Linköping Studies in Science and Technology

Thesis No. 964

# Debugging and Structural Analysis of Declarative Equation-Based Languages

by

**Peter Bunus**


**INSTITUTE OF TECHNOLOGY**
**LINKÖPINGS UNIVERSITET**

Submitted to the School of Engineering at Linköping University in partial fulfillment of the requirements for the degree of Licentiate Engineering

Department of Computer and Information Science
Linköpings Universitet
SE-581 83 Linköping, Sweden

Linköping 2002

# Debugging and Structural Analysis of Declarative Equation-Based Languages

by

**Peter Bunus**

August 2002
ISBN 91-7373-382-2
Linköpings Studies in Science and Technology
Thesis No. 964
ISSN 0280-7971
LiU-Tek-Lic-2002:37

ABSTRACT

A significant part of the software development effort is spent on detecting deviations between software implementations and specifications, and subsequently locating the sources of such errors. This thesis illustrates that is possible to identify a significant number of errors during static analysis of declarative object-oriented equation-based modeling languages that are typically used for system modeling and simulation. Detecting anomalies in the source code without actually solving the underlying system of equations provides a significant advantage: a modeling error can be corrected before trying to get the model compiled or embarking on a computationally expensive symbolic or numerical solution process. The overall objective of this work is to demonstrate that debugging based on static analysis techniques can considerably improve the error location and error correcting process when modeling with equation-based languages.

A new method is proposed for debugging of over- and under-constrained systems of equations. The improved approach described in this thesis is to perform the debugging process on the flattened intermediate form of the source code and to use filtering criteria generated from program annotations and from the translation rules. Each time when an error is detected in the intermediate code and the error fixing solution is elaborated, the debugger queries for the original source code before presenting any information to the user. In this way, the user is exposed to the original language source code and not burdened with additional information from the translation process or required to inspect the intermediate code.

We present the design and implementation of debugging kernel prototypes, tightly integrated with the core of the optimizer module of a Modelica compiler, including details of the novel framework required for automatic debugging of equation-based languages.

This thesis establishes that structural static analysis performed on the underlying system of equations from object-oriented mathematical models can effectively be used to statically debug real Modelica programs. Most of our conclusions developed in this thesis are also valid for other equation-based modeling languages.

*This work has been supported by the EU Realsim (Real-time Simulation for Design of Multi-physics Systems) project ,the Vinnova VISP (Virtuell Integrerad Simuleringsstödd Produktframtagning) project, MathCore AB, KK-stiftelsens företagsforskarskola i Linköping and the ECSEL (Excellence Center for Computer Science and Systems Engineering in Linköping) Graduate School.*

Department of Computer and Information Science
Linköpings universitet
SE-581 83 Linköping, Sweden

# Acknowledgements

In a thesis such as this, one name appears on the cover but this work only exists because of the effort of others behind the scenes. I will take this opportunity to thank to those who have contributed in one way or another to fulfill this thesis and make my wish come true.

First of all, I would like to thank my supervisor, Peter Fritzson, who has supported me along the way, for sharing his thoughts with me and suggesting the way forward at several points in my research. I am deeply indebted to him for his invaluable help, support and encouragement throughout my work and for countless hours that he dedicated to this thesis.

Many thanks also to my colleagues at PELAB (Programming Environment Laboratory), for many rewarding discussions and for contributing to the stimulating and pleasant environment. A special thanks goes to the "*pelab-modelica*" group members for contributing to many invaluable ideas. In particular, I would like to thank Peter Aronsson, Vadim Engelson, Iakov Nakhimovski and Levon Saldamli with whom I have had many stimulating discussions. Vadim Engelson deserves also recognition for reasons too numerous to list here.

This work would not have been possible without the support of MathCore AB. I would like to thank everybody who has been closely involved in my research at MathCore: Mikael Adlers, Jan Brugård, Johan Gunnarsson, Andreas Idebrant, Mats Jirstrand, Henrik Johansson, Andreas Karström, Yelena Turetskaya and Kristina Swenningsson.

A very special thanks goes to Bodil Mattsson Kihlström and Heléne Thibblin for helping me to handle the exponential grow in size and complexity of administrative tasks and supporting me in many ways. Furthermore, I am grateful to the administrative staff at IDA, in particular Lillemor Wallgren, Britt-Inger Karlsson, Bodil Carlsson and Helene Nordahl.

This work has been supported by the EU Realsim (Real-time Simulation for Design of Multi-physics Systems) project, the Vinnova VISP (Virtuell Integrerad Simuleringsstödd Produktframtagning) project, MathCore AB, KK-stiftelsens företagsforskarskola i Linköping, and the ECSEL (Excellence Center for Computer Science and Systems Engineering in Linköping) Graduate School.

Last but not least, I am in debt to my wife, Kati, who suffered most during these years of my overtime work. I thank her for all her support, understanding and love. I should now promise her that I will never write another thesis. However, I am afraid that I need to write one more thesis. Therefore her suffering will continue for some time.

Peter Bunus

Linköping
3 August 2002.

# Table of Contents

# Table of Figures

# List of Tables



# List of Algorithms

# Notation

The following notation is employed in this thesis:

| | |
|---|---|
| $G = (V, E)$ | Normal undirected graph with nodes set *V* and edge set *E*. |
| $G = (V_1, V_2, E)$ | Bipartite graph with bipartitions $V_1$ and $V_2$ and edge set *E*. (*see Definition 4-1 page 38*) |
| $\mathbf{M}(G)$ | The incidence matrix of *G*. (*see Definition 4-2 page 38*) |
| $v_{V_1}(G)$ | The vertex set $V_1$ of a graph G. $v_{V_1}(G) = V_1$. |
| $v_{V_2}(G)$ | The vertex set $V_2$ of a graph G. $v_{V_2}(G) = V_2$. |
| $v(G)$ | The vertex set of a graph G. In the case of a bipartite graph . $v(G) = v_{V_1}(G) + v_{V_2}(G) = V_1 + V_2$. |
| $\varepsilon(G)$ | The edge set of a graph G. In the case of a bipartite graph $\varepsilon(G) = E$ |
| $\varepsilon_G(u, v)$ | The set of edges joining vertex *u* to vertex *v* in graph *G*. |
| $M_G^P$ | Perfect matching associated to the bipartite graph *G*. (*see Definition 4-8 page 39*) |
| $M_G^{\max}$ | Maximum cardinality matching associated to the bipartite graph *G*. *(see Definition 4-4 page 39)* |
| $M_G^k$ | matching with *k* edges associated to the bipartite graph *G*. *(see Definition 4-3 page 39)* |
| $\Im(M_G^P)$ | The family of all perfect matchings associated to the bipartite graph *G* *(see Definition 4-15 page 40)* |
| $\lvert \Im(M_G^P) \rvert$ | The number of all perfect matchings associated to the bipartite graph *G* |
| $w(M_G)$ | The weight of a matching. *(see page 109)* |
| $V_1 = \{v_1, v_2, \ldots, v_k\}$ | $v_1, v_2, \ldots, v_k$ vertices of the first bipartition of *G*. (*see Definition 4-1 page 38*) |
| $V_2 = \{u_1, u_2, \ldots, u_k\}$ | $u_1, u_2, \ldots, u_k$ vertices of the second bipartition of *G*. (*see Definition 4-1 page 38*) |
| $E = \{(u, v) \mid u \in V_1; v \in V_2\}$ | The set of edges corresponding to the bipartite graph *G*. |
| $\vec{E} = \{(\overrightarrow{u, v}) \mid u \in V_1; v \in V_2\}$ | The set directed edges from bipartition $V_1$ to bipartition $V_2$. |
| $\overleftarrow{E} = \{(\overleftarrow{u, v}) \mid u \in V_1; v \in V_2\}$ | The set directed edges from bipartition $V_2$ to bipartition $V_1$. |
| $\overleftrightarrow{E} = \{(\overleftrightarrow{u, v}) \mid u \in V_1; v \in V_2\}$ | The set of bidirectional edges $\overleftrightarrow{E} = \vec{E} \cup \overleftarrow{E}$. |

$\overline{E} = \{(\overline{u,v}) \mid \exists(\overrightarrow{u,v}) \subseteq \overrightarrow{E}; \exists(\overleftarrow{u,v}) \subseteq \overleftarrow{E}\}$ the set of directed edges .

$\overrightarrow{G} = (V_1, V_2, \overrightarrow{E})$ Directed bipartite graph with edge orientation from $V_1$ to $V_2$ .

$\overleftarrow{G} = (V_1, V_2, \overleftarrow{E})$ Directed bipartite graph with edge orientation from $V_2$ to $V_1$.

$\overleftrightarrow{G} = (V_1, V_2, \overleftrightarrow{E})$ Directed bipartite graph with all the edges bidirectional.

$\overline{G} = (V_1, V_2, \overline{E})$ Directed bipartite graph.

$\overline{\overline{(u,v)}}$ A matching edge $(u,v) \in \varepsilon(M_G)$.
(*see Definition 4-5 page 39*)

$P = \{\overline{\overline{(u_1, v_1)}}, (v_1, u_2), \overline{\overline{(u_2, v_2)}} \cdots \overline{\overline{(u_k, v_k)}}\}$ representation of an alternating path.
(*see Definition 4-10 page 39*)

adj($v$) The list of adjacent nodes of a vertex *v*. (*see Definition 4-6 page 38*)

$inc_E(v)$ The list of incident edges with vertex *v*. (*see Definition 4-6 page 38*)

target($e$) The target node of a directed edge *e*.

source($e$) The source node of a directed edge ***e*.**

$u \xrightarrow{*} v$ A path from vertex *u* to vertex *v*.
(*see Definition 4-9 page 39*)

$u \xrightarrow{=} v$ An alternating path from vertex *u* to vertex *v*.
(*see Definition 4-10 page 39*)

$O_G^{k+}$ Over-constrained subgraph associate to the bipartite graph *G* with *k* nodes not covered by the maximum matching $M_G^{\max}$ .
(*see Algorithm 4-1 page 44*)

$U_G^{k-}$ Under-constrained subgraph associate to the bipartite graph *G* with *k* nodes not covered by the maximum matching $M_G^{\max}$ .
(*see Algorithm 4-1 page 44*)

$W_G$ Well constrained subgraph associate to the bipartite graph *G*.
(*see Algorithm 4-1 page 44*)

deg($v$) Degree of vertex v, the number off edges incident to that vertex.
(*see Definition 4-12 page 39*)

# Chapter 1

# Introduction

***Summary**: This introductory chapter provides a general overview of this thesis on the topic of debugging declarative object-oriented equation-based languages. The research problem is identified, followed by a formulation of the main research objective. The main contributions of the thesis are also stated and the content of subsequent chapters is briefly sketched.*

## 1.1 Introduction to Modeling and Simulation Environments

Simulation models are increasingly being used in problem solving and decision-making since engineers, when building new products, need to analyze and understand complex and heterogeneous physical systems. In advanced equation-based simulation environments that employ equation-based languages, users create models through a graphical user interface (GUI) or by writing custom modeling source code.

The facility of using an integrated GUI in the form of a *Model Editor* (see Figure 1-1) allows simulation practitioners and knowledge engineers to express problems in terms 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* is the selection of components from ready-made libraries, to connect components in model diagrams, and to enter parameter values for different components. More complex simulation models can be built by simply 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 towards a specific domain by giving modelers access to common model elements and terminology from that domain.

Instead of using a GUI, certain applications and most library models are written with the help of a special kind of programming languages called mathematical modeling languages. In order to support mathematical modeling and simulation, a number of object-oriented and/or declarative acausal modeling languages have emerged. The advantage of an acausal modeling language over traditional languages is that the user can

concentrate on the logic of the problem rather than on a detailed algorithmic implementation of the simulation model. The models can be created in a textual environment and then transferred to the model editor for a graphical view of the model, or vice versa. The ability of specifying user-defined libraries of reusable components in a modeling language and its graphical representation supports model management and evolution of models of large systems.

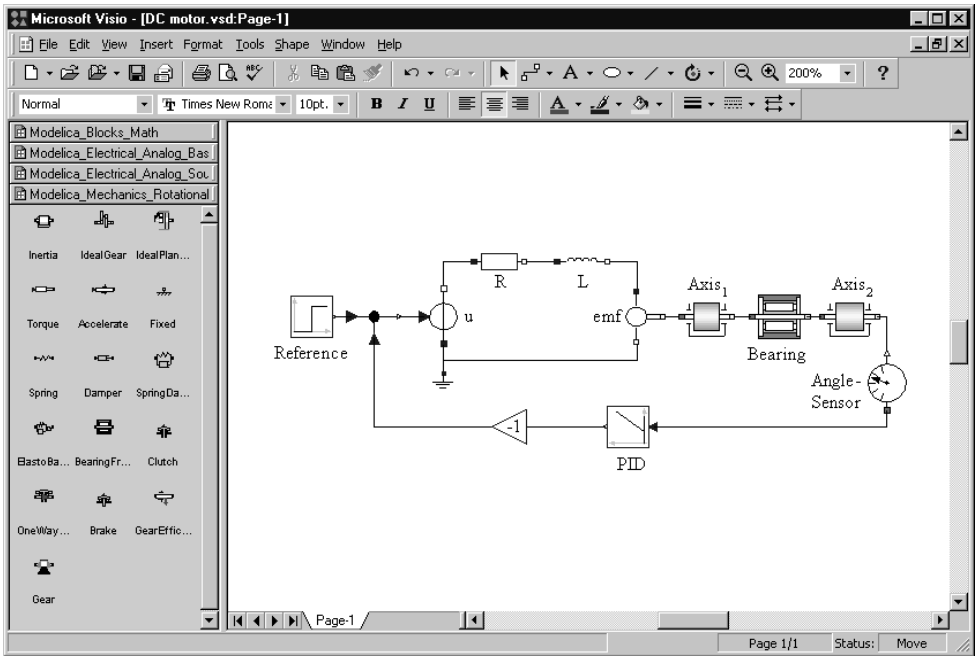



**Figure 1-1.** Screen shot of the Model Editor of the MathModelica simulation environment.

Based on the two kinds of interaction with the modeling and simulation environment we can differentiate between two distinct categories of users:

- Application developers who mostly interact with the environment through a graphical user interface such as a model editor.
- Library developers or advanced users who extend the simulation capabilities of the environment by writing new library models specified in a modeling language.

## 1.2 Overview

The high level of abstraction of equation-based models presents new challenges to modeling and simulation tools, because of the large gap from the declarative specification to the executable machine code. The process of translating models to efficient code becomes considerably more involved. This abstraction gap leads to difficulties in finding and correcting model inconsistencies and errors, 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 equations are specified in the system, leading to an inconsistent state of the simulation model. In such situations numerical solvers fail to find correct solutions to the underlying system of equations. The user should be able to deal with over-determined systems of equations by identifying the minimal set of equations that should be removed from the system in order to make the remaining set of equations solvable. For example, let us consider a physical system simulation model specified in a declarative object-oriented equation-based modeling language that consists of several hundreds of classes resulting in several thousands of equations. However, one of these equations over-constrains the overall system making it impossible to simulate. It can easily be imagined that, if a static debugger presents a small subset of over-constraining equations, from where the user can select the equation that needs to be eliminated from the overall model in order to form a structurally well posed simulation, this can greatly reduce the amount of time required to get the simulation working.

Currently there are essentially no advanced tools that can handle the debugging of equation-based languages at the source code level and provide useful error fixing solutions. The aim of the research presented in this thesis is to improve considerably the situation, especially with respect to debugging the Modelica language. In fact, the research presented is one of the first major efforts to solve this problem.

To address this need, in this thesis we propose a methodology for declarative debugging of equation-based languages by adapting graph decomposition techniques for reasoning and performing structural analysis of the underlying systems of equations. Detecting anomalies in the source code without actually solving the underlying system of equations and constraints provides a significant advantage: a modeling error can be corrected before embarking on a computationally expensive symbolic or numerical solution finding process and get the model through the compiler. Another problem to be addressed is to map error conditions in the executing simulation code or in the intermediate code back to the places in the model where the problem originated. This is currently problematic mostly due to the abstraction gap mentioned earlier. A model part may influence multiple places in the executable code due to several transformational stages in the translation process.

Our effort has also been targeted to automate as much as possible the debugging process and the software maintenance tasks involving declarative equation-based languages. In this thesis we have restricted our approach to static analysis even if the debugging process also involves a dynamic part. This is our first step in providing a complete debugging environment for the Modelica language. Debugging performed during static analysis has also been a choice influenced by efficiency criteria. Usually in a large simulation model, compiling the model is computationally expensive and most of the information that might be used to derive useful error messages has already been lost during the translation and optimization phases of the compilation process.

We have also tried to attach user-oriented program analysis modules to an existing simulation environment for the purpose of program understanding. Visualizing data structure abstractions and transformation graphs related to the transformation and compilation process turns out to be very useful for understanding the behavior and properties of complex simulation models. The program understanding modules have also

given useful hints on how to improve future versions of the Modelica compiler, especially the optimizing part of the compiler.

Our intent was to target all the user categories of the Modelica language environments by proposing a multilevel debugging environment. Error fixing solutions can be provided at:

- The diagram model level for end-users.
- The source code level for library developers.
- The intermediate code level for researchers and Modelica compiler developers by visualizing the data structures and graph-based abstractions.

By achieving this, the developed tool has turned out to be useful for detecting and automatically debuging over and under-constrained situations for software maintenance and restructuring tasks of the existing model libraries as well as for compiler construction related research purposes.

The main challenge has been to integrate a debugger as closely as possible into existing compiler architectures and to operate on the same intermediate code as the compiler. The integration of the debugger at the intermediate code level has made possible the automated debugging for certain classes of erroneous situations especially for certain over-constrained systems. When automatic debugging is not possible, obviously user intervention is necessary, which is achieved by presenting several error-fixing solutions to the user where the presentation is prioritized on the basis of annotations and language semantics filtering schemes and algorithms.

## 1.3 The Research Objective

The overall objective of this work is to demonstrate that debugging based on static analysis techniques can considerably improve the error finding process when modeling with object–oriented declarative equation-based languages. In order to realize this objective it is necessary to investigate new methods and to adapt traditional debugging techniques to the debugging of such languages. The developed methods should be independent of the choice of equation-based language. However, our first target language is Modelica and its associated simulation environments.

We also formulate a general statement of the thesis that will be motivated in the following chapters:

> *Simulation practitioners and programmers can perform faster model development if a tool is provided that identifies erroneous models and provides debugging alternatives. Such a tool will also increase the understandability and acceptance of the language.*

In (Ducassé and Noyé 1994 [30]) it is emphasized that users are reluctant to use a programming language without an appropriate programming environment. Even though many currently available equation-based modeling languages have been reported in the literature, most of the associated programming environments give little or no attention to the debugging problem. In particular there is no support for structural analysis techniques for debugging and program understanding. Use of inappropriate models or

model parameters may result in erroneous simulation results. Many of the modeling equation-based languages such as Modelica have been developed recently and sometimes need to follow a long acceptance process from the technical communities that they target. With that in mind, we have tried to enhance the debugging technology of the Modelica language, first by solving problems that are usually encountered during static analysis. Support for run-time debugging of executing simulations including the numerical solvers will be the next obvious step in providing a better interactive environment for the Modelica language.

Since the concept of debugging equation-based languages in the context of simulation environments is new, the greatest challenge for the thesis is to find new abstraction criteria for this special debugging process and to adapt well-known abstractions from the debugging community such as program slicing, data flow or control flow. The acausality of the equation-based languages makes such criteria hard to find or hard to adapt. Old criteria are mostly based on some sort of flow in the language. Some results presented in this thesis are very well known in graph theory but seem to have been ignored in the debugging community. We have tried to adapt those results to serve the goal of helping the debugging environment in providing useful error messages to the users.

The objective of this thesis is to bring the latest advances from the areas of debugging and program analysis to develop a powerful framework that will permit the integration of debugging tools into equation-based simulation environments. These debugging tools should be easy to use for a variety of end users of the simulation systems, ranging from normal application-oriented users to advanced library developers.

## 1.4 Contributions

The proposed debugging approach and debugging framework allows more complex applications to be simulated. We show how, by taking advantage of the simulation problem structure, it is often possible to extract additional information that can be used for debugging purposes.

The direct contributions of this thesis are as follows:

- The thesis illustrates that it is possible to identify a significant number of errors during static analysis of declarative object-oriented equation-based modeling languages. In this way certain numerical failures can be avoided later during the execution process.
- A new method is proposed for debugging of over- and under-constrained systems of equations. The improved approach described in this thesis is to perform the debugging process on the flattened intermediate form of the source code and to use filtering criteria generated from program annotations and from the translation rules itself. An important advantage of both of the implemented debugger prototypes is that they operate on the same intermediate form of the source code as the one used by the compiler, and therefore the analyzed program does not need be recompiled for debugging purposes. If a unique solution to the over-

constraining and under-constraining problems is found, automatic error fixing is possible.

- The properties of over-constrained bipartite graphs with multiple sources are analyzed from the mathematical and combinatorial point of view. The general debugging framework is further improved by employing several structural analysis techniques.
- In the case of under-constrained systems level-based debugging approach is proposed where the user can select different analysis levels when analyzing error fixing solutions for such systems.
- Development of an integrated debugging kernel tightly integrated with the core of the optimizer module of the compiler for supporting the Modelica-based modeling and simulation environments. Nevertheless, most of our conclusions developed in this thesis are valid for other equation-based modeling languages.
- This thesis establishes that the result of structural static analysis performed on the underlying system of equations can effectively be used to statically debug real programs.

The main contributions of this thesis are further discussed and compared to the related work in Chapter 11.

## 1.5 Thesis Outline

Given the overview and the goal of this work, the rest of this thesis is organized as follows:

*Chapter 2* starts with a discussion regarding debugging of programming languages and associated environments with an emphasis on algorithmic automated techniques developed in recent years. The particularities of debugging declarative equation-based languages are also given and the main characteristics that a debugging tool should satisfy are presented.

*Chapter 3* presents some important language characteristics and compilation techniques that are typical to declarative object-oriented languages. A brief survey of a number of declarative object-oriented equation-based languages is given, before presenting Modelica in more detail. Modelica is gently introduced by small modeling and simulation examples, models which will constitute the benchmarks for evaluating the proposed debugging techniques in the following chapters.

*Chapter 4* discusses the basic concepts of graph and combinatorial theory used in the thesis. The key concepts are defined, such as bipartite graphs, directed bipartite graphs, adjacency matrices, and matchings in bipartite graphs. The canonical decomposition algorithm for bipartite graphs is introduced here. Several graph theoretic algorithms are also introduced. A detailed knowledge of concepts in this chapter is a prerequisite for the following chapters.

*Chapter 5* provides some algorithms for detecting and debugging over-constrained situations that arise during the modeling phase with equation-based languages. It

mainly describes graph-based algorithms for detecting over-constrained situations. The combinatorial explosion of the potential solutions found by the debugger at the intermediate code level is filtered by annotations and semantics constraints generated during the translation phase of the language. Several over-constrained simulation examples are given in order to illustrate the applicability of the proposed debugging algorithms.

*Chapter 6* introduces the debugging of under-constrained simulation models by presenting algorithms that handle those situations. The combinatorial explosion of the number of solutions found by a debugger attached to the intermediate code level is even greater than in the case of over-constrained situations. For this reason automatic algorithmic debugging is greatly obstructed in these situations and enhanced user interaction is necessary. Debugging of under-constrained systems at several depth levels is introduced.

*Chapter 7* extends the debugging framework developed in the previous two chapters by introducing structural analysis methods for handling differential algebraic equations of higher index. Some shortcomings of the structural analysis methods employed for approximating the differential index of differential algebraic equations are also presented here.

*Chapter 8* presents the overall architecture of the two implemented debuggers and how these debuggers can be integrated into different simulation environments. Details on particular developed kernels are also given here.

*Chapter 9* briefly surveys related work. This chapter is divided into two parts: the first part surveys related work from the constraint programming community when the emphasis is on partial satisfaction of constraints and in detection of over- and under-constrained problems. The second part surveys related work done in the area of structural analysis, diagnosability of object-oriented modeling systems, as well as debugging and verification modules attached to simulation environments.

*Chapter 10* presents an evaluation of our debugging framework based on existing usability criteria developed for algorithmic automated debugging.

*Chapter 11* finally summarizes the conclusions of the research, the problems left unanswered, and the future work.

## 1.6 Origins of the Chapters

Many of the chapters in this thesis are revised versions of publications that have appeared elsewhere:

1. Bunus Peter and Peter Fritzson. “The Need for Debugging Tools for Declarative Equation Based Simulation Languages”. In *Proceedings of the 2000 Summer Computer Simulation Conference* (Vancouver, B.C. Canada, Jul. 16-20, 2000).
2. Bunus Peter and Peter Fritzson. “DEVS-based Multi-Formalism Modeling and Simulation in Modelica”. In *Proceedings of the 2000 Summer Computer Simulation Conference* (Vancouver, B.C. Canada, Jul. 16-20, 2000).

3. Bunus Peter, Vadim Engelson. and Peter Fritzson. ”Mechanical Models Translation, Simulation and Visualization in Modelica”. *In Proceedings of Modelica Workshop 2000* (October 23-24 Lund, Sweden).
4. Bunus Peter and Peter Fritzson. “Applications of Graph Decomposition Techniques to Debugging Declarative Equation Based Languages”. In *Proceedings of the 2001 SIMS Conference* (Telemark University College, Porsgrunn, Norway, Oct. 8-9, 2001).
5. Bunus Peter and Peter Fritzson. ”An Interactive Environment for Debugging Declarative Equation Based Languages”. In *Proceedings of the International Workshop on User-Interaction in Constraint Satisfaction*. (Paphos, Cyprus, Dec 1, 2001)
6. Bunus Peter and Peter Fritzson. ”A Debugging Scheme for Declarative Equation Based Languages” In *Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages*. (Portland, OR, USA, January 2002) LNCS 2257, Springer Verlag, 2002
7. Fritzson Peter and Peter Bunus ”Modelica, a General Object-Oriented Language for Continuous and Discrete-Event System Modeling and Simulation." In Proceedings of the 35th Annual Simulation Symposium (San Diego, California, April 14-18, 2002)
8. Bunus Peter and Peter Fritzson. "Methods for Structural Analysis and Debugging of Modelica Models." In *Proceedings of the 2nd International Modelica Conference* (March 18-19, Munich Germany). 2002.
9. Fritzson Peter, Peter Aronsson, Peter Bunus, Vadim Engelson, Henrik Johansson, Andreas Karström, Levon Saldamli. "The Open Source Modelica Project" In *Proceedings of the 2nd International Modelica Conference* (18-19 March, Munich Germany) 2002.
10. Fritzson Peter and Peter Bunus. "Modelica – A Declarative Object Oriented Multi-Paradigm Language" In *Proceedings of the Workshop on Multiparadigm Programming with Object Oriented Languages (MPOOL'02)* (June 11, Malaga, Spain 2002).
11. Bunus Peter and Peter Fritzson. "Semantics Guided Filtering of Combinatorial Graph Transformations in Declarative Equation Based Languages". To appear in *Proceedings to 2nd International Workshop on Source Code Analysis and Manipulation* SCAM2002 (Montreal Canada, October 1st 2002).
12. Bunus Peter and Peter Fritzson. "Algorithmic Automated Debugging for Declarative Equation Based Languages." Paper draft to be submitted to the *International Workshop of Automated and Algorithmic Debugging* (AADEBUG 2003)

# Chapter 2

# The Need for Debugging

*"Computer science still seems to be looking for the magic bullet that will cause people to write correct programs without having to think. Instead, we need to teach people how to think"*

Kirk L. Kroeker COMPUTER Vol. 32, No. 5: MAY 1999, pp. 48-57 Software Revolution: A Roundtable

***Summary****. In this chapter we briefly survey the debugging process and establish the need for debugging of declarative equation-based languages. We also survey some existing techniques and usability criteria developed especially for automated debugging systems. The motivation for integrating different debugging functionalities, techniques and new methods in a programming environment for declarative equation-based languages are briefly discussed. This is also our first attempt to sketch the main characteristics and properties of such debugging tools.*

## 2.1 Introduction

A significant part of the software development effort is spent on detecting deviations between software implementations and specifications, and subsequently localizing the sources of such errors. Several studies have been conducted to see how much time goes into debugging. According to (Parasoft 1997 [94]) the debugging phase of software development takes 60-70% of the overall development time and debugging is responsible for 80% of all software projects overruns. In (Robson et. al. 1991 [101]) it is indicated that programmers spend 50-90% of their debugging time comprehending existing programs. Other studies indicate that the debugging problem is constantly ignored by the computer science community (Lieberman 1997 [77]). Based on these facts we should attach substantial importance to the debugging phase during all the stages of the software development process. This will require the development of new programming environment tools for program comprehension as well as new and improved debugging techniques. Recently developed debugging tools would be especially important and beneficial for long-term improvement of the software-development process.

Sophisticated engineering systems are inherently complex. Mathematical modeling and simulation of complex physical systems is emerging as a key technology in engineering. Modern approaches built on acausal equation-based methodologies and object-oriented constructs facilitate reusability of modeling knowledge. Declarative equation-based languages and their associated modeling and simulation environments represent an emerging software technology that provides an original approach to simulation. Efficient and flexible simulation of physical systems is achieved by combining implementations of library components in the same formalism. Such languages have been designed to allow automatic generation of efficient simulation code from declarative specifications.

A major objective is to facilitate exchange of models, model libraries, and simulation specifications. Equation-based declarative programming presents new challenges in the design of programming environments. In order for these languages to achieve widespread acceptance, associated programming environments and development tools must become even easier to use. Inevitably, the use of these languages adds several difficulties to the debugging problem, difficulties that are briefly summarized in the following sections.

## 2.2 The Debug Paradigm

In (IEEE Std 610.12-1990 [62]) the following definition of the debug process can be found:

> ***Debug***. *To detect, locate, and correct faults in a computer program. Techniques include the use of breakpoints, desk checking, dumps, inspection, reversible execution, single-step operations, and traces.*

As can be seen from the above definition, debugging is a three-stage process. The first stage in the debugging process is fault detection, an operation that is mainly achieved with the help of software testing (Myers 1979 [89]) or formal verification. Obviously the second stage is to correctly locate the bug, which is not an easy task mostly due to the cause-effect gap between the time when an error occurs and the time when the error becomes apparent to the programmer (Lencevicius 2000 [76]). Another problem is to map the error back to the source code from where it originated. For example, source level debuggers of optimized code are often unable to report the expected values of a source variable at the breakpoint location (Jaramillo et al. 2000 [63]). The last stage in the debugging process is obviously fault correction when a located fault needs to be corrected in order to accommodate the system behavior according to its specification.

Several methodology paradigms have been proposed for each of the debugging stages mentioned above. In this thesis we propose a complete debugging framework that will handle all the stages mentioned above for debugging declarative equation-based languages. The proposed debugging framework in this thesis is based on automatic debugging techniques with the main goal to keep user interaction to a minimum.

### 2.2.1 Automated Debugging Techniques

The concept of algorithmic automated debugging originates from the declarative logic programming community (Shapiro 1982 [110]). Automated debuggers employ some kind of knowledge in order to successfully carry out the debugging process (Ducassé 1993 [29]). This is based on the nature of the knowledge and on the strategy that is employed. In (Ducassé 1993 [29]) algorithmic debuggers are classified into three main categories: verification with respect to specifications, checking with respect to language knowledge, and filtering with respect to a symptom.

- *Verification with respect to specification*: this strategy compares the actual program with some formal specification of the intended program. The main disadvantage of this approach is the missing formal specification for most of the programs even if in recent years enormous progress has been made to automate the process of formulation program specifications by using machine learning approaches to discover formal specifications in the original code (Ammons et al. 2002 [4]). The extracted specifications can be used later by automatic verification tools to find bugs. An example of a related system is the SLAM project (Ball and Rajamani 2002 [11]), (Ball and Rajamani 2001 [10]) that checks temporal safety properties of sequential C programs and require that the checked properties are encoded in a language called SLIC (Specification Language for Interface Checking).
- *Checking with respect to language knowledge*. This technique is based on automatic tools that parse the program under analysis and searches for language dependent errors. The disadvantage of this strategy is that it does not assume any code to be correct. Therefore all the program code needs to be checked, which can be extremely costly for large programs, in terms of computer time. Another disadvantage is that this technique only relies on knowledge of the programming language. Subtle bugs, primarily those that appear during execution, are not detected.
- *Filtering with respect to symptom.* This strategy is effective in reducing the amount of analysed code. Program slicing is a well known technique to reduce the search space for program errors and focus the debugging process on the parts of the program that are influenced by the bug symptom (Weiser 1982 [122]). This technique is effective for imperative languages (Kamkar 1993 [73]) and well suited for declarative languages, e.g. successful implementation of program slicing have been reported in the literature for logic programming languages (Szilágyi et. al. 2002 [112]) where dependencies in logic programs and proof trees are defined in terms of slices. Execution traces and proof trees have been successfully adapted to the debugging of lazy functional languages by using Evaluation Dependence Trees (EDT) that abstract from the operational details such as evaluation order, instead emphasizing the source code structure of the program (Nilsson 1998 [90]).

