the system. The directed graph obtained from the associated bipartite graph of the flattened underlying system of equations, as well as one possible corresponding maximum cardinality matching, is given in Figure 26. The correspondence between the variable and equation node labels is given in Table 4.

The uncovered variable, when using the considered maximum cardinality matching, is $var15$, and the computed eligibility set is $\{var15, var4, var2, var6, var7, var11, var9, var13\}$, with the corresponding variables `{G.p.i, R.n.i, R.p.i, R.i, R.s, AC.n.i, AC.p.i, AC.i}`. From the underconstrained directed subgraph, we can derive a number of three alternating paths (indicated in Fig. 26 by the dashed arrows) to the uncovered variable $var15$.

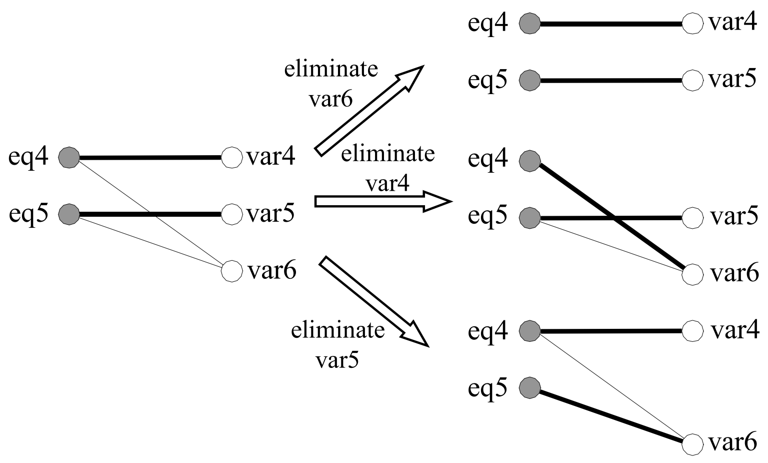


Figure 24. Error-fixing solutions when one variable is eliminated from the eligibility set

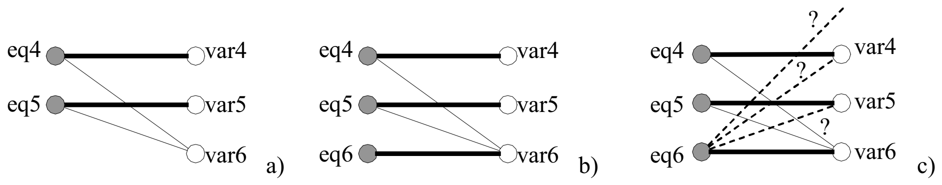


Figure 25. Error-fixing strategy involving adding an extra equation

Table 4. Flat form of the equations corresponding to the underconstrained electrical circuit model with a modified resistor

*eq*1	`R.v = -R.n.v + R.p.v`	*var*1	`R.p.v`
*eq*2	`0 = R.n.i + R.p.i`	*var*2	`R.p.i`
*eq*3	`R.i = R.p.i`	*var*3	`R.n.v`
*eq*4	`R.i * R.R = R.s * R.v`	*var*4	`R.n.i`
*eq*5	`AC.v = -AC.n.v + AC.p.v`	*var*5	`R.v`
*eq*6	`0 = AC.n.i + AC.p.i`	*var*6	`R.i`
*eq*7	`AC.i = AC.p.i`	*var*7	`R.s`
*eq*8	`AC.v = AC.VA * sin[2*time*AC.f*AC.PI]`	*var*8	`AC.p.v`
*eq*9	`G.p.v = 0`	*var*9	`AC.p.i`
*eq*10	`AC.p.v = R.p.v`	*var*10	`AC.n.v`
*eq*11	`AC.p.i + R.p.i = 0`	*var*11	`AC.n.i`
*eq*12	`R.n.v = AC.n.v`	*var*12	`AC.v`
*eq*13	`AC.n.v = G.p.v`	*var*13	`AC.i`
*eq*14	`AC.n.i + G.p.i + R.n.i = 0`	*var*14	`G.p.v`
		*var*15	`G.p.i`

By following each alternating path and by eliminating the variables one by one from the eligibility set, we notice that eliminating only one of the variables from $\{var15, var4, var7, var13, var11\}$ corresponding to `{G.p.i, R.n.i, R.s, AC.i, AC.n.i}` will not disconnect the bipartite graph. Therefore, this reduced set will be further analyzed.