### 2.2.2 Usability Criteria for Automated Debugging

In (Shahmehri et. al. 1995 [109]) the classification of automated debugging systems is based on and is influenced by the type of information given to the debugger (intended and actual program, I/O data, subject language, common bugs, application domain and general programming expertise) and as well as the type of tasks performed by the automated debugger (test generation, bug detection, bug localization, bug explanation, bug correction). It is also mentioned that no existing automated debugging system uses all these kinds of knowledge. The situation has not changed very much since the publication of the usability criteria from (Shahmehri et. al. 1995 [109]) even if highly automated debugging systems have appeared latter (Ball and Rajamani 2002 [11]), (Ball and Rajamani 2001 [10]), (Zeller and Hildebrandt [127][85]), (Cleve and Zeller 2000 [26]) (Lencevicius 2000 [76]).

The following four user-oriented usability criteria for automated debugging systems have been identified in (Shahmehri et. al. 1995 [109]).

- *Generality*. The generality property of automated debugging systems state that an automatic tool should not be limited only to a specific type of bug, it should provide error detection and error fixing solutions for a large variety of bugs.
- *Cognitive plausibility*. The debugging process should follow the user's mental model of the program under analysis. Studies conducted on cognitive processes of programmers engaged in software debugging have revealed multiple-level backward chaining mechanisms for goal management. A three-level cognitive process model is accepted by the above-mentioned studies (Hale et. al. 1999 [55]) (Hale and Haworth 1991 [56]).

  - *First level problem solving*. This level only involves the execution of simple rules to directly eliminate a program bug once the error has been located. First the program is evaluated and a rule that will eliminate an observed error is generated. At the second stage of this first level, the correction is implemented and the program is verified to see if the error has been successfully eliminated
  - *Second level problem solving*. This level is triggered when it is not possible to find a rule to be executed. In this case a secondary goal needs to be established instead of the primary goal. The secondary goal is refined and established by automatically acquiring new knowledge about the problem that will then allow the application of a direct rule.
  - *Third level problem solving*. This level is executed if all the actions plans used for defining the secondary goal at the second level have failed to produce a direct rule. At this stage new information is gathered that would help to create new action plans necessary for the second level problem solving

  The study has also revealed the recursive manner in which programmers engaged in the debugging process attack those problems for which solutions cannot be diagnosed directly. Automated debugging tools should help users to move gradually from the third level of problem solving to the first level.

- *Degree of automation.* Automating the process of bug finding by keeping the user intervention to a minimum will very much enhance the practicality of the debugging approach. The user should be consulted only when it is necessary to eliminate ambiguous situations.
- *Appreciation of user expertise.* Users of software systems usually have different levels of expertise. An automatic debugging tool should address both beginners and advanced users. For both user categories it is also necessary and useful to provide detailed explanations for the encountered bugs, not just only error fixing solutions. In this way the software debugging process can be transformed into a learning and a tutoring session that will provide the user with the necessary information and knowledge that will help him/her to avoid the same error in the future.

We will evaluate our developed debugging tool in Chapter 10, based on the above mentioned usability criteria.

## 2.3 The Equation-Based Debugging Paradigm

Traditional approaches to debugging are inadequate and inappropriate for solving the error location problem in equation-based languages. The fundamental problem is that conventional debuggers and debugging techniques are based on observation of execution events as they occur. For example, most of the commercially available debuggers implement the breakpoint paradigm giving the user the possibility of stopping the execution of a program and eventually continuing the execution step by step through the control flow of the language (Rosenberg 1996 [102]). This method is especially effective for those bugs that are immediately manifested after the execution of the faulty statement. The acausality of equation-based languages renders the breakpoint method inappropriate for such languages. In the same manner, the declarativeness of such languages eliminates the use of program execution traces as a debugging guide.

When attempting to debug equation-based languages, most environments give some misleading information about the underlying equations or variables of the simulation model. The semantics of equation-based programs is different from the common imperative execution model, which directly implies that the notion of error is declarative. The same notion of declarative error also applies to constraint and logic programming environments (Aggoun et. al. 1997 [2]). Debuggers for such declarative languages should not require the user to have knowledge of the procedural behaviour of the running system during the interaction phase. However, unfortunately, to make serious use of many existing debugging tools for declarative languages the user is required to have intimate knowledge of the language translation process from the declarative form to the procedural form as well as knowledge of numerical solvers. Making abstractions of all underlying information concerning the translation and the execution of declarative specification in the context of a debugging tool is not always possible. In such cases, tools for visualizing the translation process in a simple manner, understandable for users with limited knowledge of compiler internals, need to be developed.

For example, the declarative object-oriented way of describing systems and their behavior offered by the Modelica language is at a higher level of abstraction than the usual object-oriented programming style since some implementation details can be omitted. Users do not need to write code to explicitly transport data between objects through assignment statements or message passing code. Such code is generated automatically by the Modelica compiler based on the given equations. This property is called the "*acausality*" of the language and adds another dimension to the debugging problem.

Debugging is difficult because an error is not local but can affect many completely different fragments of model code. Typical errors that are caused by incorrect models are missing (or extra) variables or missing (or extra) equations. These lead to over-constrained or under constrained equation systems that cannot be solved. Methods to discover and fix such errors need to be developed based on graph theory and the latest achievements in constraint programming theory.

System diagnosability is a characteristic property of a system design. This is facilitated by component library design features and specific domain knowledge associated with these libraries. Taking into account the characteristics of the domain libraries, an extra reasoning layer can be added to the debugging environment, which can help in further detection of design flaws. All these factors contribute to enhance system diagnosability.

Modeling with components in component-based languages is sometimes difficult because many semantic properties that should be obeyed during the design are not formalized in the language. For instance, an electric circuit can be created from arbitrary components, but simulation of the circuit is possible only if a ground component has been included in the circuit. There exist rules that users of the components should follow in order to create semantically, mathematically, and physically correct models. If these rules are not followed, the model may include non-matching number of equations and variables, or alternatively at run-time during simulation, a solution cannot be found or the found solution makes no sense from the physical point of view.

Equation-based languages are still rather hard to debug. This is to a large extent due to the equation-based language properties mentioned previously. Currently there are essentially no tools that can handle debugging of equation-based languages. We believe that debugging is an essential phase in the process of modeling and simulation using a declarative equation-based languages, as well as for other languages.

## 2.4 Requirements for a Debugging Tool for Equation-Based Languages

This section describes the declarative program errors that the debugger should help to identify. Once the error has been correctly located, this should be presented to the user in an understandable way. In conclusion we can formulate some main characteristics of an equation-based debugger in a modeling and simulation environment:

- Declarative debugging should be fully supported: knowledge of the procedural behaviour of the running program should not be required from the user. If this

cannot be avoided, an appropriate abstraction needs to be found to explain the error to the user.

- The user should be exposed to the original language source code and not burdened with additional information from the translation process or required to inspect the intermediate code.
- The debugger should support different user categories (beginners and advanced users) and present error fixing solutions at different abstraction levels (graphical diagram or original source code level).
- The error fixing solutions should refer to manipulating original source code statements or graphical abstractions of the source code.
- The differential algebraic equations (DAE) and partial differential equations (PDAE) in the language constructs bring an additional set of mathematical and numerical issues, which inevitably lead to requirements on the debugging support. Static methods to deal with structural analysis of differential and algebraic equations should be developed and integrated into the debugging tools.
- The possibility to present advanced abstractions of the translation process related to the intermediate code. This feature is extremely useful and provides valuable information for programming language researchers. Having a complete picture of the intermediate code and being able to visualize it can improve decisions in future implementation of optimized compilers.
- The amount of user interaction needs to be kept at a minimum. The user should be consulted only when needed to eliminate ambiguous situations.

Graphic model editors in modeling and simulation environments help the user to generate and refine simulation models, to store, to group and to reuse them. From the user's point of view, the graphical editors add another view of the modeling language that lies beneath. Graphical model editors are responsible for almost all the interaction of end users who create models out of model components from model libraries. Therefore a careful selection of the debugging information needs to be made and presentation of the error fixing solutions, if possible, at the graphical level. Error fixing solutions that involve adding or removing components or manipulation of object connectors can easily be moved to this level of abstraction.

The end user who limits his/her interaction with the simulation environment to the graphical model editor should not be exposed to the underlying code. For most advanced users, such as library developers who develop and extends their model at the source code level, moving the error fixing solutions to another abstraction level such as the graphical modeling environment is not desirable. Therefore a debugging tool should have the flexibility of moving error fixing solutions between one abstraction level to another and have the possibility to easily adapt to the needs of different user categories. An important issue, once the error has been correctly located is where the error fixing information should be presented to the users and what amount of information is required to explain the error. We have identified the following presentation levels for error messages:

- *Error presentation at the graphical level.* As many as possible of the error messages should be presented at the model editor level. Erroneous components should be highlighted, wrong connections should be drawn in a different color. If user intervention is required to change the behavior of a component by re-

moving or adding equations, it should be done through carefully elaborated graphical user interfaces.

- *Error presentation at the library-developer level*. The library developers using equation-based modeling and simulation systems work at the declarative language source level. They interact with the system at the source code editor level. Any attempt to provide error messages at the graphical model editor level is contra-productive for them. All the interaction should be limited at the source code level. They also require more detailed explanation of the bugs. Prompt feedback should be given by the environment when over- and under-constrained situations are encountered in the system by in the source code editor highlighting the over-constraining equation or indicating the component where an equation should be added for under-constrained situations respectively. At this level, information well beyond the declarative specification makes sense such as domain specific information or numerical solver details.
- *Advanced error presentation.* This level is useful for language and compiler construction researchers. Detailed information about the process of translating declarative to procedural code can be given. Information related to the intermediate code symbolic transformations and numerical optimisations can be presented. Graphs that represent the abstraction of the flattened equations can also be shown and it is extremely useful when visualizing optimisations and code transformations in modeling language compilers.

In the simulation community, there is a strong demand for graphical tools which are considered to be the best way to provide an intuitive description of what is happening. Integration of software visualization techniques such as algorithm animation and typographic source code presentation are aimed at transforming the debugging process into a "*cognitively accessible multimedia experience*" (Baecker et. al. 1997 [8]). A visual environment can contribute significantly to the understandability of modeling languages to be addressed. Moreover, the integration of source code editors with debuggers needs to be substantially improved. Several tools will offer the possibility of studying the program behavior from different points of view in order to help the user to understand the simulation model behavior and to find inconsistencies in the model specifications.

# Chapter 3

# Object-Oriented Equation-Based Modeling Languages

***Summary:*** *This chapter surveys some important object-oriented equation-based modelling languages and their associated environments. All of the languages surveyed in this chapter constitute possible target implementations for our debugging framework even if our implementation first focuses on solving the debugging problem for the Modelica language and its associated environments. Whenever possible, we give concrete language usage examples in order to illustrate the main syntactic and semantic concepts employed by these kinds of modeling languages. The most important object-oriented concepts of those languages are also presented with corresponding detailed examples given in Modelica. This chapter covers only those language features that are necessary to understand the ideas presented in this thesis.*

## 3.1 Declarative Equation Based Languages and Simulation Environments

Many object-oriented equation-based languages have originated in engineering communities especially where natural laws are usually stated as equations and extreme flexibility and reusability of equations is highly valued. The brief survey of the languages and associated environments below concentrates more on the compositional aspects such as object-orientation and hierarchical structuring of models rather than on numerical simulation related aspects.

Most of these modeling languages are integrated in visual modeling programming environments where components are grouped into libraries and represented by model diagrams. Model diagrams make modeling languages easier to use by making concise physical components readily available and allowing users to select those components through the simple act of pointing, picking, and dropping the components into the graphical editing window. Inside the graphical modeling environment the connections between components can also be specified by drawing simple lines between the connection ports of the components. Visual modeling environments are not required by model-

ing languages but they contribute to ease of use of such language. The more complicated the physical model to be simulated, the more likely it is to benefit from a visual modeling interface.

### 3.1.1 Modelica

Modelica is a new language for hierarchical object-oriented physical modeling which is being developed through an international effort (Modelica Assoc. 2002 [84]), (Fritzson and Bunus 2002 [41]), (Modelica Assoc. 2000 [85]), (Elmqvist et al. 1999 [35]), and (Fritzson and Engelson 1998 [42]). The language unifies and generalizes previous object-oriented modeling languages. Modelica is intended to become a *de facto* standard. The language has been designed to allow tools to generate efficient simulation code automatically with the main objective of facilitating exchange of models, model libraries and simulation specifications. 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, thermodynamic, etc., model components within the same application model. Compared to other modeling languages available today, Modelica offers four important advantages from the simulation practitioner's point of view:

- Acausal modeling based on ordinary differential equations (ODE) and differential algebraic equations (DAE). There is also ongoing research to include partial differential equations (PDE) in the language syntax and semantics (Saldamli et. al. 2002 [107]).
- Multi-domain modeling capability, which gives the user the possibility to combine electrical, mechanical, thermodynamic, hydraulic etc., model components within the same application model.
- A general type system that unifies object-orientation, multiple inheritance, and generics templates within a single class construct. This facilitates reuse of components and evolution of models.
- A strong software component model, with constructs for creating and connecting components. Thus the language is ideally suited as an architectural description language for complex physical systems, and to some extent for software systems.

The language is strongly typed and there are no side effects of function calls. However, local assignments are allowed in the algorithmic part of the language. The reader of the thesis is referred to (Modelica Assoc. 2002 [84]), (Modelica Assoc. 2000 [85]), and (Tiller 2001 [114]) for a complete description of the language and its functionality from the perspective of the motivations and design goals of the researchers who developed it. Those interested in shorter overviews of the language may wish to consult (Fritzson 2002 [43]) (Fritzson and Bunus 2002 [41]), (Elmqvist et al. 1999 [35]) and (Fritzson and Engelson 1998 [42]). More details about the Modelica language, including several modeling examples, will be provided later in this chapter.

### 3.1.2 Object-Oriented Biomedical System Modeling (OOBSML)

The OOBSML project (Hakman 2000 [54]) has been developed at Uppsala University Hospital aimed at the development of a biomedical continuous system modeling language. The OOBSML (Hakman and Groth 1999 [52]) (Hakman and Groth 1999 [53]) is fully object-oriented and supports model inheritance, encapsulation, and model component instantiation. Besides the traditional differential and algebraic equation expressions the language also includes formal expressions for documenting models and defining model quantity types and quantity units. It supports explicit definition of model input, output and state quantities, model components and component connections. The OOBSML model compiler produces self-contained, independent, executable model components that can be instantiated and used within other OOBSML models and/or stored within model and model component libraries. In this way complex models can be structured as multilevel, multi-component model hierarchies. Technically the model components produced by the OOBSML compiler are executable computer code objects based on distributed object and object request broker technology (Pope 1997 [97]). The most basic concept of the OOBSML language is the model concept. A model encapsulates system knowledge and information related to the purpose of modeling such as: input and output quantities, state quantities, types and units, and model behavior represented by algebraic and differential equations.

The syntax of the language very much resembles Omola (Andersson 1992 [5]) (Mattson and Andersson 1992 [82]). A simple example of a two compartment model used for medical simulation is taken from (Hakman 2000 [54]) in order to illustrate the syntax employed by the language.

```
model TwoCompWitMetabolism: TwoComp has
   inputs
      Vmeta: reaction_rate [mol/L/h]
   outputs
      Fmeta: substance_flow [mol/h]
   behaviour Dynamic is
   begin
      Np'=Fpo-Fop+Fpe-Fep+Fmeta
      Fmeta=Vmeta *Vp
   end Dynamic
end TwoCompWitMetabolism
```

### 3.1.3 VHDL-AMS

The VHDL-AMS language (Christen and Bakalar 1999 [25]) is a hardware description language that has been developed to support the modeling of analog, digital, mixed-signal and mixed-technology systems. The language extends the IEEE standard digital design language VHDL (IEEE Std 1076.1-1999 [61]) to cover mixed analog and digital systems. The VHDL-AMS language constructs provide a concise notation for differential algebraic systems specifying the continuous aspects of physical system behavior as well as the discrete aspects of the VHDL part. As in the case of other equation-based modeling languages the VHDL-AMS language does not specify techniques for the solution of the expressed DAEs, leaving the selection of a suitable numerical method to

lution of the expressed DAEs, leaving the selection of a suitable numerical method to the simulation environment.

A typical VHDL-AMS simulation framework is described in (Schneider et. al. 2000 [108]) and consists of a compiler and a simulator. The compiler translates the VHDL-AMS source code into an intermediate representation structured as abstract syntax trees with annotated symbol tables and additional semantic information. Similar to other equation-based simulation environments, a hierarchical application model is flattened to a set of discrete and continuous equations. An analog kernel calculates the analog solutions of the equations at a specified time step by providing an interface to a numerical solver. In this way the original model description is compiled into intermediate object code, linked together with a simulation kernel, and directly executed. The same translation process is also used by SEAMS (Frey et. al. 1998 [40]) another VHDL-AMS based simulation framework. SEAMS incorporates an analyzer, an analog and digital kernel, a code generator and the compilation module. A design support environment called SIDES (Simulation-Interactive Design Environment System) is attached to SEAMS and includes a GUI viewer, editors, debugging and editing support, library access, and a file and data exchange database to enable all the necessary design environment-simulator interactions (Dragger et. al. 1998 [28]).

For reasoning and verification purposes, a denotational semantics of the language has been developed (Breurer et al. 1999 [23]). The general behavior of VHDL-AMS programs is derived by translating the semantics into a prototype simulator implementation.

In order to illustrate the syntax of the language the model of a simple linear resistor is given below:

```
use electrical_system.all
entity resistor is
   generic(resistance: real);
   port(terminal n1, n2: electrical);
end entity resistor;

architecture signal_flow of resistor is
   quantity vr across ir through n1 to n2;
begin
   ir == vr / resistance;
end architecture signal_flow.
```

### 3.1.4 Neutral Model Format

The Neutral Models Format language (Sahlin 1996 [104]), (Sahlin 1996 [105]) has been created to cover the needs of simulation in the building industry. However, due to its structuring capabilities and the possibility of specifying differential algebraic equations the language can be used to specify simulation models for other engineering domains as well.

The inheritance mechanism was not present in the original language proposal but was added later to the language specification (Sahlin et. al. 1995 [106]). In the same manner as other equation-based modeling languages, the inheritance mechanism where equations and declarations inherited from the parent object become part of the child ob-

ject, was introduced into NMF. Child objects have the possibility of adding new equations and declarations, specializing the behavior of the created object. Compared to other equation-based modeling languages the distinctive feature of the inheritance proposed for NMF is that overriding is permitted for uniquely identifiable declarations. A previously declared link, variable or parameter can be overridden by new declarations in the created class. In this way a mechanism to specialize link types is provided in the language.

Simulation models can be translated automatically from NMF into the local representation of a particular simulation environment. Several simulation environments have been built around the language. One of these is the IDA Simulation Environment IDA SE, which includes a graphical modeler, an NMF translator, and a solver. The models expressed with NMF are easily understandable and readable for non-experts. In order to illustrate the readability and expressiveness of the language a model of a thermal conductance is taken from (Sahlin 1996 [104]) and presented below:

```
CONTINUOUS_MODEL  tq_conductance
ABSTRACT  "Linear thermal conductance"
EQUATIONS  /* heat balance */
   0 = - Q + a_u * (T1 - T2);
LINKS
   /* type  name         variables
       TQ   terminal_1   T1, POS_IN Q;
       TQ   terminal_2   T2, POS_OUT Q;
 VARIABLES
   /* type     name   role   description
      Temp      T1     IN     "1st temp"
      Temp      T2     IN     "2nd temp"
      HeatFlux  Q      OUT    "flow from 1 to 2"
PARAMETERS
   /* type     name   role   description
      Area      a     S_P     "cross section area"
      HeatCondA u     S_P     "heat transfer coeff"
      HeatCond  a_u   C_P     "a * u"
PARAMETER_PROCESSING
   a_u := a * u;
END_MODEL
```

### 3.1.5 The χ Language for Hybrid Systems Simulations.

The χ (Fábian [37]) language has been designed with the aim of providing a general language for hybrid system simulations. A χ program consists of processes, systems and function specifications, which are later instantiated by a top-level model. As is the case with most equation-based languages, the top model instance is executed by the simulation environment. A process in χ may have a discrete-event part and/or a continuous–time part and a system may be composed from processes and system instantiations. The continuous-time behavior in χ is expressed with the help of equation statements defined on the local variables of the processes. Model composition in χ is achieved by the use of *channels* in the interface specifications of the sub-models. The *channels* have the same role as the connectors in the Modelica language. A process example taken from (Beek 2001 [17]) is given below:

```
proc P =
| [V,x: real] := 0
   , Q: real
   ,b: bool := true
  {
    V' = 2 - Q             //eq0
    ,|Q| Q = sqrt(V)       //eq1
    ,[b -> x' = 1 - x]     //eq2
     | not b -> |x| x = 2 //eq3
     ]
  }
]|
```

### 3.1.6 gProms

gPROMS (Oh and Pantelides 1996 [92]) is a general modeling language for combined lumped and distributed parameter processes described by mixed systems of integral, partial differential and algebraic equations (IPDAEs). The modeling language is integrated in a more general simulation system called gPROMS SYSTEM (Barton and Pantelides 1993 [13]), (Barton and Pantelides 1994 [12]) designed to support both continuous and discrete simulation. The language distinguishes two modeling entities: MODELs (which describes the physical system behavior equations) and TASKs (external actions and disturbances imposed on the simulation model). Below, we illustrate a lumped parameter model, taken from (Oh and Pantelides 1996 [92]) which is defined in gPROMS as follows:

```
MODEL IsothemalFlash
   PARAMETER
      Nocomp  AS   INTEGER
   VARIABLE
      M,   AS ARRAY (Nocomp) OF Holdup
      F,   AS ARRAY (Nocomp) OF Flowrate
      V,L  AS                   Flowrate
      x,y  AS ARRAY (Nocomp) OF MoleFraction
      K    AS ARRAY (Nocomp) OF Kvalue
           ..................................................
   EQUATION
      #Component mass balance
      $M = F - L * x - V * y;
      #Phase equilibrium relationship
      FOR i:= 1 TO Nocomp DO
          K(i) * x(i) = y(i);
      END
      SIGMA(x)=SIGMA(y)=1;
           ..................................................
END #model IsothermalFlash
```

In the above example it can be seen that a MODEL is composed of a set of VARIABLEs and EQUATIONs. In the equation section algebraic, differential and partial differential equations are allowed. The reusability of simulation models described with gPROMS is achieved by hierarchical submodel decomposition and by using the inheritance mechanism.

### 3.1.7 Abacuss II

Abacuss II (Advanced Batch and Continuous Unsteady State Simulator) (Tolsma et al. 2002 [117]) provides an intuitive, high level, declarative input language for describing process model simulation. The syntax of the Abacuss II language is in the tradition of Pascal and Modula-2 and is fully described in (Barton 1992 [16]). The integrated simulation environment built around the language uses DAEPACK (Differential-Algebraic Equation Package) (Tolsma and Barton 2000 [115]) to perform the required symbolic and numerical calculations.

The inheritance mechanism used by Abacuss II is in the form of single inheritance: a model which inherits from another model gains all the attributes of the ancestor model. The inheritance mechanism is implemented by the keyword `INHERITS` and the identifier of the parent immediately followed by the identifier of the new model entity. This is illustrated in (Tolsma et al. 2002 [116]) by the following example:

```
MODEL Father
  PARAMETER PI AS REAL DEFAULT 4 * ATAN(1.0)
END

MODEL Child INHERITS Father
  PARAMETER E AS REAL DEFAULT 2.718
END
```

The `Child` model will inherit all the parameters of the `Parent` model, and in this particular case `Child` will contain both `PI` and `E` as parameters. Multiple or selective inheritance are not allowed in the language. The benefit of these forms of inheritance used in the context of process engineering is discussed in (Barton 1992 [16]).

The interaction of models with the environment is realized through *stream attributes* that are subsets of the variables describing the time-dependent behavior of the system. Connecting two models in Abacuss is realized by setting up a relationship between stream attributes corresponding to each submodel providing a compressed way of specifying interaction equations among components. The connection relationships need to be declared in the `EQUATION` section of the model entity. Conditional equations are also allowed to define dynamic changes in the topology of the simulation system. Usually equality constraints stated between stream variables is the most common kind of constraint used in process simulation.

The Abacuss II simulation environment also incorporates a fairly advanced debugging module where over- and under-constrained situations can be detected (Barton 1995 [14]) (Barton 2000 [15]). This is done by using Dulmage and Mendelsohn canonical decomposition performed on the incidence matrix corresponding to the underlying system of equations that defines the behavior of the simulation model. A way of visualizing the sparsity pattern is also provided for program understanding purposes. The debugging module is briefly described in Chapter 9 and compared to our approach in Chapter 11.

### 3.1.8 Cob

Cob is a novel programming language based on the concept of constrained objects for compositional and declarative modeling of engineering structures (Jayaraman and Tambay 2002 [66]). The abstraction models are specified using a declarative object-oriented language. The associated environment includes a graphical editor (Jayaraman and Tambay 2002 [65]) for developing Cob class definitions, a domain-specific visual interface where already developed components can be combined and a translator, which takes the declarative specification and generates an equivalent CLP(R) program (Joxan et. al. 1992 [70]). The translator represents the main difference compared to the previously surveyed languages and environments. Instead of generating an imperative equivalent form of the original source code and linking it to numerical solvers, the Cob translator generates an equivalent declarative specification, which is sent to an underlying CLP(R) engine. In this way Cob extends the declarative properties of CLP by introducing object-orientation, conditional constraints and preferences and subsumes the paradigms of CLP and HCLP (hierarchical CLP) (Wilson 1993 [124]) (Wilson and Borning 1993 [125]). By introducing these additional capabilities the applicability of constraint-based languages can easily be extended in order to consider engineering applications where components are modeled much more naturally in terms of objects and inheritance hierarchies.

In order to present the expressiveness of the Cob language compared to the equivalent generated CLP form we give an example of a simple electrical circuit taken literally from (Jayaraman and Tambay 2001 [67]).

```
class component
   attributes Real V,I,R
   constraints V=I*R;
   constructors component(V1,I1,R1){
       V=V1.  I=I1.   R=R1. }
}
class parallel extends component
   attributes component [] PC
   constraints
   forall X in PC: X.V = V.
   sum Y in PC : Y.I=I.
   sum Z in PC : (1/Z.R)=1/R.
   constructors parallel(B){
      PC=B}
}
```

```
p_component([V1, I1, R1], [V, I, R]):-
    V=I*R, V=V1, I=I1, R=R1.
p_parallel ([B],[V,I,R,PC]):- PC = B,
    p_component(_,[V,I,R]), forall1(PC,X,V),
     N1=I, sum1(PC,N1, Y,I),
     N2=1/R, sum2(PC,N2,Z,R).
forall1([],X,V).
forall1([X|Tail],X,V)):-
    X_V=V, X=[X_V,X_I,X_R],
forall1(Tail,X,V).
sum2([],0,Z,R).
sum2([Z|Tail],(1/Z_R)+Sumrest,Z,R):-
    sum2(Tail,Sumrest,Z,R),
Z=[Z_V,Z_I,Z_R].
sum1([],0,Y,I).
sum2([Y|Tail],(Y_I)+Sumrest,Y,I):-
    sum2(Tail,Sumrest,Y,I),
Y=[Y_V,Y_I,Y_R].
```

The Cob language successfully extends previous object-oriented constraint imperative languages such as Kaleidoscope'91 (Benson and Borning 1992 [18]) and ThingLab (Borning 1979 [21]) with *stratified preference logic programs* that allows an efficient simulation of the constraint hierarchies (Govindarajan et. al. [50]).

## 3.2 Object-Oriented Equation-Based Language Constructs

In this section the main language features and constructs of object-oriented equation-based modeling languages are presented. For the general language constructs we are using a description which resembles the description presented in (Abadi and Cardelli 1996 [1]) in order to provide a framework for comparison between modeling languages and traditional object-oriented languages.

### 3.2.1 Acausal Modeling

At the lowest level of the languages presented previously, equations are used to describe the relations between the quantities of a model and to define the behavior of the class. As mentioned before, one of the distinctive features of object-oriented equation-based languages is the acausal programming model. The computation semantics does not depend on the order in which equations are stated. However, this property complicates the debugging process.

The acausality makes the library classes more reusable than traditional classes containing assignment statements where the input-output causality is fixed, since the language classes adapt to the data flow context in which they are used. The data flow context is defined by stating which variables are needed as *outputs* and which are external *inputs* to the simulated system. From the simulation practice point of view this generalization enables both simpler models and more efficient simulation. The declarative form allows a one-to-one correspondence between physical components and their software representation.

### 3.2.2 Class Definitions

Programs in object-oriented equation-based languages are built from classes like in any other traditional object-oriented language. A class[1] is intended to describe the structure of the object generated from the class. The main difference compared to traditional object-oriented languages is that instead of functions (methods), equations are used to specify behavior. A class declaration contains a list of variable declarations and a list of equations preceded by a keyword, usually `equation`.

The following is an example of a low pass filter in Modelica taken from (Modelica Assoc. 2000 [85]).

[1] In object-oriented modeling languages a class is also referred to as a `model`. For the Modelica examples in this thesis we usually use the reserved word `model` instead of the word `class`, even if from the "computer science" perspective the name `class` is more appropriate.

```
class LowPassFilter
   parameter Real T=1;
   Real u, y (start=1);
equation
   T*der(y) + y = u;
end LowPassFilter;
```

The model `LowPassFilter` can be used to create two instances of the filter with different time constants, "connecting" these together by an equation as follows:

```
class FilterInSeries
   LowPassFilter F1(T=2), F2(T=3);
equation
   F1.u = sin(time);
   F2.u = F1.y;
end FilterInSeries;
```

### 3.2.3 Inheritance

The natural inheritance mechanism in modeling languages allows classes to be seen as the extension of existing classes with new equations and variables as is illustrated in Figure 3-1. This is in fact the most common inheritance mechanism for object-oriented equation-based languages and the one from which we derive, later on, the rules for the graph transformations associated to the intermediate form. This inheritance presents some important benefits especially from the modeling language point of view, such as: better support for concrete visual programming systems, and any object, can be given individualized behavior (Borning 1986 [20]).

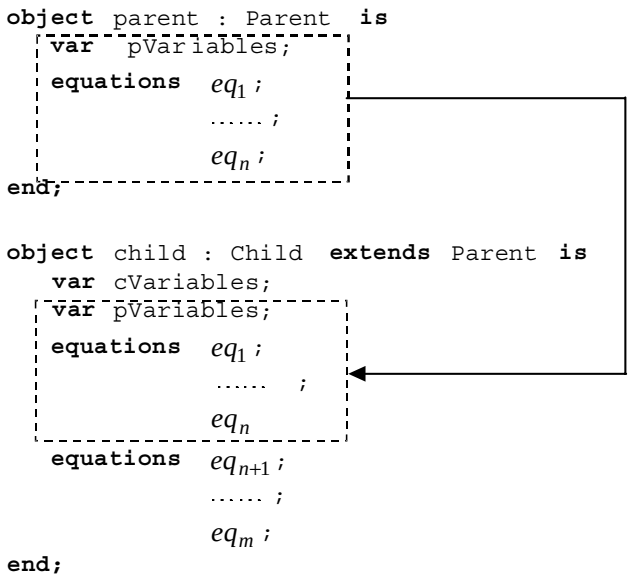



**Figure 3-1.** The inheritance mechanism

Alternatively, an equivalent model or class can be defined by a pure extension of the parent class.

```
class Child is
   var  pVariables, cVariables;
   equations eq1 ;
             ..... ;
             eqm ;
end;
```

### 3.2.4 Class Subtyping

The notion of subtyping, especially in Modelica, is influenced by the theory of objects (Abadi and Cardelli 1996 [1]). The notion of inheritance is separated from the notion of subtyping. According to the definition, a class `A` is a subtype of a class `B` if class `A` contains all public variables and parameters declared in the class `B`, and the types of these variables are subtypes of the types of the corresponding variables in `B`. For instance, the class `TempResistor` is a subtype of `Resistor`.