Based on similar reasoning for overconstrained situations, we can deduce that only $var7$ can safely be removed from the Modelica code to obtain a well-specified equation system. We will briefly explain why the removal of $var15$, $var4$, $var13$, and $var11$ has not been considered as a possible error-fixing solution to our underconstrained problem:

- Variable $var15$ (i.e., `G.p.i`). To remove variable $var15$ (i.e., `G.p.i`) from equation $eq14$ (`AC.n.i + G.p.i + R.n.i = 0`) at the intermediate code level, the con-

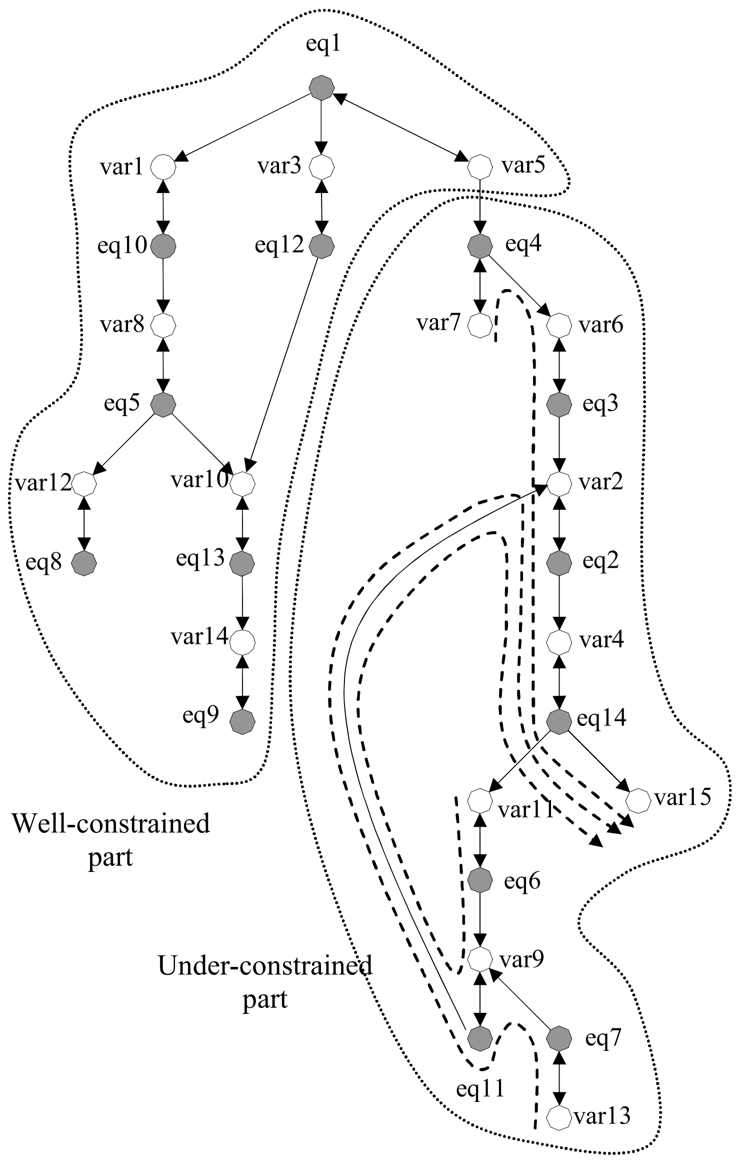


Figure 26. Directed graph corresponding to the underconstrained simple electrical circuit

nect equation `connect(AC.n, G.p)`, which connects the `Ground` element to the main circuit, needs to be removed. This implies as well the removal of equation *eq*10 (`AC.p.v = R.p.v`), an operation that will overconstrain the overall system.

- Variable *var*4 (i.e., `R.n.i`). The elimination of variable *var*4 (i.e., `R.n.i`) from equation *eq*14 (`AC.n.i + G.p.i + R.n.i = 0`) and *eq*2 (`0 = R.n.i + R.p.i`) implies the removal of the connect equation `connect(R.n, Ac.n)` from the original source code. This modification will trigger the removal of equation *eq*2 from the flattened system of equations, making the system overconstrained.

- Variable *var*13 (i.e., `AC.i`). Removing *var*13 (i.e., `AC.i`) from equation *eq*7 (`AC.i = AC.p.i`) is possible at the source code level by removing the variable `i` from the equation `i = p.i`, substituting `i` with a constant value from the `TwoPin` component. This modification will trigger at the intermediate code level the removal of *var*6 (i.e., `R.i`) from *eq*4 (`R.i * R.R = R.s * R.v`), which becomes `R.R = R.s * R.v`, and from *eq*3 (`R.i = R.p.i`), that becomes `const = R.p.i`. These modifications will disconnect the resulting bipartite graph.