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

```
class TempResistor
      extends TwoPin;
      parameter Real R, RT, Tref;
      Real T;
equation
      v = I * (R+RT * (T - Tref));
end TempResistor;
```

Subtyping is used, for example, in class instantiation, redeclarations and function calls. If variable `a` is of type `A`, and `A` is a subtype of `B`, then `a` can be initialized by a variable of type `B`.

Note that `TempResistor` does not inherit the `Resistor` class. There are different equations for the evaluation of `v`. If equations are inherited from `Resistor`, then the set of equations will become inconsistent in `TempResistor`, since Modelica currently does not support named equations and replacement of equations. For example, the specialized equation below from `TempResistor`:

```
v=i*(R+RT*(T-Tref))
```

and the general equation from class `Resistor v=R*i` are inconsistent with each other: only one of these two equations should be present in a class.

### 3.2.5 Connections and Connectors

Equations in Modelica can also be specified by using the special `connect` equation syntactic form. The equation form `connect(v1, v2)` expresses a coupling between variables. These variables are called connectors and belong to the connected objects. Connections specify interaction between components. A connector should contain all quantities needed to describe the interaction. This provides a flexible way of specifying the topology of physical systems described in an object-oriented way using Modelica.

For example, `Pin` is a connector class that can be used to specify the external interfaces for electrical components that have pins. Each `Pin` instance is characterized by two variables: voltage `v` and current `i`. A connector class is defined as follows:

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

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

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

## 3.3 Program Transformation Rules and Semantics

This section presents rules for transforming the declarative object-oriented equation-based code to the intermediate equation form. The set of program transformation rules must preserve the operational semantics of the transformed code. In later chapters, during the debugging process, the transformations applied to the corresponding graph representation of the intermediate form must to be consistent with the program transformation rules. Therefore, a precise set of rules needs to be derived from the language semantics that maps the original code statements with their corresponding intermediate flattened statement.

For the purpose of debugging we are only interested in a few aspects of the program transformations such as: the equation transformations by inheritance, by object instantiation and the equations generated by `connect` equations. Only these aspects will be covered in this thesis. The equations in flattened form, in the intermediate code, are often the initial point of investigation for an associated debugger. From the user perspective, it is useful, to investigate the cause of an error backwards in the original source code from where the bug manifests itself. From the debugging perspective it is important to map back a statement from the intermediate code to the original language as well as to determine which other statements are affected if the transformation rule is inhibited.

Let us now take the Modelica language and see how the transformations[2] from the original source code to the intermediate code takes place. It is extremely important to detail the properties of the intermediate code because most of the debugging process will take place after the flattening process and all the error fixing algorithms are performed on the flattened equation set.

A Modelica program `P` is a set of class declarations. Each class consists of a set of variable and constant declarations and a sequence of equations or connect equations. We define an object as a collection of elements and equations $[v_j^{j\in 1\ldots m}, eq_i^{i\in 1\ldots n}]$. The

[2] This set of transformations is also called flattening mainly due to the "flat" properties of the resulting intermediate language.

following main transformation steps are performed on the original code when translating from the original form to the intermediate flattened form.

- Inheritance expansion.
- Class instantiation.
- Loop unrolling.
- Connect equation expansion

The translation of the original source code program can be summarized by the following rule expressed in natural semantics:

$$[trans_{prog}] \quad \frac{p \xrightarrow{p} p_p \quad \wedge \quad p_p \rightleftharpoons p_e \quad \wedge \quad p_e \searrow p_f}{p \xrightarrow[ff]{} p_f}$$

The translation of the original source code $p$ into the intermediate form $p_f$ takes place in different stages according to the semantic rule $[trans_{prog}]$. The parsing stage denoted by $p \xrightarrow{p} p_p$ is not really part of the semantic description, but it is of course necessary to build a real translator. The other two main phases of the source code transformations are the *elaboration* and the *instantiation* denoted by $p_p \rightleftharpoons p_e$ and $p_e \searrow p_f$ respectively.

- *Elaboration*. This phase takes place only at compile time. No data objects are created during this phase. The natural semantics of the *elaboration* is defined through an "*elaborate*" relation $\rightleftharpoons$ on elements that depend on the definition of each element in the corresponding class model. For example, the elaborate relation $\rightleftharpoons$ extends naturally to the model source code by application to all equations present in an associated class. The notation $c \rightleftharpoons c_e$ means that the class $c$ is transformed through multiple steps into the elaborated form $c_e$. This phase is responsible for the code transformations related to the inheritance, modification and aggregation operations.
- *Instantiation.* The instantiation phase is responsible for the creation of the data objects and is defined through an "*instantiate*" relation $\searrow$. The notation $c \searrow o$ means that the class $c$ is instantiated and object $o$ is created.

As was mentioned previously a program is defined as a set of classes. Elaborating a program means that all classes that is not `partial` class needs to be elaborated. Following the elaboration and instantiation rules the final flattened form is produced.

Let us consider the following simple Modelica model and see how different translation rules affect the final flattened form of the model. During model elaboration the variables declared in a list of variables are expanded into separate declarations, as is the case for the `Real` variables declared for `model A` in the example below. The equations remain unchanged during the elaboration process in this particular case.

```
model A                    model A
   Real x,y;                 Real x;
equation                     Real y;
   x + y = 2;              equation
end A;                        x + y = 2;
                           end A;
```

The convention in Modelica is that the top-level class in the instantiation hierarchy is expanded since all instances in the total model belong to this. During elaboration the whole program is reduced to one flattened class. Therefore model `B` is eliminated from the final flattened form of the simulation model and the classes are reduced to the flattened form of the `model Aflattened` shown on the right.

```
model B
   Integer s;
equation
   s = 0;                   model Aflattened
end B;                         Real x;
                               Real y;
model A                        Integer b.s;
   Real x;                  equation
   Real y;                     b.s = 0;
   B b;                        x + y = 2;
equation                       x - y = 3;
   x + y = 2;               end Aflattened;
   x - y = 3;
end A;
```

Below, instead of elaborating model `B` in class `A`, as above, we are extending model `A` from model `B` using inheritance. In this case all the variables and equations declared in model `B` are simply copied into the corresponding elaborated `model A2` as is shown below:

```
model B                    model B
   Integer s;                 Integer s;
equation                   equation
   s = 0;                     s = 0;
end B;                     end B;

model A2 extends B         model A2flattened
   Real x,y;                  Real x;
equation                      Real y;
   x + y = 2;                 Integer s;
   x - y = 3;              equation
end A2;                       x + y = 2;
                              x - y = 3;
                              s = 0;
                           end A2flattened;
```

In one of the current Modelica language implementations (Fritzson et. al. 2002 [44]) (Aronsson et. al. 2002 [6]) the semantics is specified using the RML specification language (Pettersson 1999 [95]). The RML language is based on Natural Semantics (Kahn 1987 [72]), The RML source code, e.g. a Modelica specification, is compiled by the RML compiler (rml2c) to produce a translator for the described language (Fritzson et. al. 2002 [44]) (Kågedal and Fritzson 1998 [71]). The RML compiler generates an efficient ANSI C program that is subsequently compiled by an ordinary C compiler, e.g.

producing an executable Modelica translator. The generated translator is produced in ANSI C with a performance comparable to hand-written translators. The RML tool has also been used to produce compilers for Java, Pascal and a few other languages.

## 3.4 Source Code Manipulation

We categorize source code edits into a set of atomic change operations similar to those defined in (Ryder and Tip 2001 [103]) but adapted to the specific language constructs of the Modelica language. The defined atomic changes and the explanations are given in Table 3-1. The error fixing solutions for erroneous models developed in later chapters should be expressible in terms of these atomic changes. The implemented debugger will manipulate the Modelica source code in terms of atomic changes.

**Table 3-1.** Atomic change operations performed on the original Modelica source code

| Atomic change operations at the source code level | Description |
|---|---|
| `DELEQ(`*eq,model*`)` | Deletes an equation *eq* from the model *model* |
| `ADDEQ(`*eq,model*`)` | Adds an equation *eq* to *model* |
| `DELVARM(`*var,model*`)` | Deletes a variable *var* from a *model* |
| `DELVARE(`*var,eq,model*`)` | Deletes a variable *var* from an equation *eq* |
| `DELMOD(`*model*`)` | Deletes an empty *model* |
| `ADDMOD(`*model*`)` | Adds an empty *model* |
| `DELLCON(`*conn,model*`)` | Deletes a connector *conn* from the *model* |
| `ADDCON(`*conn,model*`)` | Adds a connector *conn* to the *model* |

The `DELVARM(`*var,model*`)` kind of atomic change deletes a variable declaration from the model This operation makes sense only if the variable scheduled for elimination is not present in any equation that defines the behavior of the model. Eliminating only the variable declaration will result in a program that will not compile correctly. This change is useful for the elimination of additional declared variables in the simulation model. The elimination of a variable from a model can also be achieved by transforming the variable declaration into a constant declaration. At this level of the analysis we do not distinguish between total variable elimination and transforming a variable into a constant, both cases fall into the same atomic change.

For example, the following transformed model can be obtained by successively applying the following atomic changes: `DELVARE(x,x+y=2,A)`, `DELEQ(x-y=3,A)`, and `DELVARM(x,A)`.

```
model A                    model A
   Real x,y;                 Real y;
equation                   equation
   x + y = 2;                y = 2;
   x - y =3                end A;
end A;
```

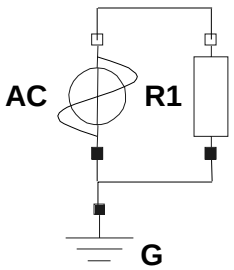
Just now, we ignore several source code changes that involve manipulation of constants (adding or deleting them to and from equations) because they only change the behavior of the model and do not affect the structural analysis method developed later in the thesis.

## 3.5 Simple Modelica Simulation Models

Let us illustrate the expressiveness and functionality of the Modelica language by several simple simulation examples that will also be used in a slightly modified configuration as examples for testing the functionality of the debugger in subsequent chapters. Additional examples will be introduced in the thesis when necessary to illustrate some aspects of the debugging process. The models developed here are trivial, almost beneath consideration from the usability point of view, but they keep the associated bipartite graphs and related algorithms to a minimum complexity.

### 3.5.1 Simple Electrical Circuit

Our first example is a simple electrical circuit where a `Resistor` component is connected in series together with an alternating current source `VsourceAC` as illustrated in Figure 3-2 on the left.




```
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 3-2**. Simple electrical circuit model.

We first start by defining the interfaces for electrical components that have a pin. The following `connector` class uses two `Real` variables: one for the current and one for the voltage. Since in an electrical circuit the current should always be summed when connecting two components, according to Kirchoff's law, the variable `i` defined in the `Pin` component will have the prefix `flow`.

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

Based on the `Pin` connector we can define a partial class for electrical components that have two pins. Obviously, such a class should instantiate the previously declared pin

class twice and provide some extra equations that define the behavior of components 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`). The partial class for `TwoPin` components is given below:

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

Now we can specialize the `TwoPin` class by defining the `Resistor` class and a `VsourceAC` class that will inherit all the equations from their respective parent class and add one equation that will define the specific behavior of the component class:

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

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

The `Ground` component is defined as follows and will lower the voltage drop to zero at the connection point.

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

The whole `Circuit` model is obtained by combining the `Resistor`, `VsourceAC` and `Ground` components `R1`, `AC` and `G` respectively through `connect` equations as illustrated in Figure 3-2 on the right.

The Modelica compiler will automatically generate the equations from the `connect` equations present in the `Circuit` model. For example, the `connect` equation that defines a connection between the positive pins of the source and resistor components `connect(AC.p, R1.p)` will be expanded into two equations at the intermediate source code level: `AC.p.v = R.p.v` and `AC.p.i + R.p.i = 0`

### 3.5.2 Direct Current Motor Example

The second example is a more complicated electrical circuit combined with mechanical components defining a direct current motor model. The components within the `DCMotor` model are taken from the Modelica Standard Library. This model illustrates the multidomain capabilities of the Modelica language by combining electrical and mechanical components in the same simulation model. The connection diagram of the `DCMotor` model is depicted in Figure 3-3.

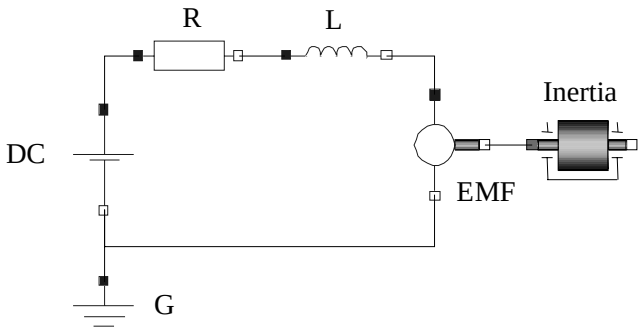



**Figure 3-3.** DC electric motor model.

The corresponding Modelica source code is given below:

```
model DCMotorCircuit
   Modelica.Electrical.Analog.Basic.Resistor R;
   Modelica.Electrical.Analog.Basic.Inductor L;
   Modelica.Electrical.Analog.Basic.EMF EMF;
   Modelica.Electrical.Analog.Basic.Ground G;
   Modelica.Electrical.Analog.Sources.ConstantVoltage DC;
   Modelica.Mechanics.Rotational.Inertia Inertia;
equation
   connect(DC.p, R.p);
   connect(R.n, L.p);
   connect(L.n, EMF.p);
   connect(EMF.n, DC.n);
   connect(G.p, DC.n);
   connect(EMF.flange_b, Inertia.flange_a);
end DCMotorCircuit;
```

At the intermediate code level this model will contain 36 equations and variables.

## 3.6 A More Complicated Example

Another example used later in our tests is a more complicated modeling example that includes control, electrical, and rotational mechanical components. The components taken from the Modelica Standard Library are combined together as is illustrated in Figure 3-4.

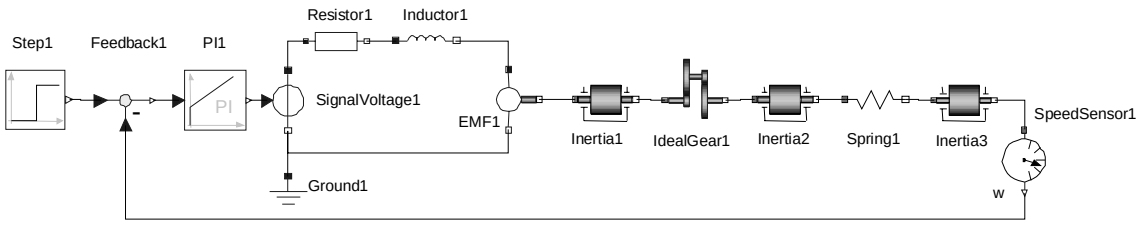



**Figure 3-4**. Simulation example involving control, electrical and rotational mechanics components.

The corresponding Modelica code is given below.

```
model Model1
   Modelica.Electrical.Analog.Basic.Resistor Resistor1;
   Modelica.Electrical.Analog.Basic.Inductor Inductor1;
   Modelica.Electrical.Analog.Sources.SignalVoltage
                                              SignalVoltage1;
   Modelica.Electrical.Analog.Basic.EMF EMF1;
   Modelica.Electrical.Analog.Basic.Ground Ground1;
   Modelica.Mechanics.Rotational.Inertia Inertia1;
   Modelica.Mechanics.Rotational.IdealGear IdealGear1;
   Modelica.Mechanics.Rotational.Inertia Inertia2;
   Modelica.Mechanics.Rotational.Spring Spring1;
   Modelica.Mechanics.Rotational.Inertia Inertia3;
   Modelica.Blocks.Sources.Step Step1;
   Modelica.Blocks.Continuous.PI PI1;
   Modelica.Blocks.Math.Feedback Feedback1;
   Modelica.Mechanics.Rotational.Sensors.SpeedSensor
                                              SpeedSensor1;
equation
   connect(Step1.outPort, Feedback1.inPort1);
   connect(Feedback1.outPort, PI1.inPort);
   connect(PI1.outPort, SignalVoltage1.inPort);
   connect(SignalVoltage1.p, Resistor1.p);
   connect(Resistor1.n, Inductor1.p);
   connect(Inductor1.n, EMF1.p);
   connect(EMF1.flange_b, Inertia1.flange_a);
   connect(Inertia1.flange_b, IdealGear1.flange_a);
   connect(IdealGear1.flange_b, Inertia2.flange_a);
   connect(Inertia2.flange_b, Spring1.flange_a);
   connect(Spring1.flange_b, Inertia3.flange_a);
   connect(SignalVoltage1.n, EMF1.n);
   connect(Ground1.p, SignalVoltage1.n);
   connect(Inertia3.flange_b, SpeedSensor1.flange_a);
   connect(SpeedSensor1.outPort, Feedback1.inPort2);
end Model1;
```

The flattened system of equations contains 78 differential algebraic equations and 78 variables.

# Chapter 4

# Graph Theoretical Preliminaries and System Decomposition

***Summary*****:** *This chapter provides a preliminary introduction to the graph theoretical concepts and related algorithms used by the debugging framework. The fundamental definitions and algorithms necessary for understanding susequent chapters are given here. The algorithm for system decomposition is also presented.*

## 4.1 Introduction

Many practical problems form a model of interaction between two different types of objects and can be phrased in terms of problems of bipartite graphs. The expressiveness of bipartite graphs in concrete practical applications has been demonstrated many times in the literature (Dolan and Aldous 1993 [27]) (Asratian et. al. 1998 [7]). We will show that the bipartite graph representation is general enough to efficiently accommodate several symbolic analysis methods in order to reason about the solvability and unsolvability of the flattened system of equations. Characteristics of the simulation model's behavior can also be implied from the bipartite graph representation. Another advantage of using bipartite graphs is that this representation offers an efficient abstraction necessary for program transformation visualization when equation-based specifications are translated into procedural form.

The bipartite graph representation with associated decomposition techniques is widely used internally by compilers that translate equation-based simulation code into procedural code (Elmqvist 1978 [36]) (Maffezzoni et. al. 1996 [78]). However, none of the existing simulation systems use this representation for debugging purposes or expose it visually for program understanding purposes. Our debugging approach follows the same philosophy as the method for reduction of constraint systems used for geometric modeling in (Ait-Aoudia et. al. 1993 [3]) and (Bliek et. al. 1998 [19]).

In this chapter it is our intention to present the basic definitions and some notation, which we shall use throughout the rest of this thesis. Although fairly brief, this introduction to bipartite graph theory will provide a sufficient basis for understanding the methods and algorithms developed in this thesis.

## 4.2 Basic Definitions

**Definition 4-1:** 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, $V_1 \cap V_2 = \emptyset$ and $E \subseteq \{\{x, y\}; x \in V_1, y \in V_2\}$ the set of edges. The vertices of $G$ are elements of $V_1 \cup V_2$. The edges of $G$ are elements of $E$.

**Definition 4-2:** Let $G$ be a graph with a vertex set $V(G) = \{v_1, v_2, \ldots, v_p\}$ and an edge set $E(G) = \{e_1, e_2, \ldots, e_q\}$. The *incidence matrix* of $G$ is the $p \times q$ matrix $M(G) = \lfloor m_{ij} \rfloor$, where $m_{ij}$ is 1 if the edge $e_{ij}$ is incident with vertex $v_i$ and 0 otherwise.

We consider the bipartite graph associated to a given system of equations resulting from 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 Chapter 2 is shown in Figure 4-1.

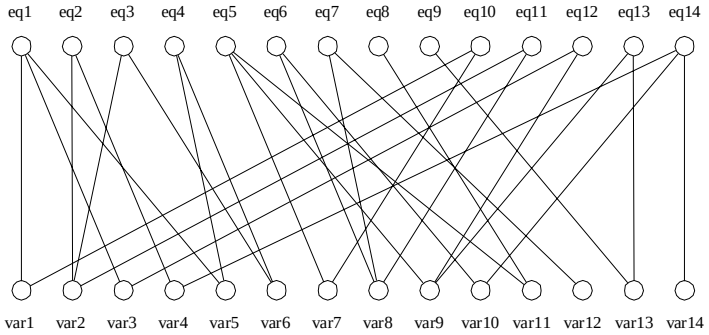



**Figure 4-1**. The associated bipartite graph of the simple electrical circuit model from Chapter 2

The incidence matrix $\mathbf{M}(G)$ corresponding to the above presented bipartite graph is given below:

| | var1 | var2 | var3 | var4 | var5 | var6 | var7 | var8 | var9 | var10 | var11 | var12 | var13 | var14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| eq1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| eq2 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| eq3 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| eq4 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| eq5 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| eq6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| eq7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| eq8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| eq9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| eq10 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| eq11 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| eq12 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| eq13 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| eq14 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

**Figure 4-2.** The incidence matrix $\mathbf{M}(G)$ of the bipartite graph from Figure 4-1

## 4.3 Bipartite Matching Algorithms.

We introduce the following notation and definitions:

**Definition 4-3:** 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 4-4:** 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 4-5:** An edge of *G* is called *admissible* if it is contained in some maximum matching in G.

**Definition 4-6:** Let *v* and *u* be vertices of a graph G. If *v* and *u* are joined by an edge *e* then *v* and *u* are said to be *adjacent*. We denote by the notation $adj(v)$ the set of all vertices adjacent to the vertex *v*. Moreover, *v* and *u* are said to be *incident* with *e* and *e* is said to be incident with *v* and *u*. We denote by $inc_E(v)$ the set of edges that are incident with the vertex *v*.

**Definition 4-7:** A vertex *v* is *saturated* or *covered* by a matching *M* if some edge of M is incident with *v*. An unsaturated or uncovered vertex is called a *free vertex*.

**Definition 4-8:** A *perfect matching* $M_G^P$ is a matching in a graph *G* that covers all vertices of *G*. In the case of a perfect matching corresponding to a bipartite graph $G = (V_1, V_2, E)$ the following relation holds: $|G| = |M| = |\partial^1 M| + |\partial^2 M|$ where the notation $|G|$ denotes the cardinality of the graph G.

**Definition 4-9:** A path *P* in a graph *G* is a finite sequence of alternating vertices and edges, beginning and ending with a vertex $v_1 e_1 v_2 e_2 \ldots e_{n-1} v_n$ such that every consecutive pair of vertices $v_x$ and $v_{x+1}$ are adjacent and $e_x$ is incident with $v_x$ and $v_{x+1}$. Typically, when writing a path *P* we omit the vertices and use a set-like notation $P = \{e_1, e_2, \ldots e_k\}$. A simple path from vertex *u* to vertex *v* can be denoted by the notation $u \xrightarrow{*} v$.

**Definition 4-10:** A path *P* in a graph *G* is called an *alternating path* of $M_G$ if it contains alternating free and covered edges. We use the notation $u \xrightarrow{=} v$ for an alternating path from vertex *u* to vertex *v*. The notation used to represent an alternating path can be extended to the following notation $P = \{(u_1, v_1), (v_1, u_2), (u_2, v_2) \cdots (u_k, v_k)\}$ where the matching and non-matching edges are represented explicitly.

**Definition 4-11**: An alternating path *P* in a graph *G* is called a *feasible path* if the end vertices are not covered by matching edges outside the end vertices.

**Definition 4-12:** The degree of a vertex is the number of edges incident to that vertex.

**Definition 4-13:** A partial order relation, denoted by $\prec$, between two directed graphs is defined as follows: $\overline{G_1} \prec \overline{G_2} \Leftrightarrow v_1 \xrightarrow{*} v_2$ for some $v_1 \in \overline{G_1}$ and $v_2 \in \overline{G_1}$.

**Definition 4-14:** A directed graph $\overline{G} = (V, \overline{E})$ is *strongly connected* if for $\forall u, v \in V$, where *V* is a set of vertices of *G*, we have $u \xrightarrow{*} v$ and $v \xrightarrow{*} u$. The strongly connected components of $\overline{G}$ are its maximal strongly connected subgraphs.

**Definition 4-15:** Suppose that *E* is a nonempty set, and let $S_1, S_2, \ldots S_n$ be non-empty subsets of *E*. We call a family $\Im$ of subsets, denoted by $\Im(S_1, S_2, \ldots S_n)$ a collection of the subsets $S_1, S_2, \ldots S_n$. In comparison to the notion of set, the notion of the family implies that an element may appear more than once.

From the computational point of view, the equation system associated to a perfect matching is structurally well-constrained and therefore can be further decomposed into smaller irreducible blocks and sent to a numerical solver. Figure 4-3 illustrates one possible maximum matching of the bipartite graph associated to the simulation model of the simple electrical circuit presented in Chapter 2. It is worth noting that in this particular case the maximum matching is also a perfect matching of the associated bipartite graph.

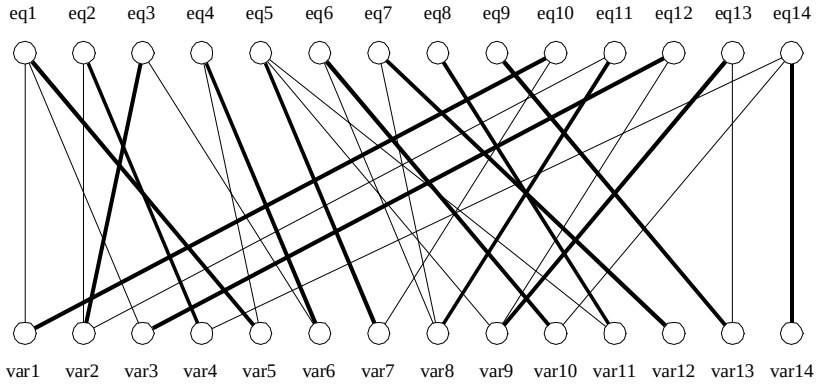



**Figure 4-3.** One possible perfect matching (marked by thick lines) of the bipartite graph associated with the electrical circuit model.

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 1973 [60]). 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 also proposed in (Fukuda and Matsui 1994 [47]), (Uno 1997 [120]) and (Uno 2001 [119]). The enumeration algorithm for all perfect matchings in bipartite graphs proposed in (Fukuda and Matsui 1994 [47]) takes $O(n^{1/2}m + mN_p)$ time where $N_p$ is the number of perfect matchings in the given bipartite graph. In (Uno 1997 [120]) and (Uno 2001 [119]) improved algorithms for finding and enumerating all perfect and maximum matchings are presented and it take only $O(n)$ and O(*log n*) time respectively per perfect matching.

It should be noted that a maximum matching or a perfect matching of a given bipartite graph are not unique. In Figure 4-4 all the possible perfect matchings of a simple bipartite graph are presented.

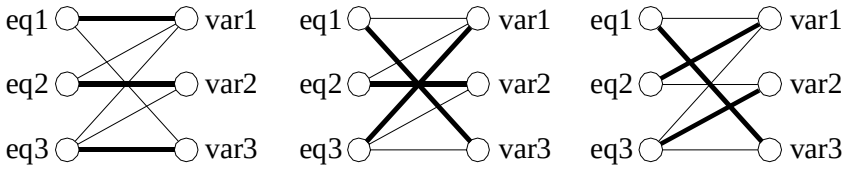



**Figure 4-4.** An example of a simple bipartite graph with all possible perfect matchings marked by thick lines.

The developed matching algorithms are based on finding alternating paths in the associated graph. A new matching associated with a bipartite graph can be obtained by exchanging matching edges with non-matching edges along an alternating cycle or along a feasible path. When exchanging edges along alternating cycles the cardinality of the obtained new matching is the same as the cardinality of the previous matching. The cardinality of the matching is not preserved when edges are exchanged along a feasible path.

**Theorem 4-1** (*Philip Hall - 1935*): Let $G = (V_1, V_2, E)$ be a bipartite graph. Then there exists a $|V_1|$ matching if and only if for all subsets S of $V_1$, $|N(S)| \geq |S|$, where $N(S) = \{u \in V_2; \exists v \in S, (v,u) \in E\}$.

A complete proof of Hall's theorem can be found in (Asratian et. al. 1998 [7]).

Regarding the existence of another perfect matching in bipartite graphs the following definition can be given:

**Theorem 4-2:** Let $M_{1G}^{P}$ be a perfect matching associated to a graph *G*. There is another perfect matching $M_{2G}^{P}$ associated to a graph *G* if there exists an alternating cycle in *G*.

In order to illustrate how a new matching can be found in a bipartite graph, based on a given perfect matching, let us consider the simple bipartite graph $G = (V_1, V_2, E)$ where $V_1 = \{u_1, u_2, u_3, u_4\}$ and $V_2 = \{v_1, v_2, v_3, v_4\}$, depicted in Figure 4-5 on the left. The associated given perfect matching is represented by the thick edges. We also show in Figure 4-5 on the right the directed graph $\overline{G}$ obtained after exchanging all the matching edges in G with bi-directional edges and orienting all other edges from $V_1$ to $V_2$.

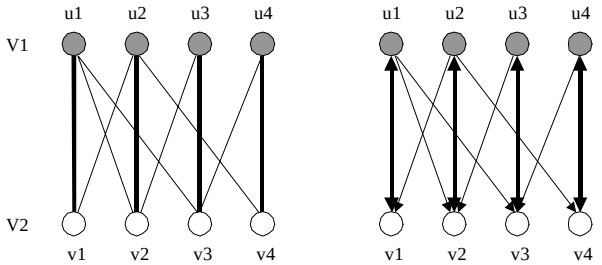



**Figure 4-5**. A simple bipartite graph with an associated perfect matching and the corresponding directed graph.

In Figure 4-6 we show the rearranged directed graph from Figure 4-5. In Figure 4-6 a) an alternating cycle and a feasible path has been found in the directed graph. In Figure 4-6 b) a new perfect matching is generated by exchanging non-matching edges with matching edges along the alternating cycle. The cardinality of the obtained new matching is the same as the cardinality of the given matching. Figure 4-6 c) illustrates the situation where a new matching is obtained by exchanging edges along the feasible path. The cardinality of the obtained new matching is not preserved any more, it is decreased by 1.

If the feasible path has an even length, the following theorem regarding the existence of a new maximum matching in bipartite graphs can be formulated (Asratian et. al. 1998 [7]):

**Theorem 4-3:** A maximum matching $M_G^{\max}$ can be obtained from any other maximum matching $N_G^{\max}$ by a sequence of transfers along alternating cycles and paths of even length.

A transfer means that the edge types in a subgraph switch position. For example in a subgraph of Figure 4-6 a) the type of edges have been rotated one step counterclockwise along a cycle and the graph from Figure 4-6 b) have been obtained. Likewise, the graph from Figure 4-6 c) have been obtained by switching the nonmatching edges with matching edges along the feasible path of the subgraph depicted in Figure 4-6 a).

For a demonstration of this theorem see (Asratian et. al. 1998 [7]). Regarding the perfect matchings in a bipartite graph the following corollary can also be given:

**Corollary 4-1:** If $M_G^p$ is a perfect matching of *G*, any other perfect matching can be obtained from $M_G^p$ by a sequence of transfers along alternating cycles relative to $M_G^p$.

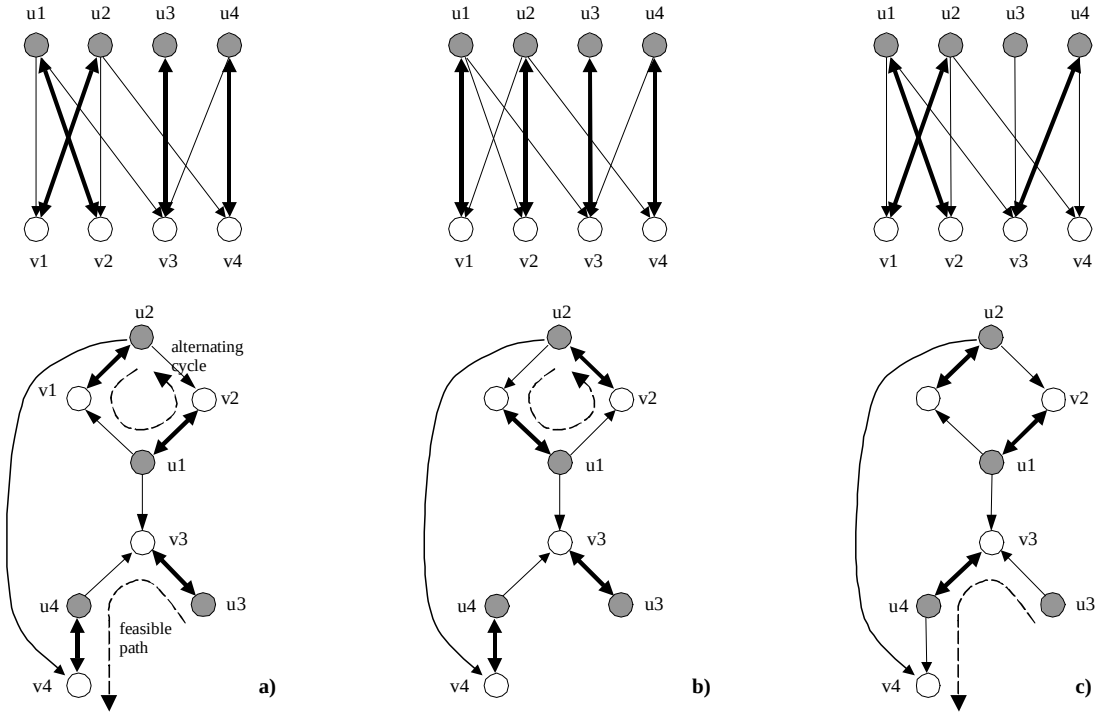



**Figure 4-6.** Finding new matchings in a directed bipartite graph.

## 4.4 Dulmage – Mendelsohn's Canonical Decomposition

In this section we shall present a structural decomposition of a bipartite graph associated with a simulation model that relies on vertex coverings. The algorithm is due to (Dulmage and Mendelsohn 1963 [33]) and canonically decomposes any maximum matching of a bipartite graph into three distinct parts: over-constrained, under-constrained, and well-constrained. Let us consider a system of linear equations its associated bipartite graph as presented in Figure 4-7. A possible maximum matching *M* is represented by the thick edges.

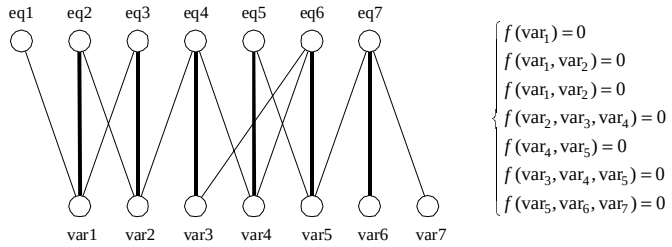



**Figure 4-7.** Dulmage-Mendelsohn's canonical decomposition of a bipartite graph associated to an equation system.

In the next step of our analysis, we exchange all the edges that are included in the matching *M* with bi-directional edges and orient all other edges from equation nodes to variable nodes. The following graph depicted in Figure 4-8 is obtained.

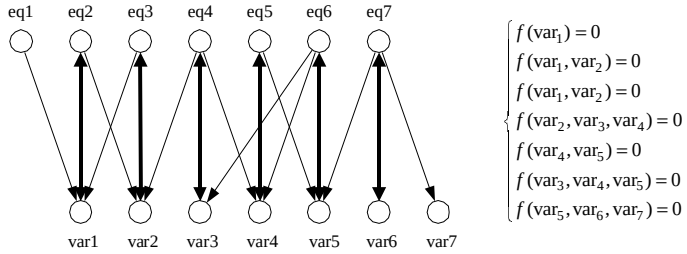



**Figure 4-8.** Oriented bipartite graph

Starting from the equation nodes that are not covered by the matching we compute the set of all nodes that are reachable from the free nodes and isolate the obtained subgraph. In a similar way, for the free variable nodes we compute 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 & 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 4-9.

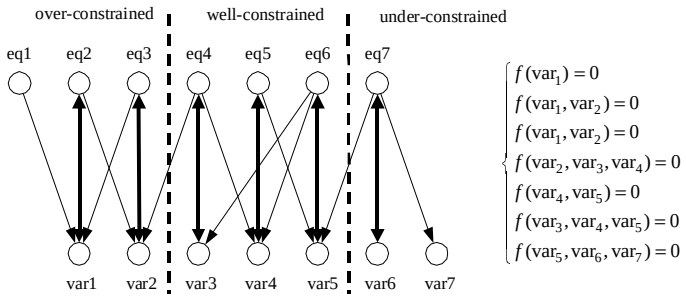



**Figure 4-9.** Canonical bipartite graph decomposition

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 canonical decomposition of bipartite graphs is given below and results in three distinct parts of the graph: the over-, under- and well-constrained parts. Furthermore, the algorithm decomposes the well-constrained part into irreducible components and establishes a partial order relation among them.

**Algorithm 4-1: Dulmage and Mendelsohn canonical decomposition**

**Input Data:** A bipartite graph $G = (V_1, V_2, E)$;

**Result:** three subgraphs: well-constrained $W_G$, over-constrained $O_G^{k+}$ and under-constrained $U_G^{k-}$.

**begin:**

- Compute the maximum matching $M_G^{\max}$ of $G = (V_1, V_2, E)$.
- Compute the directed graph $\overline{G} = (V_1, V_2, \overline{E})$ where $\overline{E}$ 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 $V_2$.
  $\overline{E} = \{(\overleftrightarrow{u,v}) \mid (u,v) \in \mathcal{E}(M_G^{\max}) \quad and \quad (\overrightarrow{u,v}) \mid (u,v) \in E - \mathcal{E}(M_G^{\max})\}$
- Let $O_G^{k+}$ be the set of all descendants of the *k* sources of the directed graph $\overline{G}$.
  $O_G^{k+}$ is the over-constrained subgraph of *G* induced on
  $\{v \in V_1 \cup V_2 \mid u \xrightarrow{*} v \quad on \quad \overline{G} \ for\ some\ u \in V_1 - \partial^1 M\}$
- Let $U_G^{k-}$ be the set of all ancestors of *k* sink of the directed graph $\overline{G}$.
  $U_G^{k-}$ is the over-constrained subgraph of *G* induced on
  $\{v \in V_1 \cup V_2 \mid v \xrightarrow{*} u \quad on \quad \overline{G} \ for\ some\ u \in V_2 - \partial^2 M\}$
- Calculate $W_G = \overline{G} - O_G^{k+} - U_G^{k-}$. $W_G$ is obtained by deleting from $\overline{G}$ all the vertices and edges of $O_G^{k+}$ and $U_G^{k-}$.
- Compute the strongly connected components $S_{Gs}$ *(s=1…n)* of $W_G$.[3]
- Compute the subgraphs $W_{Gs}$ of $W_G$ induced on $S_{Gs}$ *(s=1…n)*
- Define the partial order on $W_{Gs}$ *(s=1…n)*
- Define the partial order $O_G^{k+} \prec W_{Gs} \prec U_G^{k-}$ for any s.

**end.**

[3] This can be achieved by using a linear time method such as (Tarjan [94]), see section 4.5.

The *over-constrained* part: the number of equations in the system is greater than the number of variables. The additional equations are either redundant or contradictory and thus yield no solution. A possible error fixing strategy is to remove the additional over-constraining equations from the system in order to make the system well-constrained. Even if the additional equations 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 *under-constrained* part: the number of variables in the system is greater than the number of equations. A possible error fixing strategy would be to initialize some of the variables in order to obtain a well-constrained part or add additional equations to the system.

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

The *well-constrained* part: the number of equations in the system is equal to the number of variables and therefore the mathematical system of equations is structurally sound, having a finite number of solutions. This part can 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 numerically solving the well-constrained part means that no valid solution exists and somewhere there is numerical redundancy in the system.

The concept of D&M decomposition can be extended to the incidence matrix corresponding to the bipartite graph. In this way, we obtain the block lower triangular form of the matrix. Thus, instead of solving the whole system of equations once, the system can be solved by solving a sequence of smaller systems. Experience has shown that sequentially solving the separate blocks is far more robust than solving the whole system of equations at once. From the dynamic debugging perspective, by solving a sequence of smaller systems, numerical failures can be isolated to the smaller blocks and thus become easier to find.

We illustrate the block lower triangular form of the incidence matrix corresponding to the `DCMotor` model from Chapter 2, see Figure 4-10. Applying the D&M decomposition on the bipartite graph representing the flattened system of equations corresponding to the simulation model we obtain a single well-constrained graph with 14 strongly connected components. Thirteen of these strongly connected components contain only one equation and one big component contains 23 equations that need to be solved simultaneously. Big blocks obtained after the D&M decomposition can often be broken down into smaller blocks using tearing techniques (Mah 1990 [79]), (Elmqvist and Otter 1994 [34]).

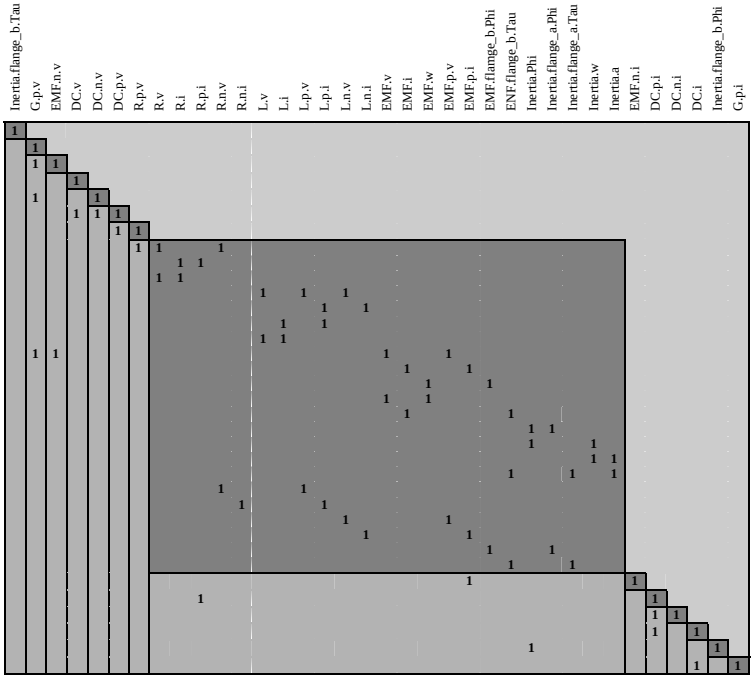



**Figure 4-10.** Block Lower Triangular form of the `DCMotor` model obtained after applying the D&M decomposition on the flat form of the equations.

Below, we present the output produced by the implemented debugger with the equations corresponding to the block lower triangular form of the simple `DCMotor` circuit. The equations are presented to the left while the variables that are solved for are given on the right and emphasized with italics letters. For the large block [7] the list of variables appears below the equations.

```
Strongly connected components: 14
STRONGLY CONNECTED COMPONENTS
[0]Inertia.flange_b.Tau == 0                    Inertia.flange_b.Tau
[1]G.p.v == 0                                   G.p.v
[2]EMF.n.v == G.p.v                             EMF.n.v
[3]DC.v == DC.V                                 DC.v
[4]G.p.v == DC.n.v                              DC.n.v
[5]DC.v == -DC.n.v + DC.p.v                     DC.p.v
[6]DC.p.v == R.p.v                              R.p.v
[7]R.v == -R.n.v + R.p.v
   0 == R.n.i + R.p.i
   R.i == R.p.i
   R.i R.R == R.v
   L.v == -L.n.v + L.p.v
   0 == L.n.i + L.p.i
   L.i == L.p.i
   L.L (L.i)'  == L.v
   EMF.v == -EMF.n.v + EMF.p.v
   EMF.i == EMF.p.i
   EMF.w == (EMF.flange_b.Phi)'
   EMF.k EMF.w == EMF.v
   EMF.flange_b.Tau == -EMF.i EMF.k
   Inertia.flange_a.Phi == Inertia.Phi
   Inertia.w == (Inertia.Phi)'
   Inertia.a == (Inertia.w)'
   Inertia.a Inertia.J == Inertia.flange_a.Tau + Inertia.flange_b.Tau
   R.n.v == L.p.v
   L.p.i + R.n.i == 0
```

```
   L.n.v == EMF.p.v
   EMF.p.i + L.n.i == 0
   EMF.flange_b.Phi == Inertia.flange_a.Phi
   EMF.flange_b.Tau + Inertia.flange_a.Tau == 0

   R.v  R.i  R.p.i  R.n.v  R.n.i  L.v  L.i  L.p.v  L.p.i  L.n.v
   L.n.i EMF.v  EMF.i  EMF.w  EMF.p.v  EMF.p.i  EMF.flange_b.Phi
   EMF.flange_b.Tau  Inertia.Phi  Inertia.flange_a.Phi
   Inertia.flange_a.Tau  Inertia.w  Inertia.a
[8]0 == EMF.n.i + EMF.p.i                  EMF.n.i
[9]DC.p.i + R.p.i == 0                     DC.p.i
[10]0 == DC.n.i + DC.p.i                   DC.n.i
[11]DC.i == DC.p.i                         DC.i
[12]Inertia.flange_b.Phi == Inertia.Phi    Inertia.flange_b.Phi
[13]DC.n.i + EMF.n.i + G.p.i == 0          G.p.i
```

**Figure 4-11**. Output of the strongly connected components corresponding to the `DCMotor` circuit model.

If the model is not correctly formulated and presents over and under-constrained components they will appear at the beginning and at the end of the sparsity pattern respectively as shown in Figure 4-12. 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 over and under-constrained blocks need to be made square blocks by eliminating extra equations, and eliminating extra variables respectively [4].

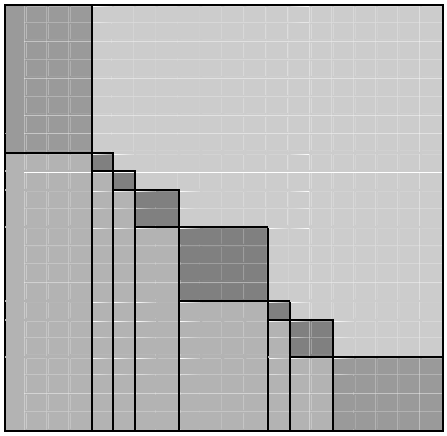


**Figure 4-12.** D&M decomposition with one under-constrained and one over-constrained block at the beginning and at the end of the incidence matrix.

Our structural analysis algorithms, employed in the following chapters, will 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.

[4] This constitutes the naïve approach to the over and under-constrained problems. The following chapters will provide additional details on how to systematically debug such components.

## 4.5 The Algorithm for Computing the Strongly Connected Components

For computing the strongly connected components of a directed graph, our debugger uses an implementation of Tarjan's algorithm (Tarjan 1972 [113]) (Duff et. al. 1986 [32]) available in the LEDA library (Mehlhorn and Näher 1999 [83]). The algorithm has a time complexity of $O(|V|+|E|)$ where $|V|$ is the number of nodes and $|E|$ is the number of edges in the analyzed directed graph. Several improvements of the original algorithm have been proposed in the literature (Nuutila and Soininen 1993 [91]).

By using a strongly connected components decomposition algorithm the original problem is decomposed into smaller problems. Then the solution of the original problem can be constructed by combining the solutions of the subproblems. As was illustrated in Figure 4-11 the numerical solutions of the underlying system of equation representing a `DCMotor` can be computed by combining the solution of the strongly connected blocks from [0] to [13] instead of solving the whole system as one simultaneous system of equations.

According to Definition 4-14, in a strongly connected directed graph every two vertices are reachable from each other. The strongly connected components represent all the possible strongly connected subgraphs. In the case of undirected graphs the notion of connected graph is used. An undirected graph is connected if every two pairs of vertices are connected by a path and the connected components represent all the connected subgraphs.

From the structural analysis and system diagnosis point of view it is interesting to take a closer look at how the strongly connected components are identified in directed bipartite graphs with an associated perfect matching. Let us again consider the directed bipartite graph from Figure 4-6, which is shown in Figure 4-13.

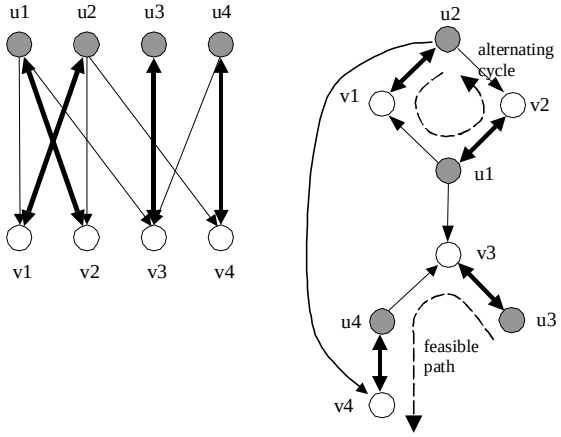



**Figure 4-13.** Simple directed bipartite graph.

In our abstraction, according to Definition 4-14 all alternating cycles and admissible edges from feasible paths represent strongly connected components. Then, we eliminate all the edges from the directed graph that are not part of any cycles or admissible edges of a feasible path. The strongly connected components are isolated in this way and the partial order relation among the components is established based on the eliminated

edges. The partial order relation has an inverted direction on the blocks compared to the direction of the eliminated edges. When the directed bipartite graph represents a system of equations, the partial order relation implies transport of variable values from one block to another block. In this way, a dependency is created among the blocks and an order of solving the blocks is implicitly established.

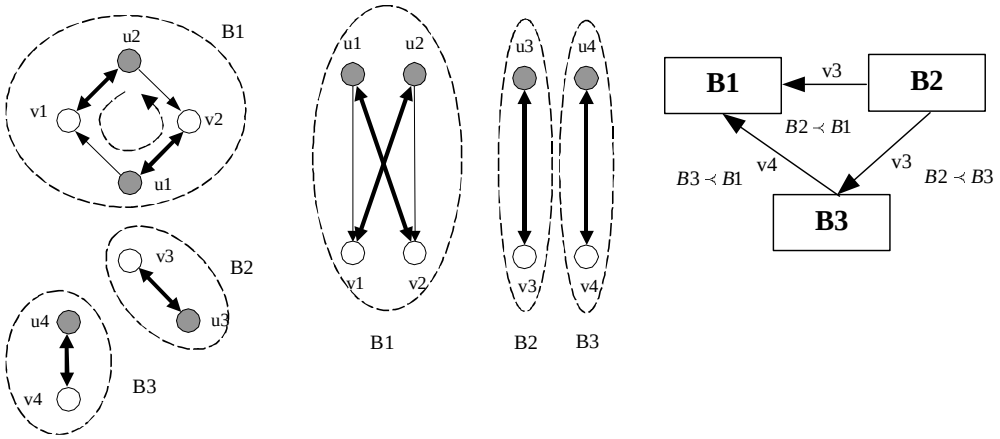



**Figure 4-14.** Strongly connected components of a directed graph and the partial order relation between the components.

Let us consider the example in Figure 4-14. The nodes marked *u* represent equations and the nodes marked *v* represent variables. Then the partial order graph from Figure 4-14 is interpreted as follows: first the block B2 needs to be solved. Then through the partial order relation $B2 \prec B1$ and $B2 \prec B3$ the value of variable *v3* is transported to the blocks B1 and B3. At the next step B3 is solved which transports the value of variable *v4* to block B1 through the partial order relation $B3 \prec B4$. In the last step block *B1* can be solved. In this way the order in which the strongly connected components are solved is given by the partial order graph of the strongly connected components.

## 4.6 Symbolic Manipulation of Equation Systems

In this section we illustrate how a simple equation system is transformed into a reduced equivalent form by symbolically substituting one variable by another. Such techniques are widely used when compiling equation-based languages in order to reduce the size of the problem (Maffezzoni et. al. 1996 [78]) and to split bigger blocks obtained after the canonical decomposition into smaller parts. We are going to show the graph transformation involved in the substitution of one variable by another when simple equality equations in the form $x_i = x_j$ are encountered in the overall system of equations.

Let us consider the simple equation system shown in Figure 4-15 and its corresponding bipartite graph with an arbitrary given matching. The directed bipartite graph is also shown where all the matching edges have been exchanged by bidirectional edges and all the other edges have been oriented from the equation nodes to the variable nodes.

$$\begin{cases} x = y \\ z - y = 1 \\ x + z = 7 \end{cases}$$

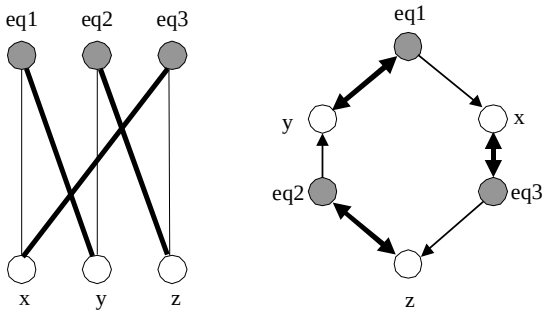



**Figure 4-15.** A simple equation system and its corresponding undirected and directed bipartite graph.

When attempting to manually solve the system of equations, one may take the first equation and substitute all the occurrences of *x* with *y* in the remaining equations. The following transformation of the system of equations can be performed:

$$\begin{cases} x = y \\ z - y = 1 \\ x + z = 7 \end{cases} \Rightarrow \begin{cases} x = y \\ z - y = 1 \\ y + z = 7 \end{cases}$$

This transformation allows us to solve the last two equations simultaneously and to obtain the values for *y* and *z*. We then substitute the obtained value of *y* into the first equation and calculate the value of *x*. Now let us take a look how this will affect the structure of the underlying directed bipartite graph.

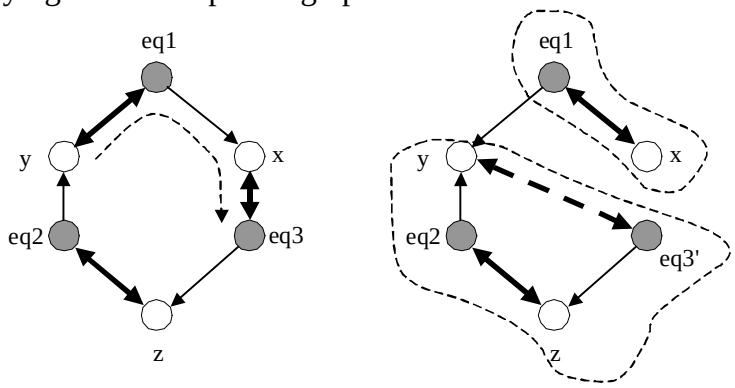



**Figure 4-16.** Graph transformation after a variable symbolic substitution

Obviously the directed graph corresponding to the untransformed system of equations constitutes a strongly connected component. By substituting variable *x* with *y* in all other equations we obtain an equivalent system of equations that will exhibit two strongly connected components in the structure of the corresponding bipartite graph instead of one strongly connected component.

A symbolic substitution of a variable at the bipartite graph level is done in the following way: first the equation node that has the special structure $x_i = x_j$ of a simple equality is identified (in our particular case *eq1*). Then we identify the first reachable equation node from the equation chosen in the previous step (*eq3* for our example) and the first variable node (*y*) that sinks into the chosen equation node. The path from the variable node that sinks into the equation node and the equation node is an alternating path.

We exchange the matching edges with non-matching edges along this path and eliminate the incident edge to the final equation node. That edge was previously part of the alternating path (in our case the edge between variable node x and the equation node *eq3*). Then an extra directed matching edge is added between the first variable node and the final equation node (the edge represented by dashed lines in Figure 4-16 between *y* and *eq3*). The final equation node is relabeled (*eq3'*) indicating that the form of the equation has been altered, by symbolically substituting a variable for another variable. When performing a strongly connected component analysis algorithm on the obtained bipartite graph, two components are detected now instead of only one as is indicated in Figure 4-16 on the left.

## 4.7 Tree Search Algorithms

Many of the algorithms presented or mentioned in the subsequent chapters that are related to the debugging of over and under-constrained systems require a systematic method of visiting the vertices of a tree. These include Tarjan's algorithm (Tarjan 1972 [113]) for computing the strongly connected components or (Duff et. al. 1986 [32]) There are two well-known methods for searching trees: *depth-first search* (DFS) and *breadth-first search* (BFS) algorithms. Both algorithms list the nodes in the order in which they are encountered during the traversal and differ only in the way in which the node lists are constructed. For these reasons we find it useful to briefly present the depth-first searching algorithm (Gibbons 1985 [48]) where DFN(*v*) is a label associated with each vertex *v*.

**Algorithm 4-2: DFS(*G,n*) Depth-First Search Algorithm**

**Input Data:** A graph $G = (V, E)$ and start node $n \in V$ .
**Result:** A list *L* of the traversed edges
**begin:**

```
Procedure dfs(v)
  begin
    DFN(v) := i;
    i := i + 1;
    for all v' ∈ adj(v) do
      if DFN(v') =0 then
        L := append(L, {(v, v')});
        dfs(v');
      end if
    end for
  end procedure.
  for all v ∈ V do DFN(v) := 0;
  L = ∅ ;i := 1;
  dfs(n);
  return L;
end.
```

The complexity of the algorithm is $O(|E|)$. For the simple labeled tree presented in Figure 4-17 the output of Algorithm 4-2 is:

*L* = { *(a,b), (b,c), (b,d), (d,e), (d,f), (a,g), (g,h), (g,i)* }

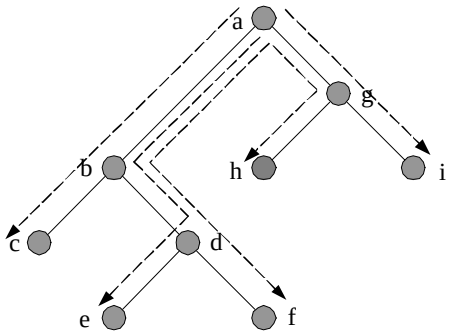



**Figure 4-17.** A simple labeled tree.

The algorithm **DFSm** is a slightly modified version of Algorithm 4-2 where the output is changed. Instead of enumerating the edges in the traversal order the paths from the start node to the terminal nodes, indicated by the dashed arrows in Figure 4-17, are represented explicitly in the output list. The output list *L* will have the following form:

*L* = { *(a,b,c), (a,b,d,e), (a,b,d,f), (a,g,h) ,(a,g,i)* } for DFSm.

# Chapter 5

# Debugging Over-Constrained Systems of Equations

***Summary***: *This chapter proposes a method for detecting the situations when over-constraining equations are present in a system simulation model specification. It primarily describes related graph-based algorithms, which constitute the core of the implemented debugging kernel for detecting such situations. An efficient way to annotate the underlying equations in order to help the implemented debugger to choose the right error fixing solution is also provided. This also provides means to report the location of an error caught by the static analyzer consistent with the user's perception of the simulation model. At the end of this chapter, special emphasis is given to over-constrained bipartite graphs with multiple free vertices which corresponds to heavily over-constrained physical system model. The algorithms and methods developed help to statically detect and repair a broad range of errors without having to execute the simulation model. Several simulation models and examples are given in this chapter in order to illustrate the main situations when over-constraining equations can appear in the system. Error detection and error solving strategies for these cases are also presented.*

## 5.1 Introduction

A typical problem which often appears in physical system modeling and simulation is when too many equations are specified in a system, leading to an inconsistent state of the simulation model. In such situations the model cannot be compiled into the form compatible with the numerical solvers. Therefore the numerical solver cannot start to find the correct solutions to the underlying system of equations. The methods proposed in this chapter present a strategy to deal with overdeterminancy by identifying the minimal set of equations that should be removed from the system in order to make the remaining set of equations solvable. The idea is to isolate the over-constraining part of the bipartite graph associated to the underlying system of equations and to perform reasoning based on specific properties of the specified subgraph. Efficient graph transformations, based on rules derived from the semantics of the modeling language are also performed on the subgraphs. We are going to show how these rules are automatically

derived from the modeling language semantics and how the associated annotations to the equations contribute to the filtering of the combinatorial explosion of possible error fixing solutions.

The possibility of a systematic approach to the elimination of over-constraining equations also needs to be considered because sometimes multiple error fixing strategies are simultaneously available as a result of the graph transformations. These error fixing strategies are always sound from the mathematical point of view (they will lead to a structural consistent system of equations) but they might not make sense from a physical modeling point of view (their behavior does not match the intended behavior) and in that case intervention of a human expert is needed.

## 5.2 A Simple Electrical Circuit Model

Obviously, each simulation problem is associated with a corresponding mathematical model. In dynamic continuous system simulation the mathematical model is usually represented by a mixed set of algebraic equations and ordinary differential equations. For some complicated simulation problems the model can be represented by a mixed set of ordinary differential equations (ODEs), differential algebraic equations (DAEs) and partial differential equations (PDEs). Simulation models can become quite large and complex, sometimes involving several thousand simultaneous equations.

The system of equations describing the overall model is obtained by merging the equations of all component models and all binding equations generated by `connect` equations.

The simple electrical circuit model from Chapter 3, section 3.5.1, is reiterated here. In Figure 5-1 the Modelica source code corresponding to the simple simulation model from section 3.5.1 consisting of a resistor connected in parallel to a sinusoidal voltage is given. The intermediate flattened form is also shown for explanatory purposes. The `Circuit` model is represented as an aggregation of instances of the `Resistor`, `Source` and `Ground` submodels connected together by means of connections between physical ports.

This model is so trivial as to be almost beneath consideration, but it serves as a straightforward vehicle for the introduction of several fundamental debugging concepts. This simulation model and this modeling example will be reiterated many times in later chapters with the purpose of illustrating concepts of structural analysis. The model is extremely useful because it keeps the associated structural graphs to a minimum size and complexity, but in the meantime it illustrates interesting structural and debugging problems.

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

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

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

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

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

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

```
Flattened equations
1.    R1.v = -R1.n.v + R1.p.v
2.    0 = R1.n.i + R1.p.i
3.    R1.i = R1.p.i
4.    R1.i*R1.R = R1.v
5.    AC.v = -AC.n.v + AC.p.v
6.    0 = AC.n.i + AC.p.i
7.    AC.i = AC.p.i
8.    AC.v =  AC.VA*Sin[2*time*AC.f*AC.PI]
9.    G.p.v = 0
10.   AC.p.v = R1.p.v
11.   AC.p.i + R1.p.i = 0
12.   R1.n.v = AC.n.v
13.   AC.n.v = G.p.v
14.   AC.n.i + G.p.i + R1.n.i = 0

Flattened Variables
1. R1.p.v    2. R1.p.i    3. R1.n.v
4. R1.n.i    5. R1.v      6. R1.i
7. AC.p.v    8. AC.p.i    9. AC.n.v
10. AC.n.i   11. AC.v     12. AC.i
13. G.p.v    14. G.p.i