- Variable *var*11 (i.e., `AC.n.i`). The elimination of variable *var*11 (i.e., `AC.n.i`) from equation *eq*14 (`AC.n.i`

`+ G.p.i + R.n.i = 0`) and $eq6$ (`0 = AC.n.i + AC.p.i`) implies the removal of the connect equation `connect(R.n, Ac.n)` from the original source code. This modification will trigger the removal of equation $eq2$ from the flattened system of equations, making the system overconstrained.

We call the set of variables obtained after performing the reasoning, based on variable annotations and filtering according to the language semantic rules, the *reduced eligibility set*. In our small example, the reduced eligibility set contains only one element: $var7$, which is exactly the variable introduced by us to underconstrain the system.

In the example presented above, the fault was detected by applying the first strategy, when debugging underconstrained systems implies the removal of a free variable node. However, if the user is not satisfied with the given solution or the reduced eligibility set is empty, the debugger can enter into the second stage, where possible connections of the adjacent equation nodes to those variable nodes that disconnect the bipartite graph are checked. If a possible coupling of a variable to those equations is found, the adjacent disconnecting variable node might also be considered for elimination.

When debugging underconstrained systems, a second strategy is used when extra equations need to be added and coupled to the free variable. For example, in our case, adding an extra equation `s = 10` in the `Resistor` class is a mathematically sound solution, even though if it might not reflect the modeler's intent. In a similar way, extra equations can be added for each variable from the eligibility set. The user has the possibility of specifying which strategy should be applied and which level of debugging he or she would like to perform on the erroneous model. In this way, some of the error messages can be filtered out, and incremental error fixing can be performed.

A general algorithm for debugging underconstrained systems can be given. It is composed of two distinct stages. First, the eligibility set corresponding to the underconstrained subgraph is computed. In the first stage, each variable from the eligibility set is checked to determine whether it disconnects the bipartite graph by elimination. If the variable does not split the graph, the elimination needs to be validated semantically by the function *validateSem*(var). The function *validateSem*(var) checks at the original source code level if the elimination of the variable *var* at the intermediate code level is possible by applying simple atomic changes to the source code.

Algorithm 3: Debugging underconstrained subsystems

Input: The underconstrained subgraph U_G^{1-} resulting after D&M decomposition has been performed on a graph G.

Output: List of all the possible error-fixing solutions.

```
begin:
    Find all the nodes $n \in v(G)$ that sink into the free
    variable node $v_{free}$ and insert them into the
    list $L$.
    Compute the eligibility set $L_{el} = \{v_1, \ldots, v_k \mid v_k \subset L$ and $v_k \in v_{V_2}(G)\}$
    //stage1: free variable elimination
    for each $v_k \in L_{el}$ do
        Hide $v_k$ from $U_G^{-1}$ and all the adjacent edges
        Compute the number of strongly connected
        components $no_{str}$ of $U_G^{1-}$.
        if $no_{str} == 1$ then
            //validate semantically the elimination of $v_k$
            from the adjacent equations
            if validateSem($v_k$) then
                output("remove $v_k$ from equation adj($v_k$)");
            end if;
        end if;
        Restore $v_k$ and all the adjacent edges
    end for;

    //stage2: add new equations
    for each $v_k \in L_{el}$ do
        classList = reach($v_k$);
        for each class $c \in classList$ do
            varList = reach(c);
            output("add new equation in class $c$ that must
            contain variable $v_k$");
            output("additional variables varList that can be
            included in the equation");
        end for;
    end for;
end.
```

The second stage is concerned with the addition of a new equation. The function *reach*() performs a reachability analysis. When this function is called with a parameter that is a variable, it returns the names of the classes in which the variable can exist. A new equation that contains the selected variable from the eligibility set can be added at the level of the returned classes. The second call to the function *reach*(), this time with a class name as a parameter, will return the set of variables that can be used and included in the new equation to be created. After consulting the set of all possible solutions, it is the user's responsibility to choose where the new equation should be added and which variables need to be included in the equation.

8. Automatic Static Analysis Module

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. AMOEBA is the static analysis module that we have designed and implemented to attach it to a Modelica compiler. The tool is able to successfully detect and provide error-fixing solutions for typical overconstrained and underconstrained situations, which might appear during the modeling stage using Modelica.