Flattened Parameters
R1.R -> 10
AC.VA -> 220
AC.f -> 50

Flattened Constants
AC.PI -> 3.14159
```

**Figure 5-1.** Modelica source code of a simple simulation model and its corresponding flattened system of equations, variables, parameters, and constants.

From the flattened intermediate form of the equations the associated bipartite graph is derived and a perfect matching is found. Since there exists a perfect matching associated to the bipartite graph, the D&M decomposition leads to a well-constrained part without any over or under-constrained parts. Given the bipartite graph and a corresponding perfect matching a directed graph is derived as shown in Figure 5-2 a). Subsequently, an algorithm for detecting the strongly connected components is executed on the obtained graph. The algorithm will decompose the system into irreducible blocks with the dependencies shown in Figure 5-2 b). By applying the algorithm for computing the strongly connected components in the directed graphs, the solving order of the equations can also be derived, as shown in Table 5-1 by the block lower triangular form of the incidence matrix corresponding to the system of equations.

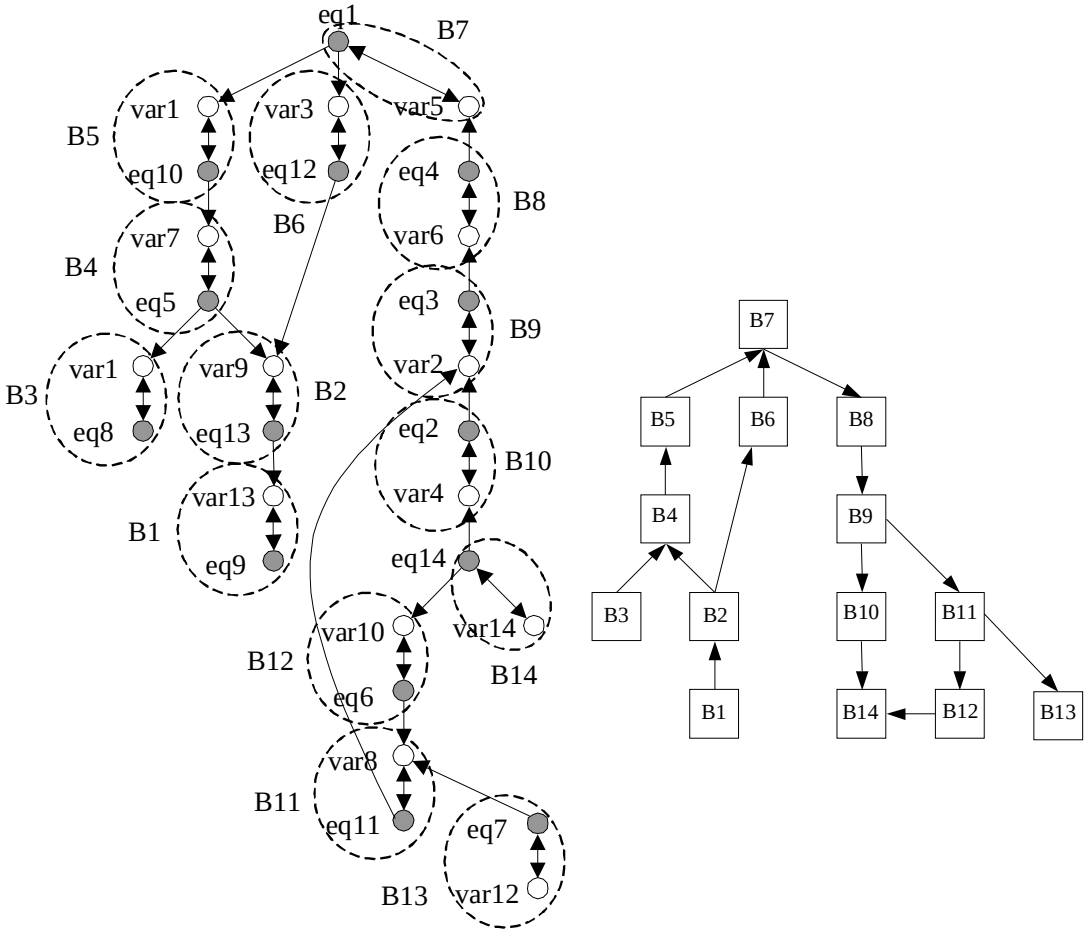



**Figure 5-2. a)** Directed graph associated to the simple electrical circuit and **b)** the corresponding decomposition into irreducible blocks containing one equation each.

**Table 5-1.** Block Lower Triangular form of the equation system .

| | G.p.v | AC.n.v | AC.v | AC.p.v | R.p.v | R.n.v | R.v | R.i | R.p.i | R.n.i | Ac.p.i | AC.n.i | AC.i | G.p.i |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| eq9 | X | | | | | | | | | | | | | |
| eq13 | X | X | | | | | | | | | | | | |
| eq8 | | | X | | | | | | | | | | | |
| eq5 | | X | X | X | | | | | | | | | | |
| eq10 | | | | X | X | | | | | | | | | |
| eq12 | | X | | | | X | | | | | | | | |
| eq1 | | | | | X | X | X | | | | | | | |
| eq4 | | | | | | | X | X | | | | | | |
| eq3 | | | | | | | | X | X | | | | | |
| eq2 | | | | | | | | | X | X | | | | |
| eq11 | | | | | | | | | X | | X | | | |
| eq6 | | | | | | | | | | | X | X | | |
| eq7 | | | | | | | | | | | X | | X | |
| eq14 | | | | | | | | | | X | | X | | X |

It should be noted that the directed graph derived from the bipartite graph associated to the system of equations only contains cycles that involve one equation and one variable node and does not contain any alternating cycles. Therefore the irreducible blocks, ob-

tained after applying the algorithm for detection of the strongly connected components, only contain single equations. In principle, as was mentioned earlier, the algorithm for detecting the strongly components operates by deleting all the edges in the directed graph which are not part of any possible perfect matching. In our case, the perfect matching associated to the directed graph was unique because there were no alternating cycles in it and all the edges not part of the perfect matching were deleted. In this case, the algorithm for computing the strongly connected components in the graph was only useful to establish the order in which the equations should be solved.

## 5.3 Equation Annotations

In order to provide a mechanism to reason about the erroneous model under consideration the equations need to be annotated. For annotating the equations we use a structure which resembles the one developed in (Flannery and Gonzales 1997 [39]). We define an annotated equation as a record with the following structure: `<Equation, Name, Description, No. of associated eqs., Class name, Flexibility level, Connector generated >.`

An example including the annotations associated with an equation extracted from the `Resistor` model is given in Table 5-2. 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.

**Table 5-2.** 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*" |
| Nr. of associated eq | 1 |
| Class Name | "*Resistor*" |
| Flexibility Level | 3 |
| Connector generated | no |

The *Class Name* says 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.

The *No. of associated eqs.* field defines the number of equations which are specified together with the annotated equation. In the example given in Table 5-2 the *No. of associated eqs.* is equal to one 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. If, during debugging, the equation `R1.i * R1.R == R1.v` is diagnosed to be an over-constraining equation and therefore needs to be eliminated, the elimination is not possible because the model will be invalidated (the *No. of associated eqs.* cannot be equal to 0) and therefore other solutions need to be investigated.

The *flexibility level*, in a similar way as defined in (Flannery and Gonzales 1997 [39]) 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. Equations, which are coming from a partial model and therefore are inherited by the final model, have a greater rigidity compared to the equations defined in the final model. For example, in practice, it turns out that the equations generated by connections are more rigid from the constraint relaxation point of view than the equations specified inside the model. Taking into account these formal rules, for an equation defined inside a Modelica class, a maximum flexibility value will be assigned. In conclusion a maximum flexibility value will be defined for the equations in the final model, followed by equations defined in partial classes and equations generated by the `connect` equations. We set the flexibility value to 0 for those equations that should not be removed or modified. These equations are *locked* for editing.

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

It is worth noting that the annotation attributes are automatically initialized by the static analyzer, incorporated in the front-end of the compiler, by using several graph representations (Harrold and Rothermel 1996 [57]) of the declarative object-oriented program code. Therefore the user does not need to manually annotate the source code. A debugger preprocessor takes care of the automatic generation and initialization of the annotating code. In this way a mapping between the intermediate code and original declarative code is kept during the translation phases.

## 5.4 Detecting an Over-Constraining Equation

In the following sections methods for iterating through all possible equations that can be removed from the system are presented. These methods are integrated in an efficient framework that permits a filtering strategy of the equations based on annotations and semantic rules derived from the source language, in this case Modelica. Several properties of the over-constraining subgraphs obtained after the D&M canonical decomposition will also be presented in the following sections.

The main idea behind the framework of debugging over-constrained situations is to filter out certain equations. Equations are eliminated based on combinatorial properties of the bipartite graphs or if they are violating the semantic rules of the modeling language. Only those error-fixing solutions (in the case of over-constraining systems elimination of extra equations) are taken into account that can be performed by manipulating language constructs in the modeling language in terms of atomic changes such as those defined in Chapter 3, section 3.4. The debugger exposes the users to the modeling language and does not burden users with information connected to the intermediate language involved in the translation phase.

Let us again examine the simple simulation example presented in Figure 5-1, where an additional equation (`i=23`) was intentionally introduced inside the `Resistor` com-

ponent in order to obtain a generally over-constrained system. The D&M canonical decomposition will lead to two different subgraphs: a well-constrained part $W_G$ and an over-constrained part $O_G^{1+}$ (see Figure 5-3).

```
"eq1"   R1.v == -R1.n.v + R1.p.v
"eq2"    0 == R1.n.i + R1.p.i
"eq3"   R1.i == R1.p.i
"eq4"   R1.i R1.R == R1.v
"eq5"   R1.i == 23
"eq6"   AC.v == -AC.n.v + AC.p.v
"eq7"   0 == AC.n.i + AC.p.i
"eq8"   AC.i == AC.p.i
"eq9"   AC.v == AC.VA*sin[2*time*AC.f*AC.PI]
"eq10"  G.p.v == 0
"eq11"  AC.p.v == R1.p.v
"eq12"  AC.p.i + R1.p.i == 0
"eq13"  R1.n.v == AC.n.v
"eq14"  AC.n.v == G.p.v
"eq15"  AC.n.i + G.p.i + R1.n.i == 0
```

```
"var1"   R1.p.v
"var2"   R1.p.i
"var3"   R1.n.v
"var4"   R1.n.i
"var5"   R1.v
"var6"   R1.i
"var7"   AC.p.v
"var8"   AC.p.i
"var9"   AC.n.v
"var10"  AC.n.i
"var11"  AC.v
"var12"  AC.i
"var13"  G.p.v
"var14"  G.p.i
```

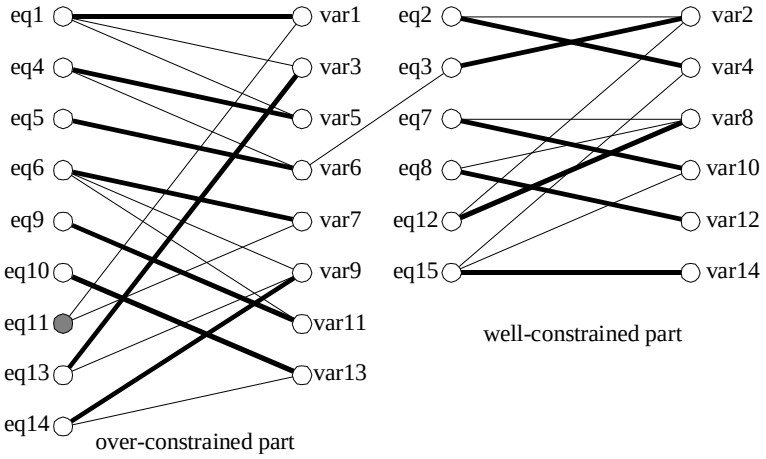



**Figure 5-3.** Canonical decomposition of an over-constrained system

Equation *eq11* is a non-saturated or free vertex of the equation set, therefore it is a source for the over-constrained part $O_G^{1+}$ . Starting from *eq11*, the directed graph can be derived from the undirected bipartite graph, as illustrated in Figure 5-4, by exchanging all the matching edges into bidirectional edges and orienting all other edges from equation to variable nodes. An immediate fix of the over-constrained part is to eliminate *eq11*, which will lead to a well-constrained part. However, first we need to examine the annotations associated to *eq11*:

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

We note that the equation is generated by a `connect` equation from the `Circuit` model and the only way to remove the equation 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, which is indicated by the *No. of associated eqs.* = 2 parameter. One should also note that the *flexibility level* of the equation is equal to 1, which is extremely low, indicating that the equation is extremely rigid. Therefore another solution need to be found, removing another equation instead of removing the equation `AC.p.v == R1.pv`.

## 5.5 Over-Constrained Bipartite Graph Properties

The general error fixing strategy in the case of over-constrained equation subsystems is to remove the extra equations. Therefore several important criteria need to be developed based on the combinatorial properties of the over-constraining subgraph and the filtering rules imposed by the modeling language semantics.

Let $G = (V_1, V_2, E)$ be a bipartite graph and $M_G^{\max}$ an associated maximum cardinality matching. The directed graph obtained after exchanging all matching edges in bi-directional edges and orienting all other edges from $V_1$ to $V_2$ is $\overline{G} = (V_1, V_2, \overline{A})$ where $\overline{A} = \{(\overleftrightarrow{u,v}) \mid (u,v) \in \varepsilon(M)$ and $(\overrightarrow{u,v}) \mid (u,v) \in E - \varepsilon(M)\}$ or described equivalently as $\overline{A} = \{(\overrightarrow{u,v}) \mid (u,v) \in \varepsilon(G)$ and $(\overrightarrow{v,u}) \mid (v,u) \in \varepsilon(M)\}$. We denote by $\varepsilon(G)$ the edge set of graph *G*.

**Definition 5-1**: The removal of a node representing an equation from a given bipartite graph *G* with an associated maximum cardinality matching $M_G^{\max}$, is *safe* if:

1. The corresponding undirected graph obtained after removing the node and its incident edges remains connected.
2. The node is not covered by the considered maximum cardinality matching $M_G^{\max}$.

By the second criterion, it should be noted that only those nodes can be considered for removal that are not covered by the maximum cardinality matching. For this reason, those bipartite graphs which admit a perfect matching have no equation nodes in their structure that are safe for removal. The second criterion also restricts the nodes which are safe for removal to the free nodes contained in the over constrained subgraph $O_G^{k+}$ corresponding to *G*. Therefore we can extend Definition 5-1 to the family of all possible equation nodes that are safe to be removed by considering the family of all maximum cardinality matchings $\Im(M_G^{\max})$. The following theorem can be formulated regarding the safety of equation nodes in $O_G^{k+}$.

**Theorem 5-1.** Any equation node $v$ of the over-constraining subgraph $O_G^{k+}$ corresponding to the bipartite graph G *is safe for removal* if by removing that equation and the corresponding incident edges $inc_E(v)$ the remaining undirected graph is connected.

**Proof:** We shall demonstrate that there is always a maximum cardinality matching $M_G^{\max}$, which does not include a given equation node $v \in V_1$, such that the second condition of Definition 5-1 always holds. Based on the D&M decomposition (Algorithm 4-1) the $O_G^{k+}$ subgraph is the set of all descendants of sources or free nodes of the bipartite graph G covered by a given maximum matching $M_G^{\max}$. A path constructed by starting from a free node and following the matching always starts with an unmatched node to a node representing the second bipartition which is covered by the matching. Then the matching edge is added to the path and the direction pointing to a node from the first bipartition is followed. The path construction stops when there are no directed edges to follow any more. It can stop with a node from the first bipartition covered by

the matching $M_G^{\max}$. The final node is always covered by the given maximum cardinality matching because otherwise it is not reachable. In that case, there is an over-constraining feasible path from the free starting node to the end node. The non-matching edges along this path can easily be exchanged with matching edges. In this way a new over-constraining feasible path is obtained which will cover the previous free node but will uncover another node from the first bipartition corresponding to $O_G^{k+}$. Since $O_G^{k+} \subseteq G$ all the feasible paths contained in $O_G^{k+}$ are also feasible paths in G. Exchanging the non-matching edges with matching edges along the path creates a new maximum cardinality matching associated to G. Therefore there exists a maximum cardinality matching of G which does not cover a certain node in the first bipartition of $O_G^{k+}$.

The path can also terminate when a cycle is detected. It terminates with a node from the second bipartition which was previously traversed. In this case the non-matching edges and the matching edges can also be exchanged along a feasible path excluding the last edge. Since the terminating node has already been traversed once, there is a guarantee that it is on the feasible path. By exchanging the edges, this node will always be covered by a maximum cardinality matching. Following all the free edges and all the associated feasible paths the subfamily of the maximum cardinality matching $\Im_S(M_G^{\max}) \subseteq \Im(M_G^{\max})$ associated to $G$ can be deduced. This proves that there exists at least one maximum cardinality matching which does not cover a node contained in the first bipartition of an over-constrained subgraph $O_G^{k+}$.

It should be noted that the notion of safe removal of equation nodes only refers to the bipartite graph representation of the intermediate code of the flattened set of equations. This is influenced by combinatorial properties of the bipartite graph. By including removal criteria derived from the semantics of the modeling language the notion of safe removal can further be extended to the modeling language source code.

## 5.6 Calculating the Safe Set of Over-Constraining Equations

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

**Definition 5-2:** The *equivalent over-constraining equation set* associated to an over-constrained part of a system of equations is the set of equations $\{eq_1, eq_2, \cdots eq_n\}$ from where the elimination of any of the component equations will lead to a well constrained system of equations and the associated undirected bipartite graph remains connected.

**Definition 5-3:** The *reduced equivalent over-constraining equation set* is the subset of safe equations obtained from the equivalent over-constraining equations after the constraints derived from the language semantics have been applied.

Based on the reduced over-constrained list the automated debugger can present the user several options to resolve over-constrained conflicting situations.

From the over-constrained part $O_G^{1+}$ resulting from the D&M decomposition**,** depicted in Figure 5-4, we can construct an algorithm to find the equivalent over-constraining set based on the associated directed graph of the over-constrained part.

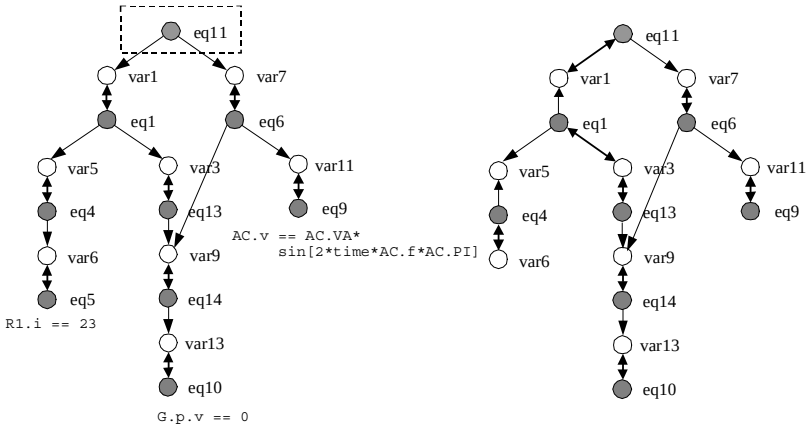



**Figure 5-4. a)** A directed graph associated to the over-constrained part starting from *eq11*. **b)** The fixed well-constrained directed graph by eliminating equation *eq5*.

We describe the algorithm as follows:

---

**Algorithm 5-1: Finding the equivalent over-constraining equations set**

**Input Data:** The directed over-constrained graph $O_G^{k+}$ resulting after D&M decomposition applied to *G*.

**Result:** The reduced equivalent over-constraining equation set *L*.

**begin:**

Initialize $L=\{\}$;

**for each** free node $v_k \in v(\overline{O_G^{k+}})$ **do**

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** $n \in v_{V_1}(O_G^{k+})$ (n is an equation node) **then**

- Remove *n* from $O_G^{k+}$

- Compute the number of strongly connected components $no_{str}$ of $O_G^{k+}$.

**if** $no_{str} == 1$ **then**

- add *n* to *L*

**endif**.

**endif**.

**endfor**.

**endif**.

Output the equivalent over-constraining equations set *L*.

**end.**

---

The basic idea behind the algorithm for enumerating all the perfect matchings in a bipartite graph, presented in (Fukuda and Matsui 1994 [47]) and an improvement of the previous algorithm given in (Uno 1997 [120]) and (Uno 2001 [119]), is the exchange of

matching edges with other edges along an alternating cycle or a feasible path. Enumerating all of the maximal matchings which include the source vertex (in our case *eq11*) is not interesting, since they will lead to equivalent debugging solutions. All the obtained maximal matchings will have *eq11* as a free node. Therefore we are interested in finding other maximal matchings which include the source vertex and leave other equation nodes free. One important property of the over-constrained bipartite graph is that it only contains alternating paths because it is constructed from a maximum matching and a supplementary free edge. We can easily obtain all the maximal matchings in the over-constrained graph by exchanging matching edges with non-matching edges along an alternating path that starts from a free node.

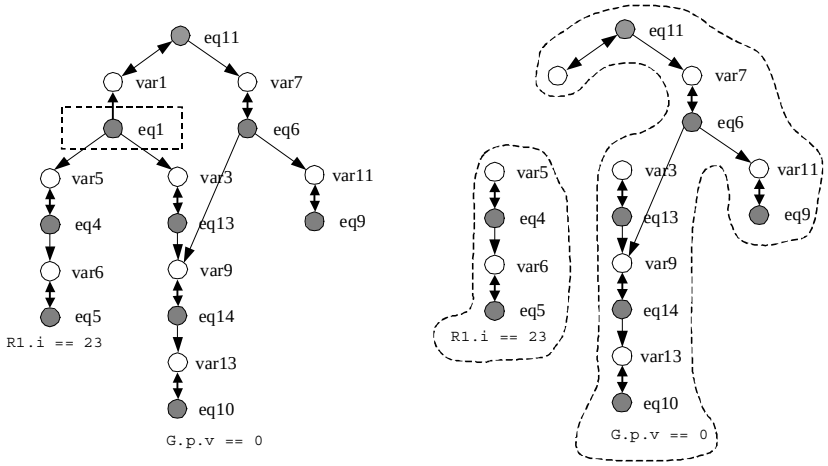



**Figure 5-5.** The elimination of an unsafe equation node from the over-constrained subgraph leads to two disconnected components.

The previously computed reduced equivalent list can further be refined taking into account simple rules derived from the language semantics. An algorithm for computing the reduced equivalent over-constraining equation list is given below:

---

**Algorithm 5-2: Annotation based equation set reduction**

**Input Data:** A reduced equation set *L* taken from the output of Algorithm 5-1.

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

**begin:**

- Eliminate from the list all of the equations that are generated by a `connect` equation and for which the *No. of associated eqs.* parameter exceeds the number of free nodes.
- Eliminate all the equations for which the *No. of associated eqs.* parameter is equal to 1. Add those equations to the history list.
- Sort the remaining equations in decreasing order of flexibility level
- Output the sorted list of equations $L_r$ .

**end**.

---

If the length of the reduced equivalent over-constraining list is equal to the number of free nodes of the over-constrained graph, automatic debugging of the model is possible by eliminating the equation from the simulation model without any supplementary user-intervention. Of course, the history list together with the elimination is output to the

user. If the length of the list is greater than the degree of freedom, this means that several error fixing strategies are possible and therefore user intervention is required for selecting the appropriate subset of equations that need to be removed. The reduced list is output to the user starting with the equation which has the highest flexibility level.

In our case the set of equivalent over-constraining equations is {*eq11, eq13, eq10, eq5, eq9*} with their associated annotations shown in Table 5-3. The equation node *eq11* was already analyzed and can therefore be eliminated from the set. Equation node *eq13* is eliminated as well, for the same reasons as equation *eq11*. Analyzing the remaining equations {*eq10, eq5, eq9*} one should note that they have the same flexibility level and therefore are candidates for elimination with equal probability. However, analyzing the value of the *No. of associated eqs.* parameter, equation *eq10* and *eq9* have this attribute equal to one, which means that they are singular equations defined inside the model. Eliminating one of these equations will invalidate the corresponding model, which is probably not the intention of the modeler.

Examining the annotations corresponding to equation *eq5* one can see that it can safely be eliminated because the flexibility level is high. Moreover, removing the equation will not invalidate the model since there is another equation defined inside the model. The directed bipartite graph obtained after the removal of eq5 is depicted in Figure 5-5. 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`), exactly indicating to the user which equation needs to be removed in order to make the simulation model mathematically sound. In this case the debugger correctly located the faulty equation previously introduced by us in the simulation model.

By examining the annotations corresponding to the set of equations which need to be eliminated, our implemented 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 prioritized 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 doesn't have sufficient information to decide which equation is over-constraining. The advantage of this approach is that the debugger automatically identifies and solves several anomalies in the declarative simulation model specification without having to execute the system.

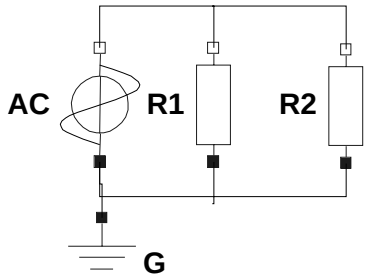
**Table 5-3.** The associated annotations of the equivalent over-constraining equation set

| Name | Equation | No. of assoc.eq. | Class name | Flex. level | Connector Generated |
|---|---|---|---|---|---|
| eq11 | `AC.p.v = R1.p.v` | 2 | `connect(AC.p,R1.p)` | 1 | Yes |
| eq13 | `R1.n.v = AC.n.v` | 2 | `connect(R1.n,C.n)` | 1 | Yes |
| eq10 | `G.pv = 0` | 1 | `Ground` | 2 | No |
| eq5 | `R1.i = 23` | 2 | `Resistor` | 2 | No |
| eq9 | `AC.v = AC*VA* sin[2*t*AC.f*AC.PI]` | 1 | `VsourceAC` | 2 | No |

## 5.7 Over-Constrained Subgraphs

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

We construct a simple electrical circuit model by connecting two resistors in parallel with a voltage source as shown in Figure 5-6. The Modelica definition of the `Ground`, `VsourceAC` and `Resistor` classes is reused from the previous example. The `TwoPin` class is modified by introducing an additional over-constraining equation (`i=10`) in the model definition. This extra equation will be inherited by all the classes, which extend the `TwoPin` class. Therefore each instance of the `Resistor` and `VsourceAC` models will contribute to one extra over-constraining equation to the final flattened system of equations.




```
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 5-6.** Electrical circuit with two resistors connected in parallel with an additional equation introduced in the `TwoPin` class.

During model translation the flattened set of equations corresponding to the simulation model is derived (shown in Table 5-4) and the associated bipartite graph $G$ is constructed. The flattened model corresponding to the simple electrical circuit now contains three extra over-constraining 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 over-constrained subgraph $O_G^{3+}$ is obtained, depicted in Figure 5-7. The maximum cardinality matching for the corresponding bipartite graph leaves three vertices uncovered corresponding to the equations *eq9*, *eq18* and *eq17*. The set of edges $\varepsilon(O_G^{3+})$ is given below:

$$\varepsilon(O_G^{3+}) = \{(eq9,var12),(\overline{var12,eq8}),(eq8,var8),(\overline{var8,eq19}),(eq19,var2),(\overline{var2,eq3}),$$
$$(eq19,var14),(\overline{var14,eq13}),(eq13,var18),(\overline{var18,eq14}),(eq18,var7),(\overline{var7,eq6}),$$
$$(eq6,var11),(\overline{var11,eq10}),(eq10,var12),(eq6,var9),(\overline{var9,eq21}),(eq21,var15),$$
$$(\overline{var15,eq22}),(eq22,var19),(\overline{var19,eq16}),(eq18,var1),(\overline{var1,eq1}),(eq1,var3),$$
$$(\overline{var3,eq20}),(eq20,var9),(eq1,var5),(\overline{var5,eq5}),(eq5,var6),(\overline{var6,eq4}),$$
$$(eq17,var13),(\overline{var13,eq11}),(eq11,var15),(eq11,var17),(\overline{var17,eq15})\}$$

**Table 5-4.** Flat form of the equations corresponding to the over-constrained electrical circuit model from Figure 5-6.

| | | | |
|---|---|---|---|
| ***eq1*** | `R1.v==-R1.n.v+R1.p.v` | ***var1*** | `R1.p.v` |
| ***eq2*** | `0==R1.n.i+R1.p.i` | ***var2*** | `R1.p.i` |
| ***eq3*** | `R1.i==R1.p.i` | ***var3*** | `R1.n.v` |
| ***eq4*** | `R1.i==10` | ***var4*** | `R1.n.i` |
| ***eq5*** | `R1.i R1.R==R1.v` | ***var5*** | `R1.v` |
| ***eq6*** | `R2.v==-R2.n.v+R2.p.v` | ***var6*** | `R1.i` |
| ***eq7*** | `0==R2.n.i+R2.p.i` | ***var7*** | `R2.p.v` |
| ***eq8*** | `R2.i==R2.p.i` | ***var8*** | `R2.p.i` |
| ***eq9*** | `R2.i==10` | ***var9*** | `R2.n.v` |
| ***eq10*** | `R2.i*R2.R==R2.v` | ***var10*** | `R2.n.i` |
| ***eq11*** | `AC.v==-AC.n.v+AC.p.v` | ***var11*** | `R2.v` |
| ***eq12*** | `0==AC.n.i+AC.p.i` | ***var12*** | `R2.i` |
| ***eq13*** | `AC.i==AC.p.i` | ***var13*** | `AC.p.v` |
| ***eq14*** | `AC.i==10` | ***var14*** | `AC.p.i` |
| ***eq15*** | `AC.v==AC.VA*sin[2*time*AC.f*AC.PI]` | ***var15*** | `AC.n.v` |
| ***eq16*** | `G.p.v==0` | ***var16*** | `AC.n.i` |
| ***eq17*** | `AC.p.v==R1.p.v` | ***var17*** | `AC.v` |
| ***eq18*** | `R1.p.v==R2.p.v` | ***var18*** | `AC.i` |
| ***eq19*** | `AC.p.i+R1.p.i+R2.p.i==0` | ***var19*** | `G.p.v` |
| ***eq20*** | `R1.n.v==R2.n.v` | ***var20*** | `G.p.i` |
| ***eq21*** | `R2.n.v==AC.n.v` | | |
| ***eq22*** | `AC.n.v==G.p.v` | | |
| ***eq23*** | `AC.n.i+G.p.i+R1.n.i+R2.n.i==0` | | |

Following the sets of the D&M canonical decomposition by starting from the free vertices, all of the descendants can be computed. Each free vertex induces an over-constrained subgraph. This over-constrained subgraph may contain cycles as well. In the example from Figure 5-7 the over-constrained subgraph is a tree and only contains alternating paths starting from the free equation nodes.

Three equations need to be eliminated from the over-constrained subgraph in order to make the system well-constrained. One equation needs to be eliminated from each over-constrained part $O_{1G}^{1+}, O_{2G}^{1+}, O_{2G}^{1+}$ (see Figure 5-9).

The following equations are included in the over-constraining sets:

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

$\{eq18,eq6,eq10,eq9,eq8,eq19,eq3,eq13,eq14,eq1,eq20,eq21,eq22,eq16,eq5,eq4\} \in v(O_{1G}^{1+})$

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

The set of safe over-constraining equations associated with each over-constrained subgraph can be computed using Algorithm 5-1. We obtain the following reduced set of equation nodes:

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

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

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

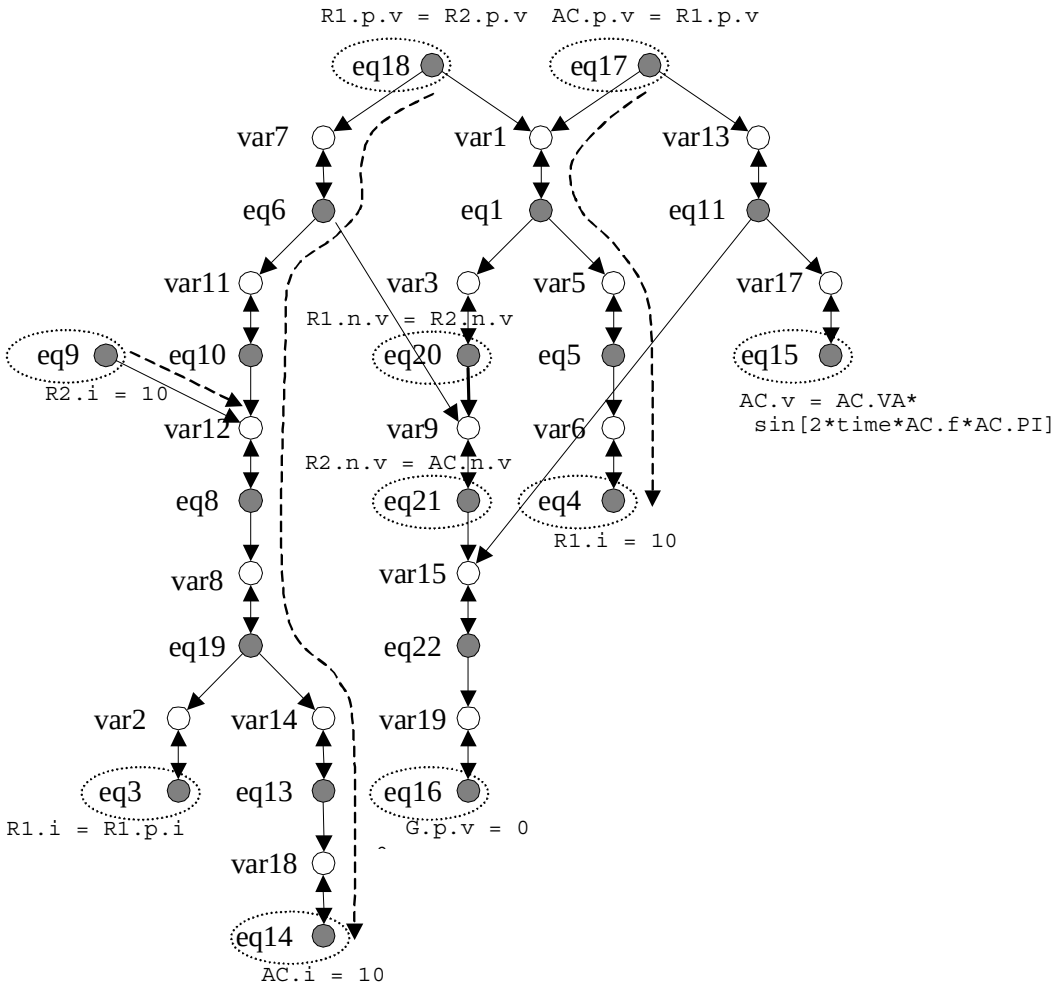



**Figure 5-7.** The over-constrained directed graph

All the possible combinations that can be scheduled for elimination are represented by the following graph where at the top we have the equations which are included in $O_{1G}^{1+}$, in the middle the equation nodes from $O_{2G}^{1+}$ and at the bottom equations from $O_{3G}^{1+}$. If three nodes (one from each subset) are linked together with edges, this means that it constitutes a valid set that can be considered for elimination.