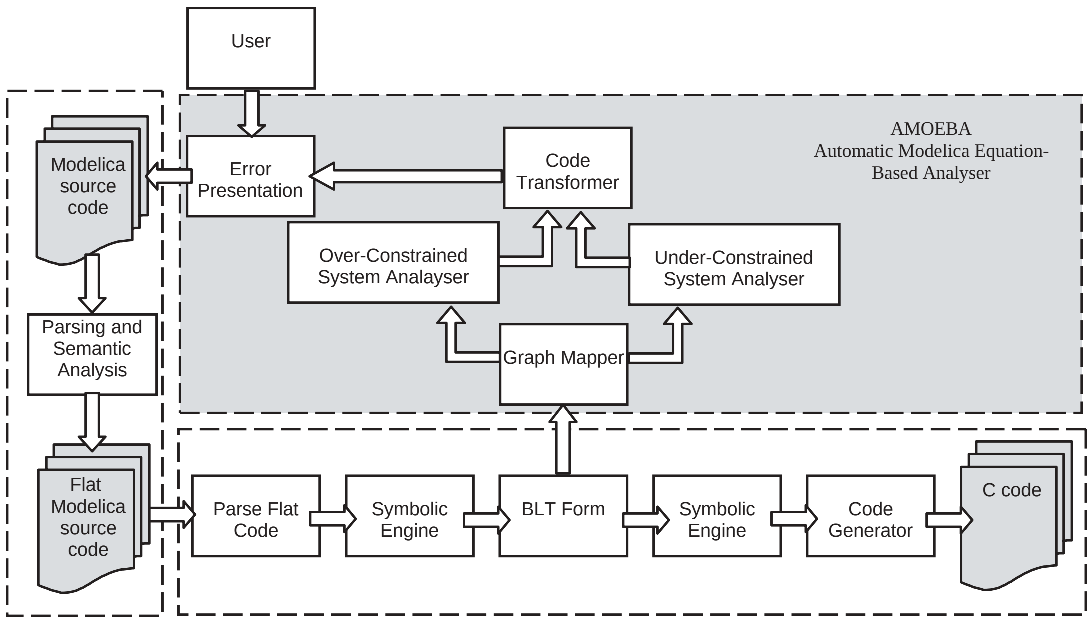

User
AMOEBA
Automatic Modelica Equation-Based Analyser
Modelica source code
Error Presentation
Code Transformer
Over-Constrained System Analayser
Under-Constrained System Analyser
Graph Mapper
Parsing and Semantic Analysis
Flat Modelica source code
Parse Flat Code
Symbolic Engine
BLT Form
Symbolic Engine
Code Generator
C code

Figure 27. AMOEBA (Automatic Modelica Equation-Based Analyzer) integration into the compilation framework

Figure 27 presents 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 overconstrained system of equations (too many equations are present), an underconstrained 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 underconstrained or overconstrained situations are detected, this means that there are some inconsistencies in the model.

The overconstrained and underconstrained system analyzers apply the algorithms presented in previous sections 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 ensure 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 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 equation 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 with the programmer's view of the model structure should be selected.

9. Summary and 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 preprocessing a system of equations pays high dividends by reducing the time for finding inconsistencies and efficiently correcting them. From the user's 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 equation system.

Nowadays, complex component-based simulation models are often specified with the help of programming languages such as logic/constraint or equation-based languages. Despite this close connection to programming languages, much of the literature on structural analysis primarily focuses on purely mathematical methods and the mathematical characterization of the modeled systems. Obviously, a connection between structural analysis and programming languages needs to be established to improve the debugging technology available for modeling and simulation environments.

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. We show how existing graph-theoretical decomposition techniques can be adapted and integrated into debugging tools integrated into simulation environments that employ such languages. We have tried to automate our debugger as much as possible and to keep the user interaction to a minimum. To achieve this goal, we propose an automated debugging framework. We claim that the techniques developed and proposed in this article 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.

The presented work was limited to static aspects of the debugging process—namely, the detection and debugging of overconstrained and underconstrained situations. Obviously, the next step would be to improve the already developed techniques by providing better user interaction and extending the debugger to handle dynamic situations.

10. References

[1] Fritzson, Peter, and Vadim Engelson. 1998. Modelica, a general object-oriented language for continuous and discrete-event system modeling and simulation. In *Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP'98)*, July, Brussels, Belgium.