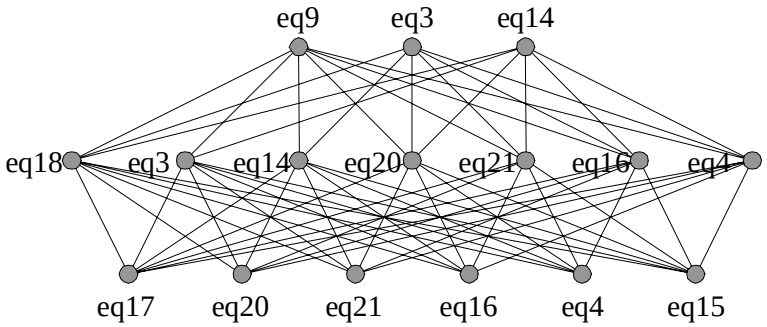



**Figure 5-8.** The possible elimination combinations.

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 to the subgraphs $O_{1G}^{1+}$ and $O_{2G}^{1+}$. This means that *eq3* can be made a free vertex by exchanging matching edges with non-matching edges along the path $eq9 \xrightarrow{=} eq3 \in O_{1G}^{1+}$ or $eq18 \xrightarrow{=} eq3 \in O_{2G}^{1+}$. If *eq3* is scheduled for elimination 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 to the subgraph. Moreover, if *eq13* 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 non-matching edges with matching edges along the path $eq9 \xrightarrow{=} eq3 \in O_{1G}^{1+}$ will affect the path $eq18 \xrightarrow{=} eq14 \in O_{2G}^{1+}$ isolating *eq14.* Therefore *eq14* cannot be selected any more for elimination from $O_{2G}^{3+}$ even if it previously had a valid path to the free node *eq18.*

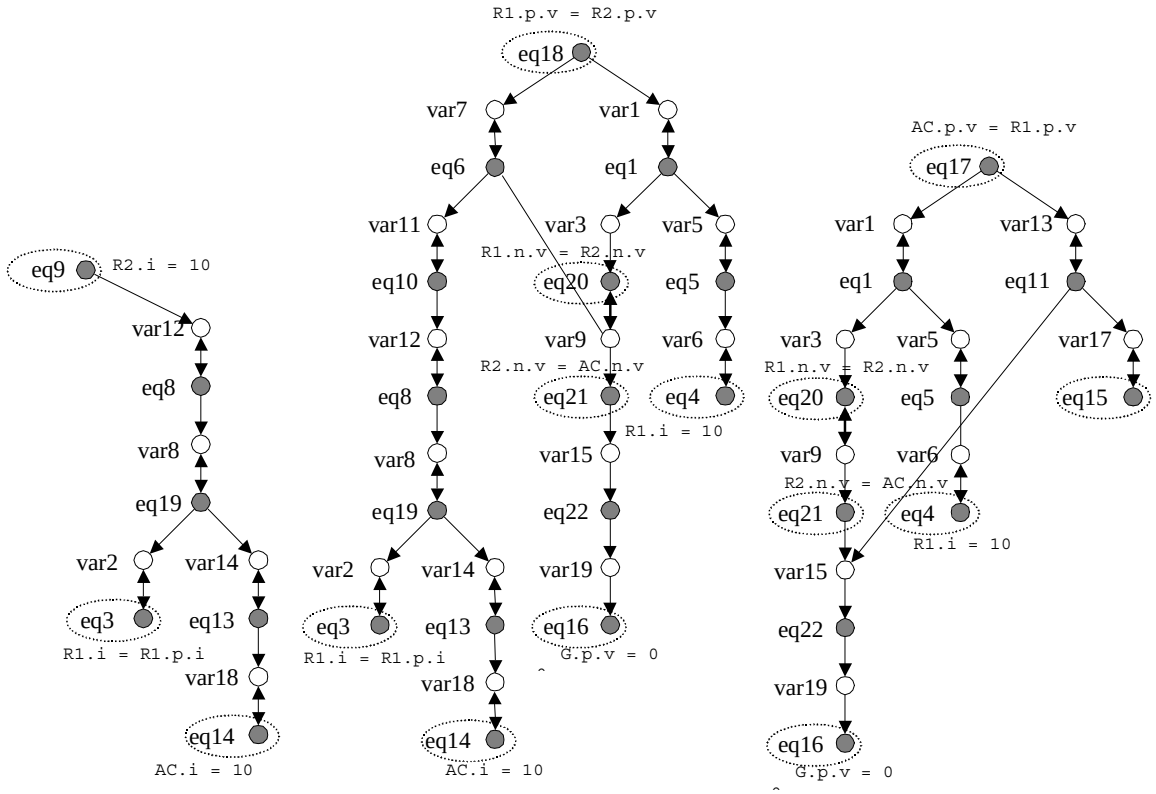



**Figure 5-9**. The $O_{1G}^{1+}, O_{2G}^{1+}, O_{2G}^{1+}$ components of the $O_G^{3+}$ over-constrained subgraph.

The previous example has demonstrated that a mechanism to quickly check if certain equation subsets can constitute a safe removal set is needed. The following section introduces a special graph structure which captures the dependencies among the equations. We also present an algorithm that verifies the validity of certain equation subsets.

## 5.8 Alternating Paths Dependency Graphs

In order to illustrate the path selection algorithm and to show the dependencies among the over-constraining variables, the set of safe over-constraining equations associated with each over-constrained subgraph is expanded to a set of paths where each equation element is replaced by the path from the free variable to itself. The safe over-constraining equation sets become:

$\{eq\,9 \xrightarrow{*} eq\,9, eq\,9 \xrightarrow{=} eq\,3, eq\,9 \xrightarrow{=} eq\,14\}$ for $O_{1G}^{1+}$

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

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

A cut variable node is the first shared variable node along two considered paths, e.g. as depicted in Figure 5-10. Each path can be extended by including the cut variable node in the path. For example, considering the paths $eq9 \xrightarrow{=} eq3 \in O_{1G}^{1+}$ and $eq18 \xrightarrow{=} eq14 \in O_{2G}^{1+}$ the first common variable node which is included in both paths is *var12*. Including the cut variable node the path $eq9 \xrightarrow{=} eq3$ becomes $eq9 \xrightarrow{=} \mathrm{var}12 \xrightarrow{=} eq3$. The list of all cut variable nodes {*var12, var1, var9, var5*} associated with $O_G^{1+}$ can be computed and used to expand the path sets as shown below, and depicted in Figure 5-10.

$\{eq9 \xrightarrow{*} eq9, eq9 \xrightarrow{=} \mathrm{var}12 \xrightarrow{=} eq3, eq9 \xrightarrow{=} \mathrm{var}12 \xrightarrow{=} eq14\}$

$\{eq18 \xrightarrow{*} eq18, eq18 \xrightarrow{=} \mathrm{var}12 \xrightarrow{=} eq3, eq18 \xrightarrow{=} \mathrm{var}12 \xrightarrow{=} eq14,$
$eq18 \xrightarrow{=} \mathrm{var}1 \xrightarrow{=} eq20, eq18 \xrightarrow{=} \mathrm{var}1 \xrightarrow{=} eq21, eq18 \xrightarrow{=} \mathrm{var}1 \xrightarrow{=} eq16,$
$eq18 \xrightarrow{=} \mathrm{var}1 \xrightarrow{=} eq4, eq18 \xrightarrow{=} \mathrm{var}9 \xrightarrow{=} eq21, eq18 \xrightarrow{=} \mathrm{var}9 \xrightarrow{=} eq16\}$
$\{eq17 \xrightarrow{*} eq17, eq17 \xrightarrow{=} \mathrm{var}1 \xrightarrow{=} eq20, eq17 \xrightarrow{=} \mathrm{var}1 \xrightarrow{=} eq21,$
$eq17 \xrightarrow{=} \mathrm{var}1 \xrightarrow{=} eq16, eq17 \xrightarrow{=} \mathrm{var}1 \xrightarrow{=} eq4, eq17 \xrightarrow{=} \mathrm{var}5 \xrightarrow{=} eq16,$
$eq17 \xrightarrow{=} \mathrm{var}5 \xrightarrow{=} eq4, eq17 \xrightarrow{=} eq15\}$

An edge starting from a free equation and pointing back to itself denotes a free variable. For generality, it can be considered as an alternating path with the number of edges equal to zero. If a free equation is chosen for elimination, then only the associated path is eliminated as well as the incident edges.

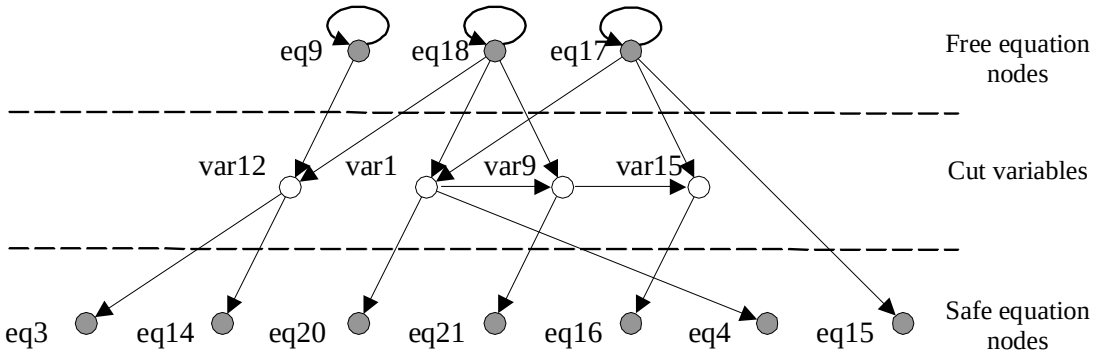



**Figure 5-10.** Shortened representation of the alternating paths.

Any further computation involving the equation node elimination can now be performed on the shortened representation of the alternating paths. Let us illustrate the reasoning performed on the above mentioned graph representation by choosing equation node *eq3* for elimination from the set of safe nodes included in $O_{1G}^{1+}$. In order to safely eliminate *eq3,* the matching edges need to be exchanged with non-matching edges along the path $eq9 \xrightarrow{=} eq3 \in O_{1G}^{1+}$. By performing exchange of edges all the paths that have common edges with the modified path are affected making the safe nodes unreachable from the free equation node. Therefore all the edges incident to *eq3* can be eliminated. Then we follow the chosen path backwards including the cut variable *var12.*

All the associated edges incident with the node *var12* will be eliminated as well from the path graph. We go backwards on the chosen path one more time and reach the free equation node. All the incident edges with the free equation node are also removed. An edge between the free equation node and the chosen safe equation node can be drawn indicating that equation node *eq3* has been made a free node instead of equation node *eq9* as shown in Figure 5-11 a). It should be noted that the safe equation node *eq14* becomes isolated and cannot reach a free node any more.

In the next step another equation node need to be selected from the safe equation sets associated with the second over-constrained subgraph $O_{2G}^{1+}$. Equation node *eq13* has already been selected for elimination in subgraph $O_{1G}^{1+}$ and *eq14* is not reachable anymore. An equation from the set $\{eq18, eq9, eq14, eq20, eq21, eq16\} \in v(O_{1G}^{1+})$ needs to be chosen for elimination. Let us choose *eq21* along the path $eq18 \xrightarrow{=} var9 \xrightarrow{=} eq21$. All the edges incident with nodes *eq21, var9,* and *eq18* are removed from the graph together with the cut variable node *var9.*

Moving to the next over-constrained subgraph $O_{3G}^{1+}$ we might choose *eq4* for elimination, which is available along the path $eq17 \xrightarrow{=} \mathrm{var}1 \xrightarrow{=} eq4$ as depicted in Figure 5-11 b). By following the alternating path backwards and eliminating the extra edges and the cut variable node, we finally obtain the graph shown in Figure 5-11 c).

It should be noted that the three safe equation nodes *eq3, eq21,* and *eq4* have been arbitrarily chosen just for illustrating the functionality of the path graph reduction algo-

rithm. All the possible combinations of three equations need to be taken into account by the debugger when generating error-fixing hypotheses.

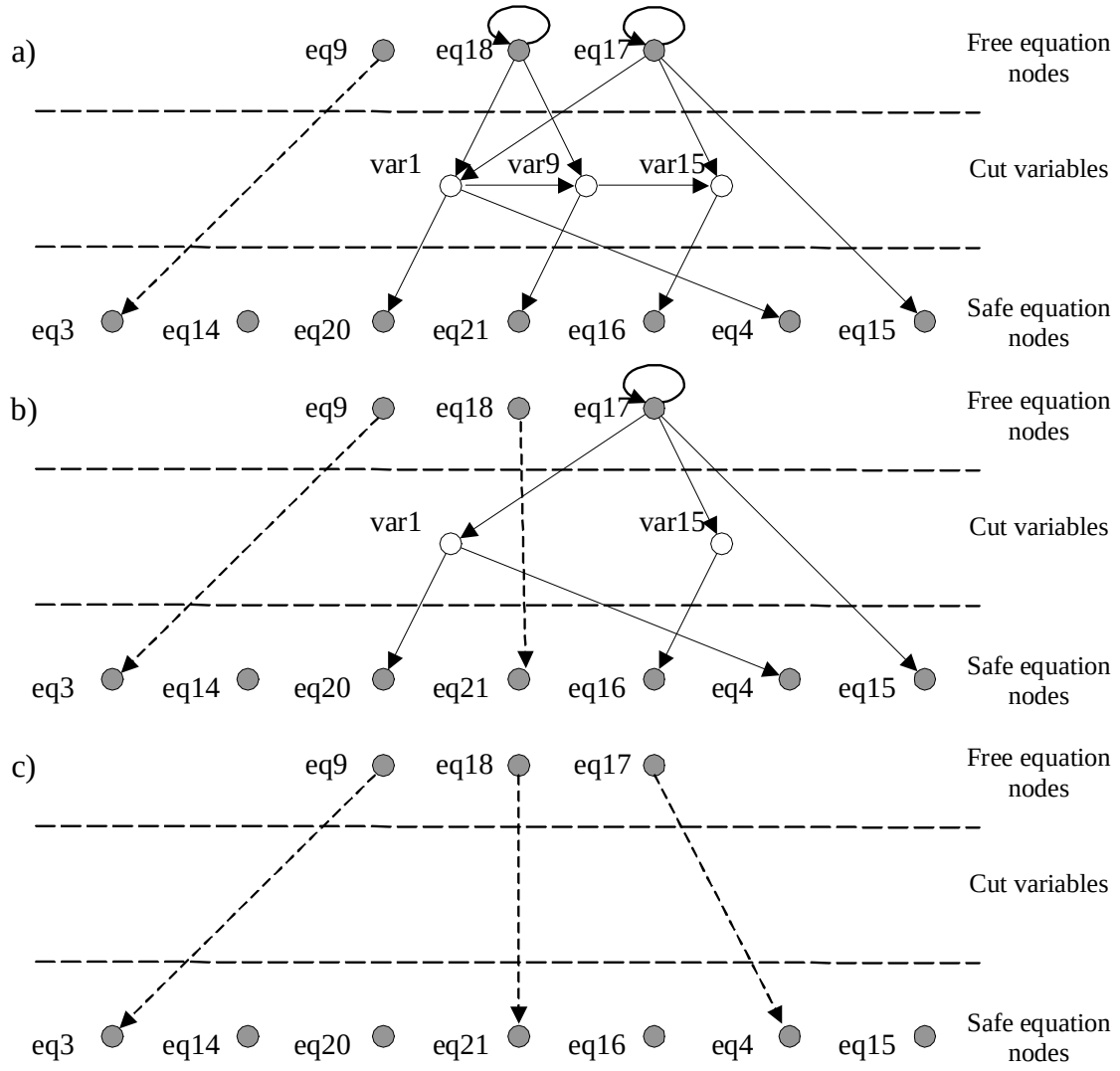



**Figure 5-11.** Transformation of the shortened alternating path graph by choosing **a)** *eq3* **b)** *eq3* and *eq21* **c)** *eq3, eq21* and *eq4* for elimination.

The shortened alternating path graph is useful for quickly checking if a given subset of safe equations chosen by the user or by the debugger, if eliminated from the overall system of equations lead to a remaining well-constrained system of equations. In conclusion the following recursive algorithm Algorithm 5-3 is given for automatically checking the validity of an equation subset. The number of equations in the subset which is checked by the algorithm must be equal to the number of free nodes in the over-constrained subgraph. Table 5-5 describes functions called from Algorithm 5-3.

**Algorithm 5-3:** CHKOC( $SO_G^{k+}$ , $\{eq_1, eq_2 \ldots eq_k\}$ ) **Checks if a subset of equation constitutes a valid elimination set to remove from the flattened set of equations**

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

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

```
begin:
    Procedure chkoc(G, eqList)
    begin
        if (size(eqList) != 0) then
            eq_k = pop_last(eqList);
            pathList := DFSm(G, eq_k );
            optimize(pathList);
            for all p' ∈ pathList do
                for all nodes v ∈ p' do
                    hide_node_ and_inc_ed ges(G, v) ;
                end for;
                if eq_k ⊄ L then L := append(L, { eq_k });
                chkoc (G,eqList);
                restore_no des_and_in c_edges (G, v) ;
            end for;
            push_back(eqList ,eqk);
        endif;
     end procedure.
    L = ∅ ;
    chkoc( SO_G^{k+} , {eq_1, eq_2 ... eq_k} )
    if (L == {eq_1, eq_2 ... eq_k} ) then return true;
    else return false;
end.
```

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

In Algorithm 5-3 the following functions and conventions have been used:

**Table 5-5.** Functions called from CHKOC

| Function | Convention description |
|---|---|
| pop_last(*list*) | Returns the last element from the list and modifies the lis by deleting the popped element. |
| push_back(*list ,elem*) | Adds an element *elem* at the end of the list *list.* |
| hide_node_and_inc_ edges(*G,v*) | Hides the node *v* and all the incident edges from the graph *G*. |
| restore_node_and_inc_ edges(*G,v*) | Restores the hidden node *v* and all the incident edges from the graph *G.* |
| append(*list, elemt*) | Appends an element *elem* to a list *list.* |
| size(*list*) | Returns the size of the list *list*. |
| optimize(*list*) | Optimizes the path list by eliminating all those paths that do not terminate with a safe edge. |
| delete(*node*) | Deletes the node and all the incident edges from the graph. |
| DFSm(*G,node*) | The modified version of the Depth First Search algorithm |

By checking each possible combination with Algorithm 5-3 the graph from Figure 5-8 can be reduced. The resulting reduced graph is shown in Figure 5-12, which still has a high complexity.

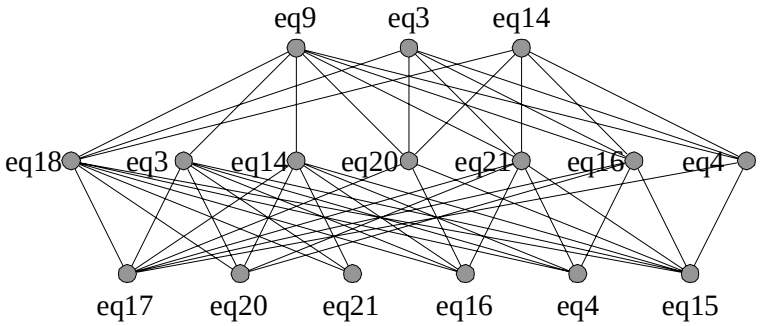



**Figure 5-12.** Reduced safe nodes combinations graph.

## 5.9 Filtering with Constraint Rules Based on Language Semantics

Applying Algorithm 5-1 and taking into account the structural information regarding the over-constrained subgraph, the number of possible combinations of safe nodes have been reduced. There are still too many combinations of safe equations. Presenting them to the user at this stage is not very useful. These combinations represent the error fixing solutions at the flattened intermediate code level. However, the user has only the possibility of making modifications at the source code level. Many of the possible combinations are impossible to get just by applying atomic changes to the original source code. Therefore a filtering mechanism to remove invalid solutions from the modeling language point of view is needed.

Let us consider the mapping between the original Modelica source code and the generated intermediate form of the flattened equations shown in Figure 5-13. In Figure 5-13 only the relevant statements from the original Modelica code that directly generate the intermediate form are shown. Model definitions, variable declarations and other language constructs for our purposes irrelevant have intentionally been eliminated. Only the equation including connect equations have been kept.

Until now the over-constraining detection algorithm only works on the intermediate form of the equations. However, only those fixing strategies can be taken into account that can operate on the original language constructs.

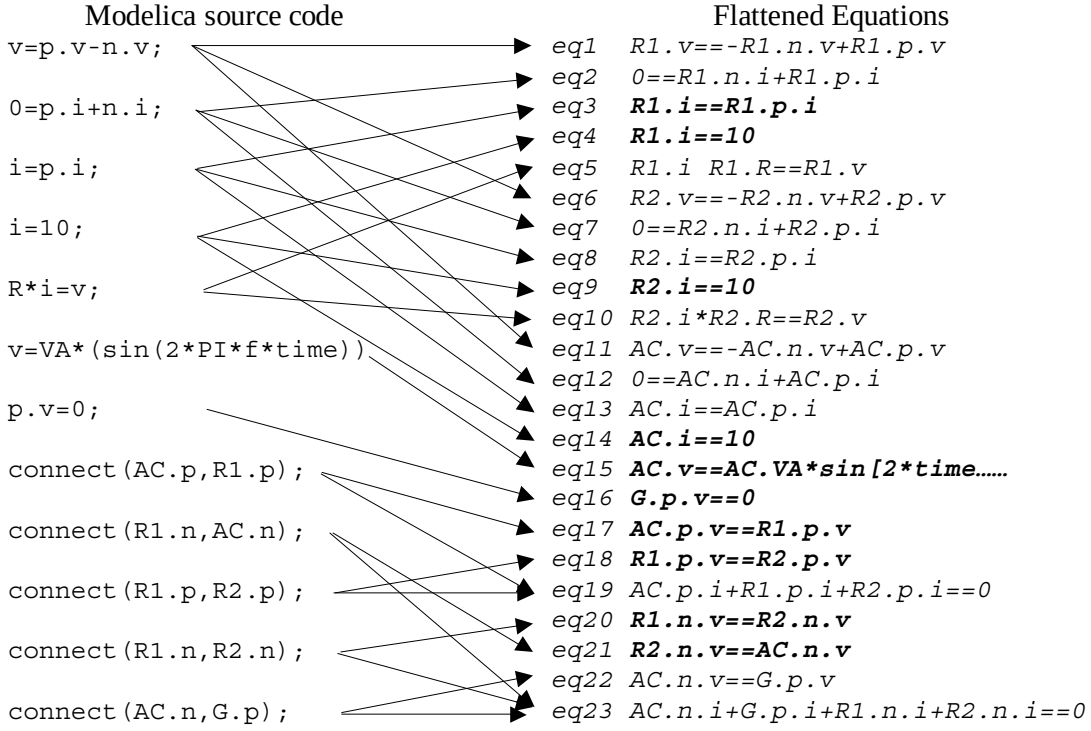



**Figure 5-13.** Correspondence between the Modelica source code statements and the corresponding generated set of flattened equations. Safe equations are in bold font.

The safe equations which are also present in the shortened alternating path are emphasized in Figure 5-13 by bold letters. As was mentioned earlier, three equations from the set of safe equation nodes need to be eliminated. Therefore any further analysis can be performed on this set and on the associated shortened alternating path. The correspondence graph from Figure 5-13 can be simplified only taking into account the set of safe equation nodes and the corresponding source code statements which generated them through program transformations. Those original code statements that generate intermediate equations that are outside the set of safe equations can be discarded from the graph. The simplified form of the correspondence graph is given below:

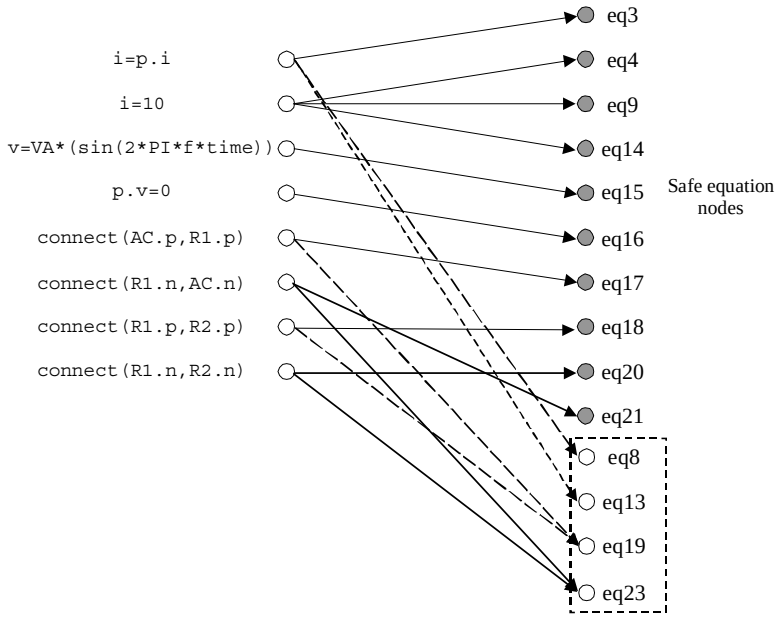



**Figure 5-14.** Reduced correspondence graph between the original source code and the corresponding generated set of flattened equations.

As can be seen in Figure 5-14 we have only kept those statements in the original source code that generate at least one equation in the reduced set of safe equations. Only these statements can be manipulated by the user with the help of atomic changes performed on the original source code.

Let us now analyze the equations: *eq4, eq9,* and *eq14*. They were obtained by inheritance from the original equation `i=10` in the `TwoPin` class. By eliminating the original source code statement, using the atomic change command, DELEQ(`i=0,TwoPin`) all the three equations will be removed from the flattened intermediate form. Any attempt to eliminate only one of the equations from the intermediate form by removing a statement from the original source code will not succeed. In conclusion, all the incident edges with the nodes that represent *eq4, eq9,* or *eq14* in the graph from Figure 5-12 that includes a node which is not among them, can safely be removed.

Inheriting the equation `i=p.i` will generate three equations *eq3, eq8* and *eq13* in the intermediate code. Only *eq13* is among the equation nodes that need to be eliminated. It should be noted that the elimination of *eq3* is only possible by removing `i=p.i` DELEQ(`i=p.i,TwoPin`). However, the removal of `i=p.i` will trigger the elimination of two additional equations which are not members of the safe equation set. Therefore *eq3* cannot be considered for elimination and all the incident edges with *eq3* in the graph of Figure 5-12 can be removed. For the same reason all the incident edges with *eq17, eq18, eq20, eq21* can also be removed from the graph that represents the valid combinations.

After performing all the simplifications we obtain the graph of Figure 5-15 that only contains edges linking *eq4, eq9,* and *eq14* equation nodes. This means that the user can eliminate the statement `i=10` from the `TwoPin` component in the original source code. This statement was exactly the additional statement introduced at the beginning of this analysis in order to over-constrain the simulation model.

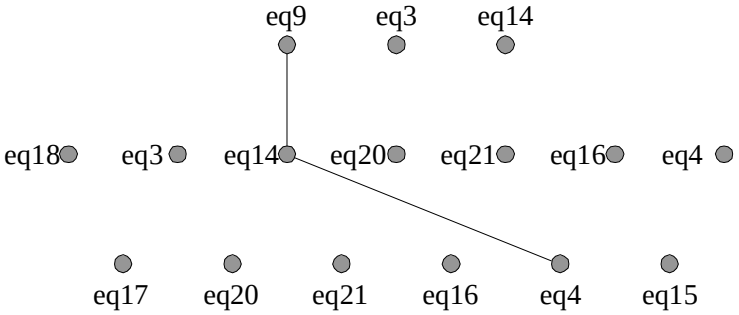



**Figure 5-15.** The simplified graph denoting the possible equation node combinations that can be scheduled for elimination.

By eliminating *eq9*, *eq14* and *eq4*, the over-constrained graph from Figure 5-7 becomes a well-constrained graph as illustrated below:

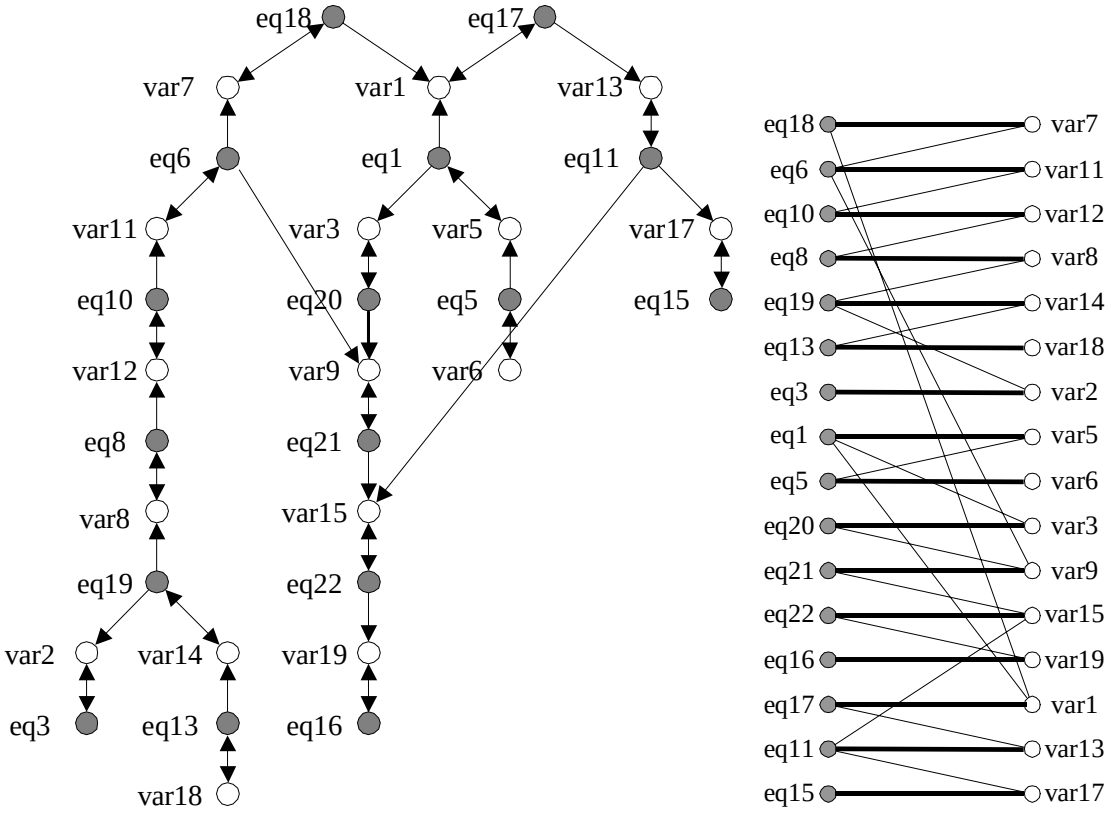



**Figure 5-16.** Well-constrained graph obtained after elimination of three over-constraining equations.

The same over-constraining final effect, and the same form of the flattened intermediate code can be achieved by over-constraining each class derived from `TwoPin` instead of over-constraining the parent class itself with an extra equation. In conclusion the classes `Resistor`, `VsourceAC` will get an extra equation `i=10` each. The generated flattened form of the equations will be the same and therefore only the last part of the error fixing algorithm is responsible for detecting and providing the user the right solution. The graph from Figure 5-14 will be changed and is presented in Figure 5-15.

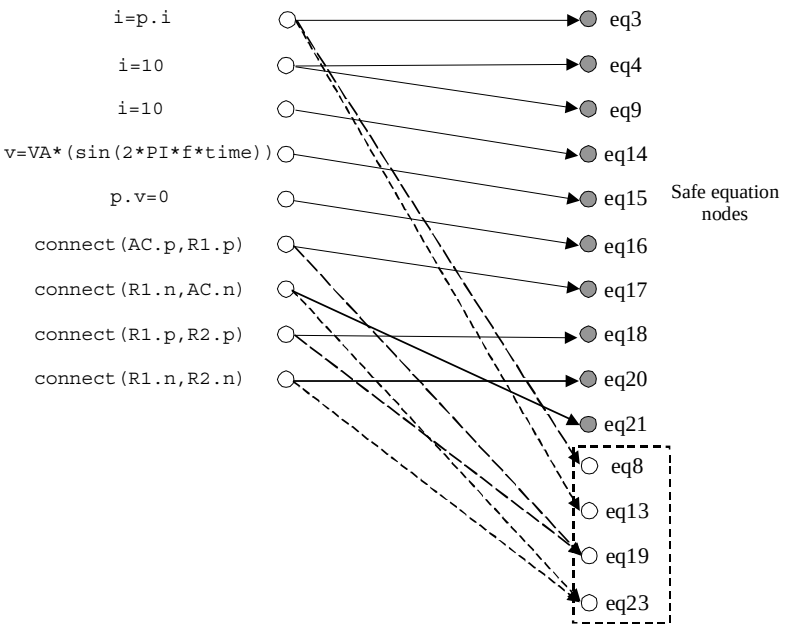



**Figure 5-17.** Reduced correspondence graph when the `Resistor` and `VsourceAC` component are over-constrained.

Reasoning based on the semantic language constraints, we obtain the following graph that represents the valid combinations, depicted in Figure 5-18. However there is an additional constraint that *eq9* and *eq4* should be selected together (they cannot appear independently in the selection set).

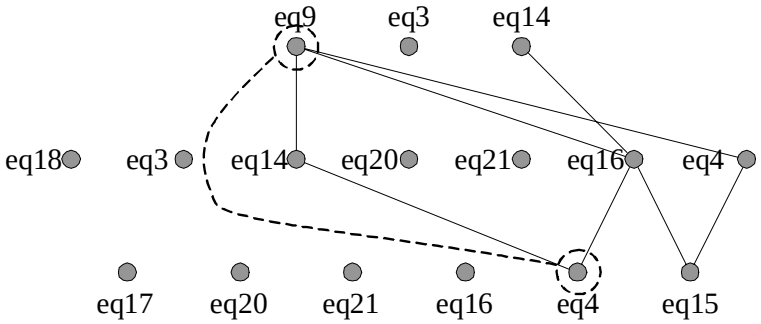



**Figure 5-18.**Possible equation combinations scheduled for elimination.

The following equation sets can be selected for elimination:

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

By using the equation annotations from section 5.3 and reasoning based on the attached annotations the option set can be further reduced. It can easily be seen that equation *eq16* (`p.v = 0`) from the `Ground` component cannot be eliminated because it is the only equation that defines the behavior of this component. In conclusion, after performing all the filtering algorithms including the filtering based on the equation annotations, for this particular example the debugger will present the following message to the user as depicted in Figure 5-19.

```
ERROR!!!
Over-constrained situation detected in model Circuit.

No. of equations = 23
No. of variables = 20

General debugging options: Level 1
Secondary debugging options: Level 1

equation elimination

solution 1:
  model Resistor
    remove equation i = 10
  model VSourceAC
    remove equation i = 10

solution 2:
  model Resistor
    remove equation i = 10
  model VSourceAC
    remove equation v = VA * sin(2 * PI * f * time)

annotation based filtering of solutions.

solution 3:
  model Resistor
    remove equation i = 10
  model Ground
    remove equation p.v = 0

solution 4:
  model Resistor
    remove equation i = 10
  model VSourceAC
    remove equation i = 10
    remove equation v = VA * sin(2 * PI * f * time)
```

**Figure 5-19.** Debugger output for the simple `Circuit` model when the `Resistor` and `VsourceAC` classes are over constrained.

The debugger operates by using general level and secondary level settings that are explained in Chapter 8 section 8.3.6. In this case an annotation based filtering was also used that makes solutions 3 and 4 less probable. By implementing solution 3 the only equation that defines the behavior of the `Ground` component is removed. In a similar manner, the implementation of solution 4 will remove all the equations that define the behavior of the `VsourceAC` component.

## 5.10 Conclusions

The case studies presented in this chapter indicate the spectrum of potential applications of the developed algorithms for debugging and structural analysis of simulation models. While the case studies demonstrate the applicability of the bipartite graph decomposition and over-constrained subgraph properties in detecting over-constraining equations, they also reveal some shortcomings and troublesome aspects of the simulation models specified in object-oriented equation-based modeling languages.

For example, when an over-constraining equation is present in a partial class, this will lead to a number of over-constraining equations in the flattened set of equations equal to the number of the instantiated models present in the final simulation model which inherits that partial class. Therefore the number of free nodes in the over-constrained subgraph will be equal to the number of instances corresponding to those components which extend the over-constrained faulty parent class. Our debugging approach first detects all the over-constraining equations in the instantiated models, and then, based on the information provided by the equation annotations and on the correspondence graph between the original source code and the intermediate code, detects the over-constraining equation that originates the failure. Of course, detecting the over-constraining equations in the instantiated model is computationally expensive and in the worst case implies a traversal of almost all equations in the flattened set of equations.

# Chapter 6

# Debugging of Under-Constrained Systems of Equations

***Summary****: Debugging of under-constrained systems of equations is much more difficult than the debugging of over-constrained systems mostly due to the combinatorial explosion in error fixing solutions at the intermediate code level. When debugging such systems two distinct strategies can be considered. The first strategy considers the removal of the free variables while the second strategy considers the addition of new equations to the overall system of equations, which must contain the free variables. Additionally, the second strategy takes into account extra variables that can be added to the introduced new equation. New equations can be introduced at different levels in the object hierarchy. The multitude of error fixing solutions requires a lot of user interaction and makes an associated debugger sometimes very tedious to use. Another problem is the complex interaction between the under- and over-constrained subsystems when they appear simultaneously. In this chapter, methods for detecting under-constrained situations and several algorithmic techniques to automate the error fixing process are proposed.*

## 6.1 Overview of the Under-Constrained Problem

Let us consider the number of equations *m* from a model and the number of variables *n* incident in those equations. For a typical under-constrained situation the number of variables is greater than the number of equations *(n>m).*

**Definition 6-1**: The *degree of under-constraining* is the difference between the number of variables and the number of equations $D_u = n - m$. In a similar way as in (Ramirez 1998 [98]) $D_u$ is called the number of *degrees of freedom* of the problem.

In the following sections we will illustrate the possible error fixing solutions for a typical under-constrained situation and the reasoning involved in the graph transformation system. Let us consider the following system of equations presented on Figure 6-1 on the right, with the degree of under-constraining $D_u = 1$. The corresponding bipartite graph to the system of equations is also given in Figure 6-1 on the left.

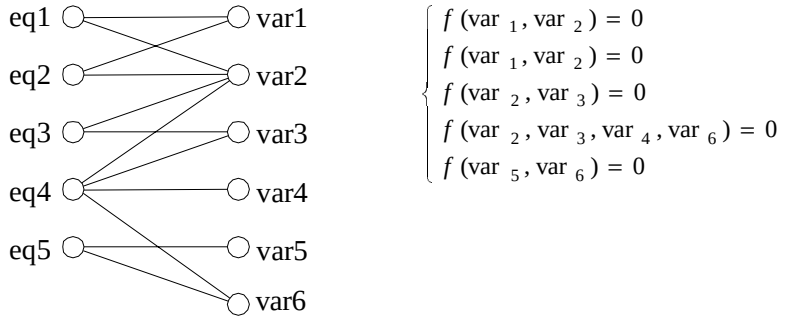



**Figure 6-1**. A simple system of equations with the associated bipartite graph.

One possible maximal matching (represented by thicker edges) of the bipartite graph the D&M canonical decomposition is given in Figure 6-2:

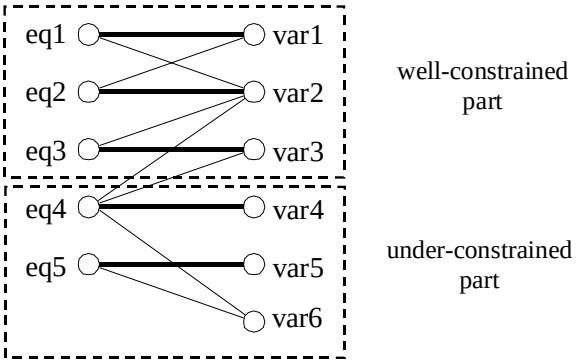



**Figure 6-2**. Maximum matching and canonical decomposition of the bipartite graph.

The first step when performing the canonical decomposition algorithm is to transform the undirected bipartite graph $G$ into a directed graph $\overline{G}$ by exchanging all the edges which are part of the maximal matching for bi-directional edges, and by orienting all other edges from equations to variable nodes. The corresponding directed graph $\overline{G}$ is shown in Figure 6-3.

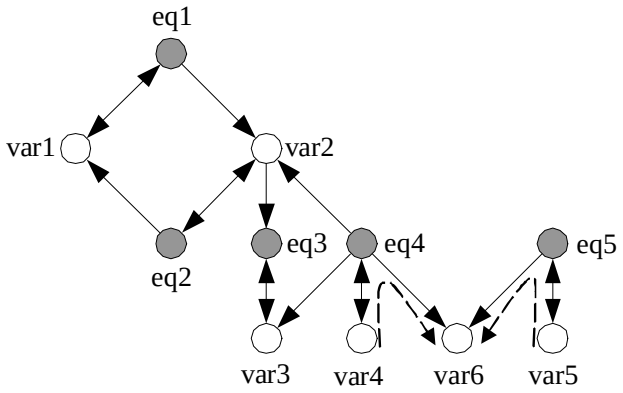



**Figure 6-3.** Directed graph associated with the system of equations.

Based on the D&M canonical decomposition algorithm the under-constrained part $U_G^{-1}$ contains all the equation and variable nodes that sink into the free variable node. The variables contained in an under-constrained 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.

The issue of under-constrained simulation models in object-oriented declarative equation-based frameworks is discussed in (Ramirez 1998 [98]). The work presented in (Ramirez 1998 [98]) is particularly concerned with issues involving modeling and solutions of conditional models where the system of equations in the model is different for each conditional alternative. In (Ramirez 1998 [98]) the eligibility set is constructed by following the Steward paths in the incidence matrix (see Figure 6-4) associated to a system of equations.

| | var1 | var2 | var3 | var4 | var5 | var6 |
|---|---|---|---|---|---|---|
| eq1 | X | X | | | | |
| eq2 | X | X | | | | |
| eq3 | | X | X | | | |
| eq4 | | X | X | X | | X |
| eq5 | | | | | X | X |

**Figure 6-4.** Incidence matrix corresponding to the system of equations from Figure 6-1.

The method of computing the Steward paths is similar to the construction of the under-constrained subsystem of the D&M canonical decomposition method. First one variable is assigned to each equation, which is similar to finding a matching in the corresponding bipartite graph. Such an assignment is indicated in Figure 6-5 by a circle drawn around the symbol that indicates the incidence of the variable in the equation. A Steward path starts from a free variable and then moves horizontally in the incidence matrix to an assigned variable, marking as eligible each variable encountered along each path as is illustrated by the arrows starting from the free variable *var6* in Figure 6-5.

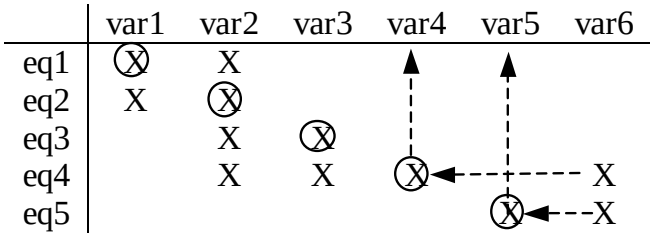



**Figure 6-5.** The eligibility set computation following the Steward paths.

Our analysis of the under-constrained systems is performed on the bipartite graph associated with the system of equations instead of on the incidence matrix as in (Ramirez 1998 [98]). As seen in Figure 6-2, variable *var*6 is not covered by the maximal matching and therefore is a free vertex. In the directed graph $\overline{G}$, 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 6-3):

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

Exchanging the matching edges with non-matching edges and the non-matching edges with matching edges along an alternating path, a new matching can be obtained which covers the free vertex *var6* but will uncover another vertex from the eligibility set. Therefore, an error fixing strategy must take into account 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 which involve the elimination of a variable from the eligibility set are taken into account. We have the following possible solutions illustrated in Figure 6-6.

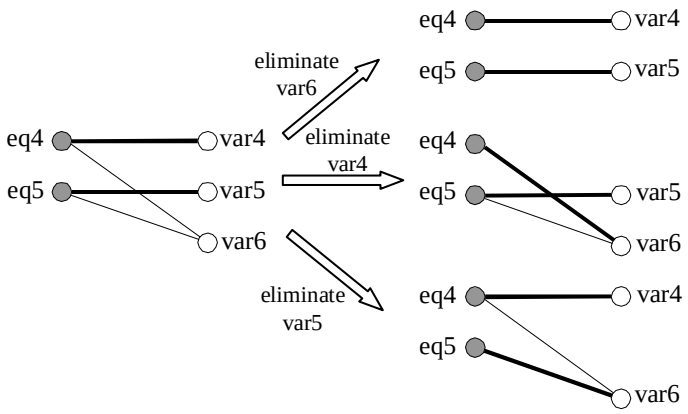



**Figure 6-6.** Error fixing solutions when one variable is eliminated from the eligibility set.

By removing *var6* from the under-constrained subsystem $U_G^{-1}$ the considered maximum matching becomes a perfect matching of the remaining bipartite graph. Therefore the associated system of equations can be considered to be structurally sound. However, by removing *var6* the resulted bipartite graph will be disconnected and an independent edge (*eq5*,*var5*) appears in the system, which is not connected to the main bipartite graph. This situation is unusual in physical system modeling and it means that some variables are computed locally, inside a component, without contributing to the general behavior of the simulated system. As an example, the following Modelica `Resistor` model integrated in a circuit model will produce two disconnected subgraphs.

```
model Resistor
   extends TwoPin;
   parameter Real R;
   Real s;
equation
   R*i=v;
   s=10;
end Resistor
```

The variable `s` and the equation `s=10` are redundant in the system. Therefore the situation when an extra variable is eliminated and the remaining bipartite graph is discon-

nected needs to be analyzed further. In our particular case from Figure 6-1, for example, a solution that involves the elimination of variable *var6* and the presence of an extra variable *var1*, *var2*, *var3* or *var4* in equation *eq5* might be acceptable.

It should also be noted that multiple error fixing strategies are possible in the case of an under-constrained subsystem. Another error fixing strategy for an under-constrained 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 6-7 where an extra equation *eq6* is added to the overall system of equations.

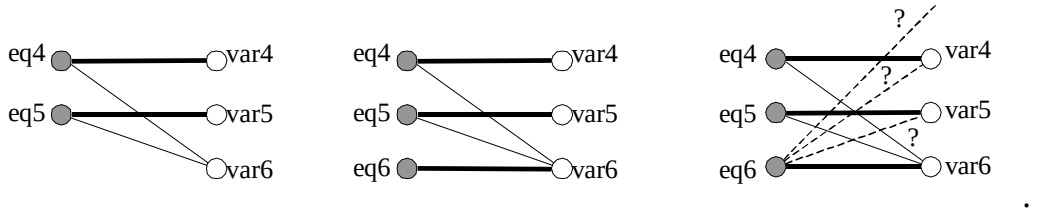



**Figure 6-7.** Error fixing strategy involving adding an extra equation.

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 in order to see if other variables from the system might be present in the added equation. This last step turns out to be very useful, because it helps the users to reconstruct missing equations from simulation models.

When modeling with object-oriented equation-based languages extra equations can be added at different levels in the class hierarchy, complicating the debugging process. The user must be able to select from a number of possible classes where the extra equation can be added. A detailed example of how this process is automatically performed by the debugger will be given in the following sections.

Under-constrained situations appear mostly when an equation has been accidentally deleted from the simulation model. Based only on structural information, the general form of the eliminated equation needs to be reconstructed. In such a case, just eliminating the free variables from the eligibility set is obviously not a valid error fixing solution. The debugger must provide the location of the extra equation and the possible variables that can be included in that equation. The user decides on a subset of variables that can be used in the equation and will also provide the right form of the equation.

To summarize, we consider two main strategies when debugging under-constrained equation systems:

- Removal of the free variable nodes
- Addition of new equation nodes to the overall system of equations and inserting the free variable into the created equations.

## 6.2 Simple Under-Constrained Circuit Example

Let us again analyze the simple electrical circuit model from Chapter 3, section 3.5.1, where the `Resistor` component is changed again by declaring an extra variable `(Real s)` and introducing this variable into the `Resistor` model equations as shown below:

```
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 and one possible corresponding maximum cardinality matching, is given in Figure 6-8. The correspondence between the variable and equation node labels is given in Table 6-1.

**Table 6-1.** Flat form of the equations corresponding to the under-constrained electrical circuit model with a modified resistor.

| | | | |
|---|---|---|---|
| ***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.s * 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*** | `R.s` |
| ***eq8*** | `AC.v = AC.VA * sin[2*time*AC.f*AC.PI]` | ***var8*** | `AC.p.v` |
| ***eq9*** | `G.p.v = 0` | ***var9*** | `AC.p.i` |
| ***eq10*** | `AC.p.v = R.p.v` | ***var10*** | `AC.n.v` |
| ***eq11*** | `AC.p.i + R.p.i = 0` | ***var11*** | `AC.n.i` |
| ***eq12*** | `R.n.v = AC.n.v` | ***var12*** | `AC.v` |
| ***eq13*** | `AC.n.v = G.p.v` | ***var13*** | `AC.i` |
| ***eq14*** | `AC.n.i + G.p.i + R.n.i = 0` | ***var14*** | `G.p.v` |
| | | ***var15*** | `G.p.i` |

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 under-constrained directed subgraph we can derive the following alternating paths (indicated in Figure 6-8 by the dashed arrows) to the uncovered variable *var15*:

$$\{(var15,eq14),\overline{(eq14,var4)},(var4,eq2),\overline{(eq2,var2)}(var2,eq3),\overline{(eq3,var6)},(var6,eq4),\overline{(eq4,var7)}\}$$

$$\{(var15,eq14),\overline{(eq14,var4)},(var4,eq2),\overline{(eq2,var2)}(var2,eq11),\overline{(eq11,var9)},(var9,eq6),\overline{(eq6,var11)}\}$$

$$\{(var15,eq14),\overline{(eq14,var4)},(var4,eq2),\overline{(eq2,var2)}(var2,eq11),\overline{(eq11,var9)},(var9,eq7),\overline{(eq7,var13)}\}$$

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*} corresponding to {`G.p.i, R.n.i, R.s, AC.i`} will not disconnect the bipartite graph. Therefore this reduced set will be further analyzed.

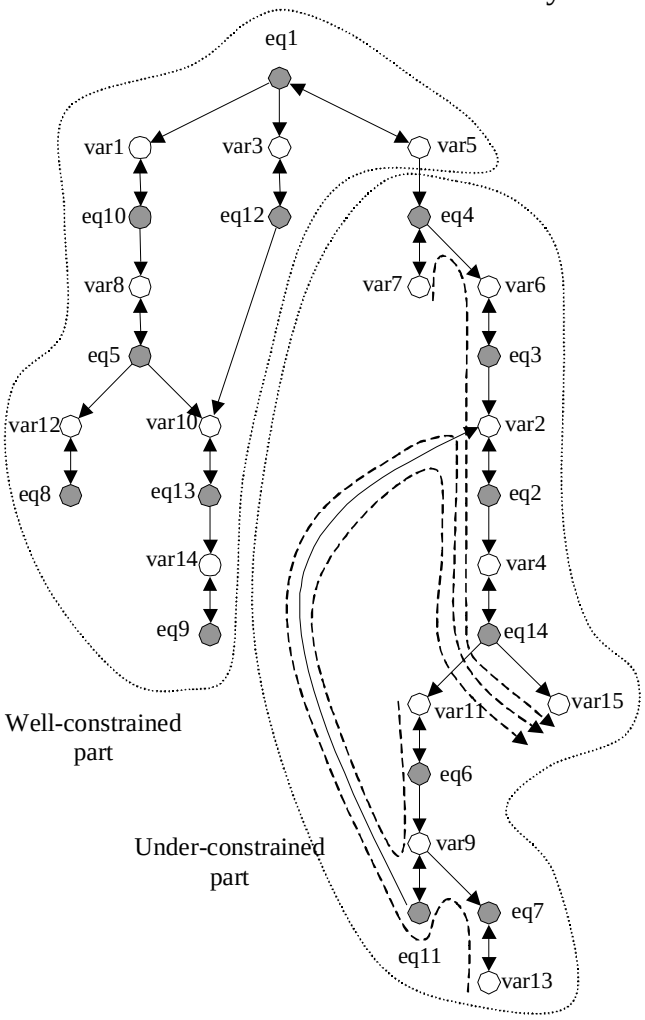



**Figure 6-8.** Directed graph corresponding to the under-constrained simple electrical circuit.

Based on a similar reasoning as for over-constrained situations, we can deduce that only *var7* can safely be removed from the Modelica code in order to obtain a well-specified equation system. We will briefly explain why the removal of *var15*, *var4*, and *var13* have not been considered as possible error fixing solutions to our under-constrained problem:

- Variable *var15* i.e. `G.p.i` In order 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 connect equation `connect(AC.n, G.p)` which connects the `Ground` element to the main circuit needs to be removed. This implies the removal of equation *eq10*(`AC.p.v = R.p.v`), an operation that will over-constrain the overall system.
- Variable *var4* i.e. `R.n.i`. The elimination of variable *var4* i.e. `R.n.i` from equation *eq14* (`AC.n.i + G.p.i + R.n.i = 0`) and *eq2* (`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 *eq2* from the flattened system of equations, making the system over-constrained.

- Variable *var13* i.e. `AC.i`. Removing *var13* i.e. `AC.i` from equation *eq7* (`AC.i = AC.p.i`) is possible at the source code level by removing the variable `i` from the equation `i = p.i` by substituting `i` with a constant value, from the `TwoPin` component. This modification will trigger at the intermediate code level the removal of *var6* i.e. `R.i` from *eq4* (`R.i * R.R = R.s * R.v`) that becomes `R.R = R.s * R.v` and from *eq3* (`R.i = R.p.i`) that becomes `const = R.p.i`. These modifications will disconnect the resulting bipartite graph.

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.*

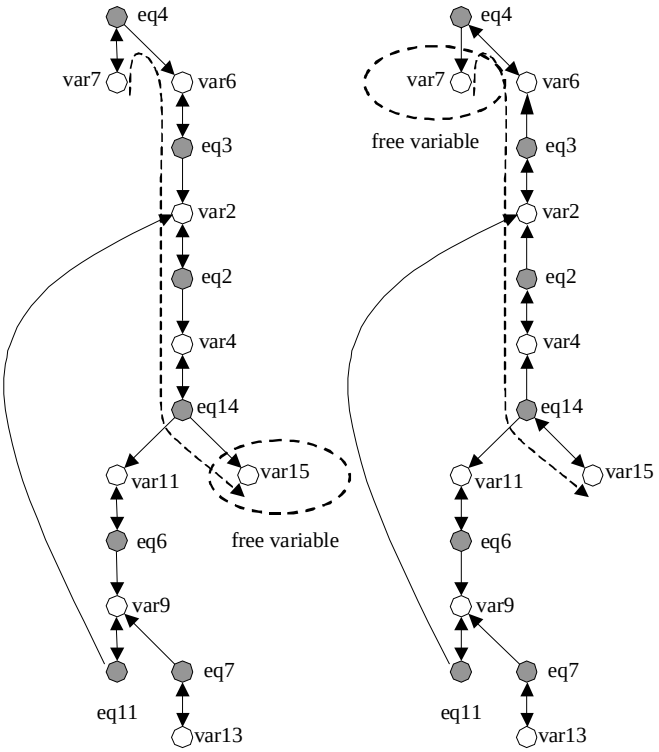



**Figure 6-9.** Exchanging matching edges with non-matching edges along an alternating path for obtaining a new matching that will cover *var7* and let *var15* be uncovered.

In the example presented above the fault was detected by applying the first strategy when debugging under-constrained systems that 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 variables 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. The possible coupling of variables to equations is performed by a *variable reachability analysis* based on algorithms applied to the inheritance graph of the underlying simula-

tion model. The variable reachability analysis computes the set of the variables that can be inserted into certain equations at certain levels in the class hierarchy.

A second strategy when debugging under-constrained systems is 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 on page 86, is a mathematically sound solution even though if it might not reflect the modeler 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/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.

## 6.3 Variable Reachability Analysis

When extra equations need to be added to the overall system of equations in order to repair under-constrained situations it is useful to know which variables can be included in the inserted new equations. We start by constructing the inheritance-instantiation graph of the simple electrical circuit (see Figure 6-10).

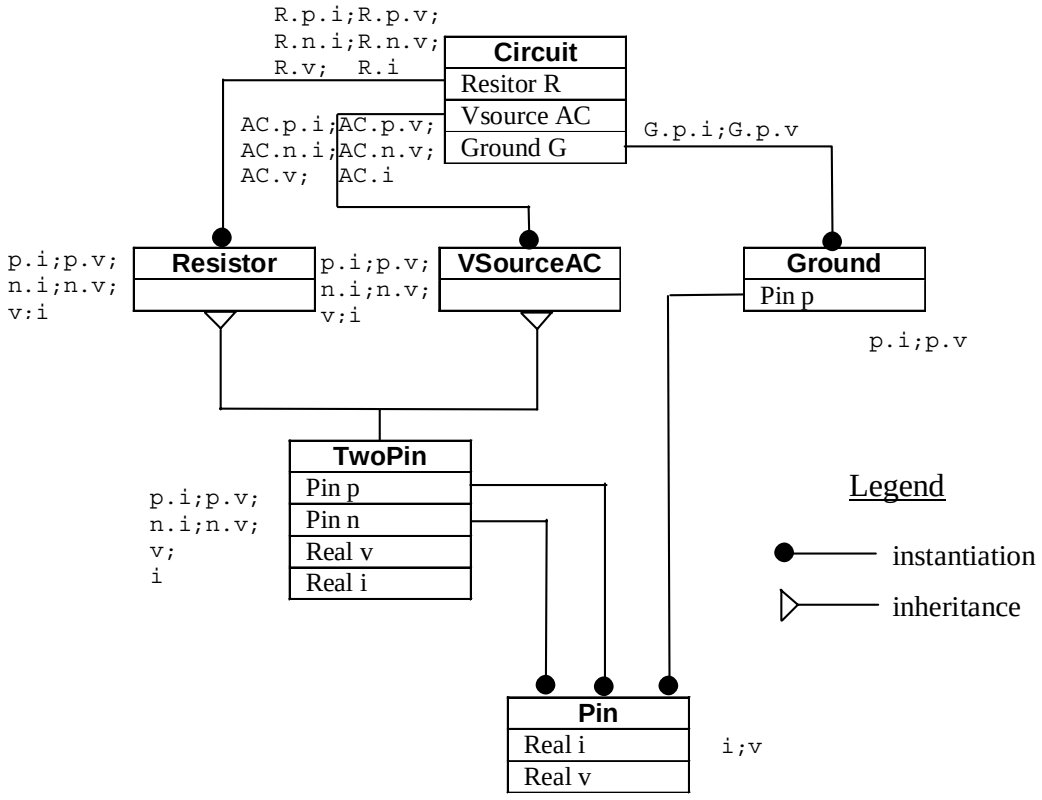



**Figure 6-10.** The variable propagation via the inheritance and instantiation relation for the `Circuit` model simulation, shown using UML graph notation.

The analysis starts from the bottom of the graph. In our example the `Pin` class declares two `Real` variables `i` and `v`. Only these two variables can be used in an equation that is

defined inside the `Pin` class. The declared variables `i` and `v` are propagated via the instantiation relation to the `TwoPin` and `Ground` objects where they get a prefix during the translation process. For example, by instantiating the `Pin` component (`Pin p`) in the `TwoPin` class, the real variables `i` and `v` declared in `Pin` are transformed into `p.i` and `p.v`. The variable transformations are repeated for the other declarations of a `Pin` component (`Pin n`) and the obtained variable names are added to the set of variables that can be used inside the `TwoPin` class. The other variables declared in the `TwoPin` class are also added to this set. The set of variables that can be used in a class is propagated unchanged through the inheritance relation. The `Resistor` and `VsourceAC` components will inherit the set of variable from the `TwoPin` class. Since the `Resistor` and `VsourceAC` classes don't declare any additional variables, the set remains unchanged and it is propagated through the instantiation relation to the `Circuit` class where each variable from the set is prefixed by the corresponding instance name.

The correspondence between the variable names at the class levels and the variable names at the intermediate flattened code level is illustrated in Table 6-2. It should be noticed that at the `Circuit` class level all the variables present in the flattened intermediate form can be used when creating a new equation at this level.

**Table 6-2.** Correspondence table between the variable names at the class levels and the variable names at the intermediate flattened code level.

| Model | Variables available (original code form) | Variables available (intermediate code form) | Constants Available |
|---|---|---|---|
| `Circuit` | `R.p.i R.p.v`<br>`R.n.i R.n.v`<br>`R.v R.i`<br>`AC.p.i AC.p.v`<br>`AC.n.i AC.n.v`<br>`AC.v AC.i`<br>`G.p.i G.p.v` | `R.p.i; R.p.v;`<br>`R.n.i; R.n.v;`<br>`R.v; R.i`<br>`AC.p.i; AC.p.v`<br>`AC.n.i; AC.n.v`<br>`AC.v; AC.i`<br>`G.p.i; G.p.v` | |
| `Resistor` | `p.i p.v`<br>`n.i n.v`<br>`v i` | `R.p.i; R.p.v`<br>`R.n.i; R.n.v`<br>`R.v; R.i` | R |
| `VSourceAC` | `p.i p.v`<br>`n.i n.v`<br>`v I` | `AC.p.i AC.p.v`<br>`AC.n.i AC.n.v`<br>`AC.v AC.i` | |
| `Ground` | `p.i p.v` | `G.p.i G.p.v` | |
| `TwoPin` | `p.i`<br>`p.v`<br>`n.i`<br>`n.v`<br>`v`<br>`i` | `{R.p.i; AC.p.i}`<br>`{R.p.v; AC.p.v}`<br>`{R.n.i; AC.n.i}`<br>`{R.n.v; AC.n.v}`<br>`{R.v; AC.v}`<br>`{R.i; AC.i}` | |
| `Pin` | `i`<br><br>`v` | `{R.p.i; R.n.i; AC.p.i;`<br>`AC.n.i; G.p.i;}`<br>`{R.p.v; R.n.v; AC.p.v;`<br>`AC.n.v; G.p.v;}` | |

Based on Table 6-2 and Figure 6-10 we illustrate the transformations performed on variable `i` declared at the `Pin` class level in Figure 6-11. The variable names below the class names, as shown in Figure 6-11, are used at the original source code level when new equations are added to the corresponding class behavior. When adding new additional equations to the model, the fact that they might generate multiple equations at the

flattened source code level should also be taken into account. For example, introducing an extra equation at the `TwoPin` class level that uses the variable `p.i` will create two equations in the flattened flat form: one equation that uses variable `R.p.i` and another equation that will use variable `AC.p.i`.

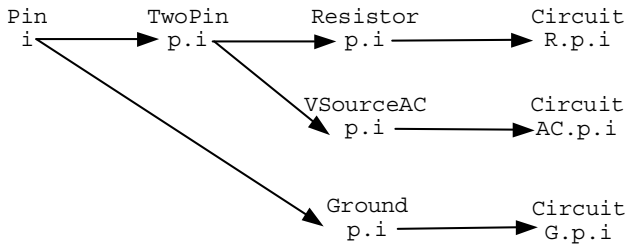



**Figure 6-11.** Transformations performed on the variable `i` declared in the `TwoPin` component through the inheritance and instantiation relations.

## 6.4 Insertion of Additional Equations

In order to illustrate the usability of the variable reachability analysis, presented in the previous section, let us consider a simple simulation model of a liquid tank model with a simple PID controller taken from (Fritzson 2002 [43]) and presented in Figure 6-12. Fluid can enter the tank through a pipe at a rate controlled via a valve and leaves the tank via another pipe. In our case, a fluid source object generates the fluid entering the tank. The fluid level `h` in the tank must be maintained at a fixed level as closely as possible by introducing a `PID` (proportional integrative derivative) continuous controller.

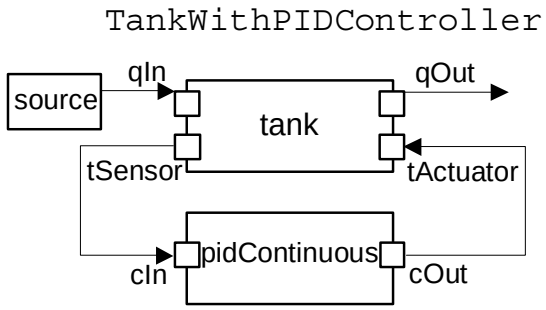



**Figure 6-12.** Simple `Tank` model with a PID controller.

The Modelica source code of each component is given below. A complete description of the model behavior including several simulation results is presented in (Fritzson 2002 [43]).