[2] Elmqvist, Hilding, Sven Erik Mattsson, and Martin Otter. 1999. Modelica—a language for physical system modeling, visualization and interaction. In *Proceedings of the IEEE Symposium on Computer-Aided Control System Design*, August, Hawaii.

[3] Fritzson, Peter, and Peter Bunus. 2002. Modelica, a general object-oriented language for continuous and discrete-event system modeling and simulation. In *Proceedings of the 35th Annual Simulation Symposium*, April, San Diego, CA.

[4] Fritzson, Peter. 2003. *Principles of object-oriented modeling and simulation—with Modelica 2.1*. IEEE Press.

[5] Tiller, Michael M. 2001. *Introduction to physical modeling with Modelica*. Boston: Kluwer Academic.

[6] Modelica Association. 2002. *Modelica—a unified object-oriented language for physical systems modeling: Language specification version 2.0*. Modelica Association.

[7] Modelica Association. 2002. *Modelica—a unified object-oriented language for physical systems mmodeling: Tutorial version 2.1 and design*. Modelica Association.

[8] Fritzson, Peter, Peter Aronsson, Peter Bunus, Vadim Engelson, Levon Saldamli, Henrik Johansson, and Andreas Karström. 2002. The Open Source Modelica Project. In *Proceedings of the 2nd International Modelica Conference*, March, Munich, Germany.

[9] Duff, Iain S., and J. K. Reid. 1981. On algorithms for obtaining a maximum transversal. *ACM Transactions on Mathematical Software (TOMS)* 7 (3): 315-30.

[10] Duff, Iain S., and J. K. Reid. 1981. Algorithm 575: Permutations for a zero-free diagonal. *ACM Transactions on Mathematical Software (TOMS)* 7 (3): 387-90.

[11] Duff, Iain S., and J. K. Reid. 1978. An implementation of Tarjan's algorithm for the block triangularization of a matrix. *ACM Transactions on Mathematical Software (TOMS)* 4 (2): 137-47.

[12] Duff, Iain S., and J. K. Reid. 1978. Algorithm 529: Permutations to block triangular form. *ACM Transactions on Mathematical Software (TOMS)* 4 (2): 189-92.

[13] Health, Michael T. 2002. *Scientific computing: An introductory survey*. 2d ed. New York: McGraw-Hill.

[14] Brenan, Kathryn Eleda, Stephen L. Campbell, and Linda Petzold. 1996. *Numerical solution of initial-value problems in differential-algebraic equations*. Philadelphia: Society for Industrial & Applied Mathematics.

[15] Poulsen, Mikael Zebbelin. 2002. Structural analysis of DAEs. Ph.D. diss., Informatics and Mathematical Modelling, Technical University of Denmark.

[16] Deuflhard, Peter, and Folkmar Bornemann. 2002. *Scientific computing with ordinary differential equations*. New York: Springer Verlag.

[17] Bendtsen, Claus, and Per Grove Thompsen. 1999. Numerical solution of differential algebraic equations. Technical report IMM-REP- 1999-8, Department of Mathematical Modeling, Technical University of Denmark.

[18] Hopcroft, J. E., and R. M. Karp. 1973. An n(5/2) algorithm for maximum matchings in bipartite graphs. *SIAM Journal of Computing* 2 (4): 225-31.

[19] Fukuda, K. T., and T. Matsui. 1994. Finding all the perfect matchings in bipartite graphs. *Applied Mathematical Letters* 7 (1): 15-18.

[20] Uno, Takeaki. 1997. Algorithms for enumerating all perfect, maximum and maximal matchings in bipartite graphs. In *Proceedings of the Eighth Annual International Symposium on Algorithms and Computation*, December, Singapore.

[21] Uno, Takeaki. 2001. A fast algorithm for enumerating bipartite perfect matchings. In *Proceedings of the Twelfth Annual International Symposium on Algorithms and Computation*, December, Christchurch, New Zealand.

[22] Dulmage, A. L., and N. S. Mendelsohn. 1963. Coverings of bipartite graphs. *Canadian Journal of Mathematics* 10:517-34.

[23] Flannery, L. M., and A. J. Gonzalez. 1997. Detecting anomalies in constraint-based systems. *Engineering Applications of Artificial Intelligence* 10 (3): 257-68.

***Peter Bunus** is an assistant professor in the Department of Computer and Information Science at Linköping University, Sweden.*

***Peter Fritzson** is a professor in the Department of Computer and Information Science at Linköping University, Sweden.*