```
connector ActSignal                connector ReadSignal
  Real act;                          Real val;
end ActSignal;                     end ReadSignal;

connector LiquidFlow
  Real lflow;
end FlowIn;

partial model Controller
  parameter Real Ts = 0.1;     //sampling time[s]
  parameter Real K = 2;        //gain
  parameter Real T = 10;       //time constant[s]
  parameter Real ref = 0.25;
  Real error, outCtr;
  ReadSignal cInp; ActSignal  cOut;
equation
  error = ref-cInp.val;
  cOut.act = outCtr;
end Controller;

model PIDcontinuousController
  extends Controller(K = 2,T = 10);
  Real  x; // state variable of continuous controller
  Real  y; // state variable of continuous controller
equation
  der(x) = error/T;
  y= T * der(error);
  outCtr = K * (x + error + y);
end PIDcontinuousController;

model LiquidSource
  LiquidFlow qOut;
  parameter Real flowLevel=0.02;
equation
  qOut.lflow =  flowLevel;
end LiquidSource;

model Tank
  ReadSignal tSensor;   // Connector, reading tank level
  ActSignal tActuator;  // Connector, actuator controlling
                        // the input flow
  parameter Real area =0.5;       //[m2]
  parameter Real flowGain =0.05; //[m2/s]
  Real h(start=0.0);              //tank level [m]
  LiquidFlow qIn;   // flow through input valve[m3/s]
  LiquidFlow qOut;  // flow through output valve[m3/s]
equation
  der(h) = (qIn.lflow - qOut.lflow)/area; //mass balance
  qOut.lflow = -flowGain * tActuator.act;
  tSensor.val = h;
end Tank;
```

The final simulation model is obtained by connecting together the `Tank` and the `PID-continuousController` model instances `tankm` and `pid`. The `LiquidSource` model connected to the `Tank` provides the input liquid flow.

```
model TankWithPIDController
  LiquidSource source(flowLevel = 0.02);
  PIDcontinuousController pid(ref = 0.25);
  Tank tankm(area = 1);
equation
  connect(source.qOut,tankm.qIn);
  connect(tankm.tActuator, pid.cOut);
  connect(tankm.tSensor, pid.cInp);
end TankWithPIDController;
```

Let us show a simple debugging session performed on this model. Before starting the translation process we eliminate the equation

```
qOut.lflow  = - flowGain * tActuator.act;
```

from the `Tank` model. Obviously, this modification will lead to an under-constrained underlying system of equations. The flat form of the equations and the variables corresponding to the final simulation model, after the above mentioned equation removal, are given in Table 6-3 below:

**Table 6-3.** Flat form of the equations and variables corresponding to the simple Tank simulation model

| | | | |
|---|---|---|---|
| ***eq1*** | `source.qOut.lflow = source.flowLevel` | ***var1*** | `source.qOut.lflow` |
| ***eq2*** | `pid.error = pid.ref - pid.cInp.val` | ***var2*** | `pid.error` |
| ***eq3*** | `pid.cOut.act = pid.outCtr` | ***var3*** | `pid.outCtr` |
| ***eq4*** | `(pid.x)'= pid.error / pid.T` | ***var4*** | `pid.cInp.val` |
| ***eq5*** | `pid.y = pid.T(pid.error)'` | ***var5*** | `pid.cOut.act` |
| ***eq6*** | `pid.outCtr = pid.K *(pid.error+pid.x+pid.y)` | ***var6*** | `pid.x` |
| ***eq7*** | `(tank.h)' = (tankm.qIn.lflow – tankm.qQut.lflow) / tankm.area` | ***var7*** | `pid.y` |
| ***eq8*** | `tankm.tSensor.val==tankm.h` | ***var8*** | `tankm.tSensor.val` |
| ***eq9*** | `source.qOut.lflow==tankm.qIn.lflow` | ***var9*** | `tankm.tActuator.act` |
| ***eq10*** | `tankm.tActuator.act==pid.cOut.act` | ***var10*** | `tankm.h` |
| ***eq11*** | `tankm.tSensor.val==pid.cInp.val` | ***var11*** | `tankm.qIn.lflow` |
| | | ***var12*** | `tankm.qOut.lflow` |

The intermediate flattened form of the equations is mapped into bipartite graphs, a maximum cardinality matching is computed. All the edges that are not matching edges are directed from equation to variable nodes, and all the matching edges are transformed into bidirectional edges. We obtain the directed graph presented in Figure 6-13:

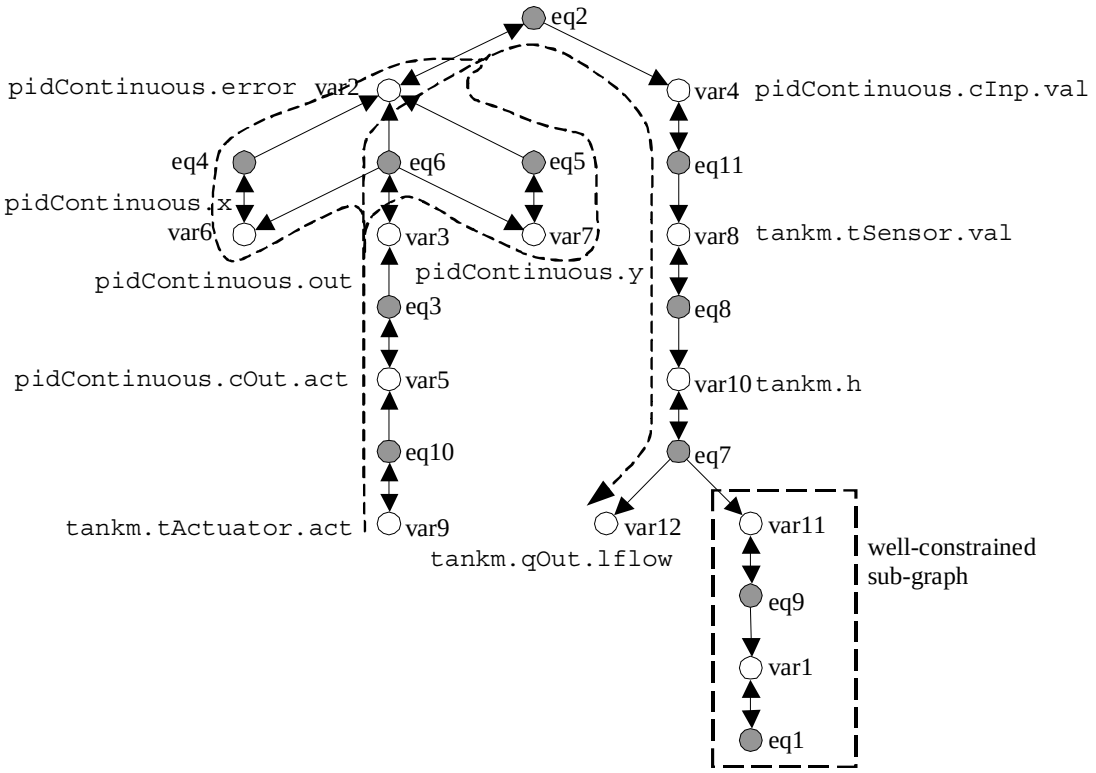



**Figure 6-13.** Directed graph corresponding to the tank simulation model.

By performing a D&M canonical decomposition the under- and well-constrained parts are isolated. In our particular example the well-constrained part is relatively small and contains only two equations and two variable nodes indicated by the dashed rectangle in Figure 6-13. The chosen maximum cardinality matching does not cover variable *var12*. A dashed arrow indicates the path via the equation and variable nodes that sinks into the free variable node *var12*. Any of the variable nodes along this path can be made into a free variable by exchanging matching edges by non-matching edges along the subpath from the chosen variable to the free variable *var12*.

At the fist stage, we consider the debugging alternative when a free variable is eliminated from the system. The set of variables that represents the eligibility set along the alternating path is:

{*var12, var10, var8, var4, var2, var6, var7, var5, var9*}

We eliminate from the set all the variable nodes that will disconnect the underlying bipartite graph if they are removed. We obtain the following set:

{*var12, var6, var7, var9*}.

The set {*var12, var6, var7, var9*} represents the set of variables at the intermediate code level but the error fixing messages need to be presented to the user at the original Modelica source code level. Let us first examine the error fixing solutions involving the elimination of the free variables from the set:

- Remove variable *var12* from equation *eq7*. This removal can be achieved by modifying the mass balance equation from the `Tank` model:
  `der(h) = (qIn.lflow - qOut.lflow)/area;` into

`der(h) = qIn.lflow / area;`
The declaration `LiquidFlow qOut` from the `Tank` model can also be removed.

- Remove variable *var6* from *eq4* and *eq6*. The removal can be achieved by removing variable `x` from
`der(x) = error / T` and from `outCtr = K * (x + error + y)`
in the `PIDcontinuousController` model. The equations are transformed into: `const = error / T` and `outCtr = K * ( error + y);`
The variable declaration `Real x;` can also be removed from the `PIDcontinuousController` model.
- Remove variable *var7* from *eq5* and *eq6*. In a similar way as for the previous case this modification is achieved by removing variable `y` from
`y = T * der(error)` and `outCtr = K * (x + error + y).`
The equations are transformed into:
`const = T * der(error)` and `outCtr = K * ( error + y);`
- Remove variable *var9* from *eq10*. The removal is achieved by modifying the connect equation `connect(tankm.tActuator,pid.cOut)` from the `TankWithPIDController` model. However, entirely eliminating the `connect` equation will eliminate *eq10*. The system is still under-constrained because we have eliminated one variable and one equation instead of eliminating only one variable. The debugger has all the necessary information to automatically invalidate and discard this case.

If none of the above mentioned error fixing solutions is convenient, the debugger can enter the second phase where instead of removing free variables from the system the possibility of adding a new equation which contains at least one free variable is taken into account.

We perform a variable reachability analysis and obtain the following correspondence table where X marks the possible presence of a variable inside the equations of the classes shown in the rows. For example, *var12* can be present in equations defined inside the `TankWithPIDController`, `Tank`, and `LiquidFlow` classes. For the exact form of the variables at the source code level a table that resembles Table 6-2 is also constructed.

**Table 6-4.** Variables from the eligibility set that can be used inside the classes of the simulation model.

| Model | var12 | var10 | var8 | var4 | var2 | var6 | var7 | var5 | var9 |
|---|---|---|---|---|---|---|---|---|---|
| TankWithPIDController | X | X | X | X | X | X | X | X | X |
| Tank | X | X | X | | | | | | X |
| PIDcontinuousController | | | | X | X | X | X | X | |
| LiquidSource | | | | | | | | | |
| LiquidFlow | X | | | | | | | | |
| ActSignal | | | | | | | | X | X |
| ReadSignal | | | X | X | | | | | |

By choosing the second level of debugging, multiple error fixing alternatives can be found. The user is, at this stage, required to provide the location where the additional equation should be inserted. For example, if the user chose to fix the problem at the

`Tank` model level, the set of free variables filtered by the reachability analysis is presented to the user. The additional equation that needs to be introduced will be constructed based on the provided set of free variables and set of parameters and constants that can be used at that level. The `LiquidFlow`, `ActSignal`, and `ReadSignal` models are not considered in any error fixing solution for this case because they are connector classes and cannot have an equation section. This semantic rule prevents the addition of an extra equation at this level. Let us now analyze the complete output given by the debugger:

```
ERROR!!!
Under-constrained situation detected in model TankWithPIDController.
No. of equations = 11
No. of variables = 12
General debugging options:
  General debugging options: Level 1
  Secondary debugging options: Level 4

Debugging STAGE1: free variable elimination

solution 1:
  model Tank
    remove variable qOut.lflow from der(h) = (qIn.lflow - qOut.lflow) / area;
    remove declaration  LiquidFlow qOut;

solution 2:
  model PIDcontinuousController
    remove variable x from der(x) = error / T  and from outCtr = K * (x + error + y)
    remove declaration Real x;

solution 3:
   model PIDcontinuousController
    remove variable y from y  = T * der(error) and from outCtr = K * (x+ error + y)
    remove declaration Real y;

Debugging STAGE2: adding extra equations

solution 4:
  model TankWithPIDController
    add new equation
    list of variables {pid.error, pid.outCtr, pid.cInp.val, pid.cOut.act, pid.x, pid.y,
    tankm.tSensor.val, tankm.tActuator.act, tankm.h, tankm.qOut.lflow}

solution 5:
  model Tank
    add new equation
    set of variables {  tSensor.val, tActuator.act, h, qOut.lflow}

solution 6:
model TankWithPIDController
    add new equation
    set of variables {error, outCtr, cInp.val, cOut.act, x, y}
```

**Figure 6-14.** Debugger output for the under-constrained `Tank` simulation model

Unfortunately refining the quality and quantity of the output relies heavily on the variable and equation annotations. The annotations can greatly reduce the number of possible options. Let us assume that the `PIDcontinuousController` model class has been taken from a previously developed library and has been used successfully in several simulation models. The user is confident regarding the behavior and the correct func-

tionality of this model, and he/she can annotate the equations with low flexibility levels or even lock them. Any error fixing solution involving the equations of the `PIDcontinuousController` model can be filtered out by the debugger. The previous debugger output presented in Figure 6-14 becomes:

```
ERROR!!!
Under-constrained situation detected in model TankWithPIDController.

No. of equations = 11
No. of variables = 12

General debugging options:
  General debugging options: Level 1
  Secondary debugging options: Level 4

Debugging STAGE1: free variable elimination

solution 1:
  model Tank
    remove variable qOut.lflow from der(h) = (qIn.lflow - qOut.lflow) / area;
    remove declaration  LiquidFlow qOut;

Debugging STAGE2: adding extra equations

solution 2:
  model TankWithPIDController
    add new equation
    set of variables {pid.error, pid.outCtr, pid.cInp.val, pid.cOut.act, pid.x, pid.y,
    tankm.tSensor.val, tankm.tActuator.act, tankm.h, tankm.qOut.lflow}

solution 3:
  model Tank
    add new equation
    set of variables {  tSensor.val, tActuator.act, h, qOut.lflow}
```

**Figure 6-15.** Filtered output of the debugger for the under-constrained `Tank` model.

By analyzing the first solution given by the debugger and performing the prescribed modifications, the tank model will be transformed into a model from where the output valve (previously represented by `LiquidFlow qOut`) is eliminated as shown below. Even if by this modification the underlying system of equations is mathematically sound, the model becomes physically incorrect because no output valve is provided any more.

```
model Tank
  ReadSignal tSensor;   // Connector, reading tank level
  ActSignal tActuator;  // Connector, actuator
  parameter Real area =0.5;       //[m2]
  parameter Real flowGain =0.05; //[m2/s]
  Real h(start=0.0);             //tank level [m]
  LiquidFlow qIn;   // flow through input valve[m3/s]
equation
  der(h) = qIn.lflow /area; //mass balance
  tSensor.val = h;
end Tank;
```

The user, based on modeling and physical considerations, can reject this solution, which was automatically provided by the debugger.

Analyzing the second solution provided by the debugger, one can notice that an error fixing solution involves too many combinations of variables that come from instances declared in the `TankWithPIDController` model. User intervention at this level can be postponed and is required only if other error fixing solutions provided by the debugger have failed. In general it is a good strategy to postpone the error fixing solutions that involve modifications in the final instantiated model.

Obviously the correct solution is to add an extra equation at the `Tank` model level involving the variables `qOut.lflow` and `tActuator.act`. This situation is covered by solution 3 provided by the debugger output shown in Figure 6-15. The additional equation that needs to be introduced is: `qOut.lflow = - flowGain * tActuator.act`; that provides an acceptable solution, both from the mathematical and the physical modeling points of view. One may also notice that the same effect can be achieved by introducing a new equation in the `TankWithPIDController` model:

```
tankm.qOut.lflow = - tankm.flowGain * tankm.tActuator.act
```

However, this solution will fix the problem for the present `TankWithPIDController` example and also results in a correct simulation model. An attempt to use the `Tank` model in another configuration will yield the same error.

The correct error fixing solution that involves the insertion of a new equation: `qOut.lflow = - flowGain * tActuator.act` in the `Tank` model is illustrated in Figure 6-16, where the previous under-constrained subgraph from Figure 6-13 is transformed into a well-constrained graph:

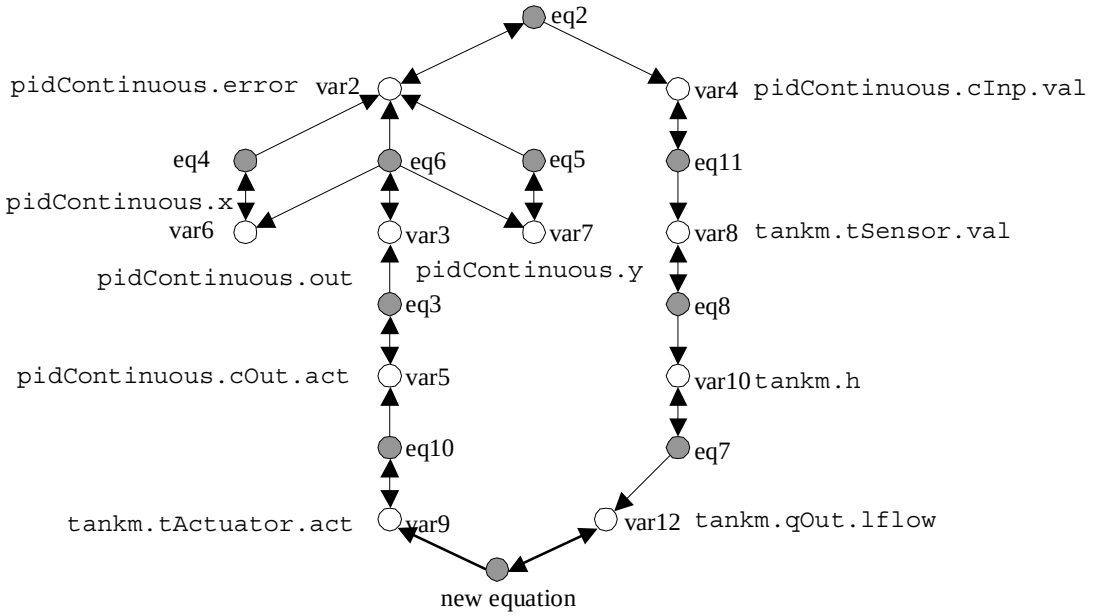



**Figure 6-16.** Making an under-constrained subgraph well-constrained by introducing an extra equation at the `Tank` model level.

A general algorithm (Algorithm 6-1) for debugging under-constrained systems with the degree of under-constraining $D_u = 1$, can be given. It is composed of two distinct stages. First the eligibility set corresponding to the under-constrained subgraph is computed. In the first stage, each variable from the eligibility set is checked to determine

whteher it disconnects the bipartite graph by elimination. If the variable doesn't split the graph the elimination need to be validated semantically by the function *validate-Sem*(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 6-1: Debugging under-constrained subsystems**

**Input Data:** The under-constrained subgraph $U_G^{1-}$ resulting after D&M decomposition has been performed on a graph *G*.
**Result:** 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 put them into the list *L*.
Compute the eligibility set $L_{el} = \{v_1, \ldots v_k \mid v_k \subset L \text{ and } v_k \in v_{V_2}(G)\}$

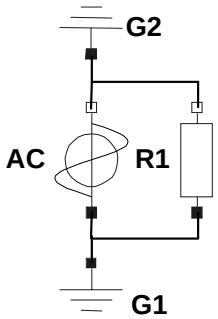
```
//stage1: free variable elimination
for each v_k ∈ L_el do
   remove v_k from U_G^1-
   compute the number of strongly connected components no_str of U_G^1- .
   if no_str == 1 then
      //validate semantically the elimination of v_k from the adjacent equations
      if validateSem( v_k ) then
          output("remove v_k from equation adj( v_k ) ");
      end if;
   end if;
end for;

//stage2: add new equations
for each v_k ∈ L_el do
    classList = reach( v_k );
    for each class c ∈ 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 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.

## 6.5 Debugging Simultaneous Over and Under-Constrained Situations

Let us consider the following simple electrical circuit where two `Ground` objects are present in the circuit instead of only one. The definition of the `Resistor`, `VsourceAC` and `Ground` objects are reused from the previous examples, only the `Circuit` model is modified by introducing a second instance of the `Ground` component and connecting it to the main circuit, as shown in Figure 6-17.




```
model TwoGroundCircuit
   Resistor R(R=10);
   VsourceAC AC;
   Ground G1,G2;
equation
   connect(AC.p,R.p);
   connect(R.n,AC.n);
   connect(AC.n,G1.p);
   connect(AC.p,G2.p);
end TwoGroundCircuit;
```

**Figure 6-17.** Modified circuit model with two `Ground` objects.

Obviously, the `TwoGroundCircuit` model is structurally inconsistent. Let us analyze the structure of the corresponding bipartite graph of the flattened form of the associated system of equations. By applying the D&M canonical decomposition we obtain an over-constrained subgraph and an under-constrained subgraph simultaneously present in the simulation model as shown in Figure 6-18. Both subcomponents have one node uncovered by the maximum matching, which means that we need to eliminate one equation node from the over-constrained component $O_{1G}^{1+}$ and one variable node from the under-constrained component $U_{1G}^{1-}$.

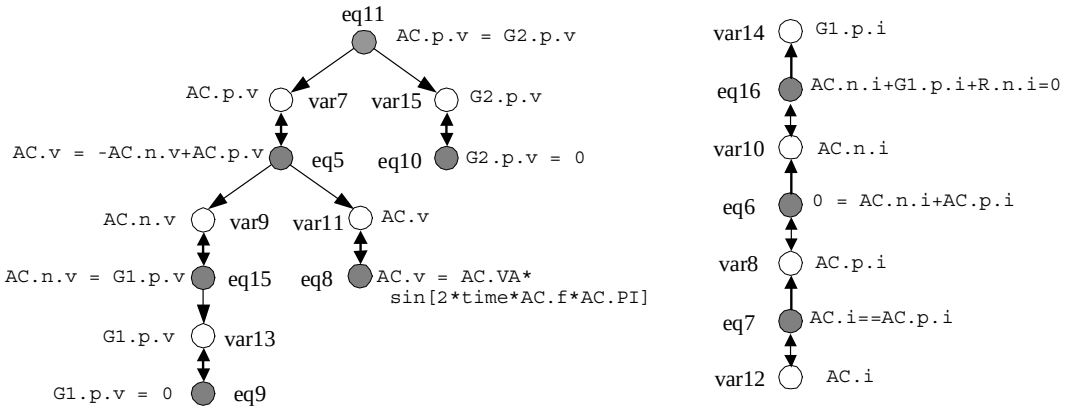



**Figure 6-18.** The over-constrained and under-constrained subgraphs obtained after the canonical decomposition of the bipartite graph representing the `TwoGround-sCircuit`.

The easiest way to solve this problem is to introduce the free variable into the free equation. In this case, both the under- and over-constrained situations can be solved simultaneously. A variable reachability analysis will immediately indicate that introducing *var14* into equation *eq9* is possible for example by modifying the equation `p.v = 0` from the `Ground` class to `p.v = 2 * p.i`.

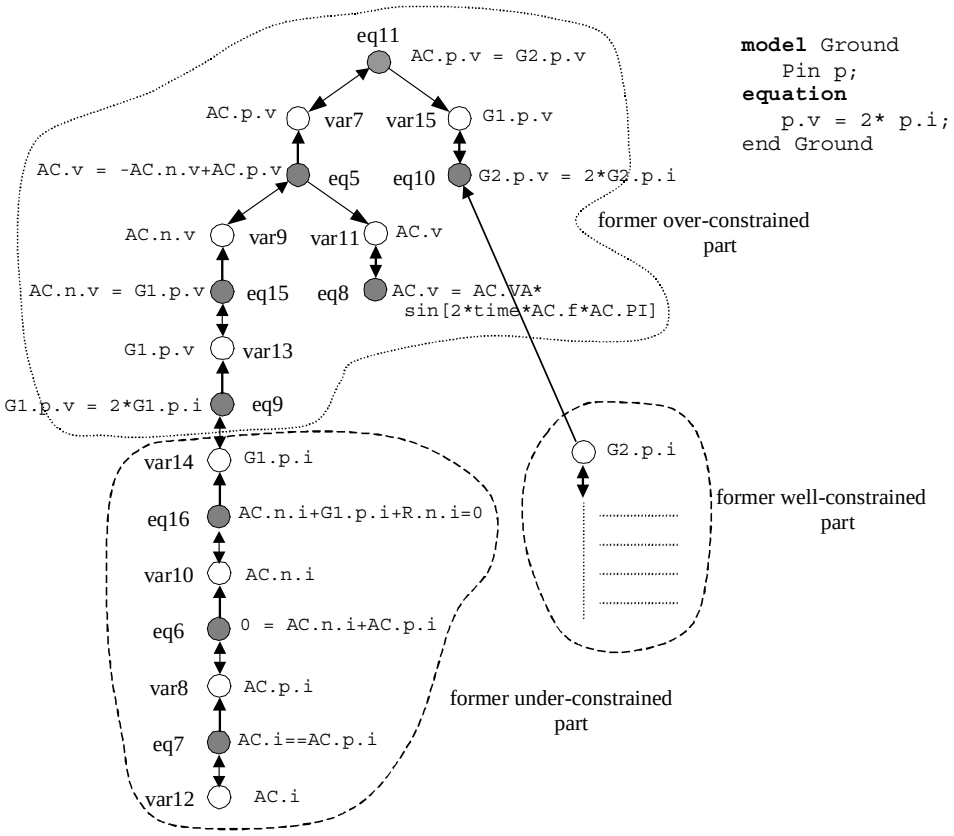



**Figure 6-19.** The transformation of the under- and over-constrained subgraphs corresponding to the `TwoGroundCircuit` into a well-constrained graph by introducing the equation `p.v = 2 * p.i` in the `Ground` component.

Performing this modification the model becomes consistent and it can be successfully compiled and even executed. But obviously the behavior of the model will not match the specification behavior and therefore another error fixing solution needs to be found. In other examples of simulation models introducing a free variable into an over-constraining equation might be a valid error fixing solution.

We start the debugging process by first analyzing the under-constrained component $U_{1G}^{1-}$. Variable nodes *var10* and *var8* can immediately be eliminated from any further consideration because their removal disconnects the remaining bipartite graph which is not feasible. Therefore only variable nodes *var14* and *var12* are included in the safe variable list. Let us first consider *var12* for elimination. The only way to eliminate *var12* (`Ac.i`) from *eq7* (`AC.i==AC.p.i`) is by removing the equation `i = p.i` from the `TwoPin` class. However, this removal will trigger the removal of the equation `R.i = R.p.i` that is not in the equation node present in the over-constrained graph $O_{1G}^{1+}$. In conclusion, taking away *var12* from the under-constrained graph is not possible by simple source code manipulations without structurally affecting the whole underlying bipartite graph.

The next step is to check whether the elimination of *var14* (`G1.p.i`) is possible or not. The only way to remove *var14* from equation *eq16* `(AC.n.i+G1.p.i+R.n.i=0)` is to remove the connect equation `connect(AC.n,G1.p)` from the `TwoGroundCircuit` model. Equation *eq16* will not totally be removed because the connect equation `connect(R.n,AC.n)` which still is present in the model will generate the equation `AC.n.i + R.n.i=0`. However the removal of the connect equation will totally remove another equation from the flattened intermediate form: *eq15* (`AC.n.v = G1.p.v`).

The removal of *eq15* will disconnect the associated bipartite graph and isolate *eq9* and *var13*. This solution is only acceptable if we are also allowed to eliminate *eq9* and *var13* from the graph. However, by more carefully examining *eq9* and *var13* we notice that they are variable and equation nodes that define the behavior of the `Ground` component as is illustrated in Table 6-5.

**Table 6-5.** The `Ground` component and its equations and variables.

| Ground Model Definition | Generated equations and variables | Scheduled for elimination |
|---|---|---|
| **model** Ground<br>Pin p;<br>**equation**<br>p.v=0;<br>**end** Ground | `G1.p.i` | yes |
| | `G1.p.v` | yes |
| | `G1.p.v = 0` | yes |
| **connect**(AC.n,G1.p) | `AC.n.i+G1.p.i+R.n.i=0` | remove only `G1.p.i` |

All the variables and equations that define the instance `G1` of the `Ground` component can be removed from the bipartite graph and a new bipartite graph with a perfect matching can be obtained as is illustrated in Figure 6-20. The removal of *eq15* from the over-constrained subgraph was dictated by a variable removal from the under-constrained subgraph. If we would attempt to treat the over-constrained problem in isolation, *eq15* wouldn't be considered for elimination due to the fact that its removal disconnects the remaining bipartite graph.

This example has illustrated that when over- and under-constraining problems appear simultaneously in a model they cannot be treated individually. Modifications in one subgraph can easily trigger modifications in the other subgraph and the error fixing solution validation need to be made globally. This example also revealed the interaction between the variable and equation elimination mechanisms. Finding systematic and algorithmic methods to efficiently treat the complex interactions between the under- and over-constrained situations when they appear simultaneously is still part of the future work of this debugging framework.

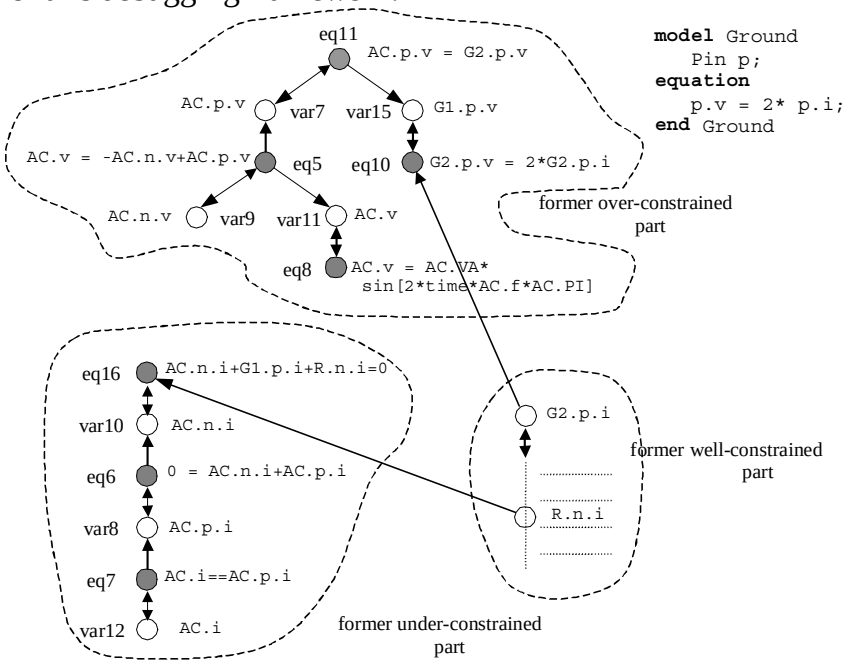



**Figure 6-20.** The bipartite graph corresponding to the simple circuit model from which the `Ground` component `G1` has been removed.

## 6.6 Conclusions Regarding Under-Constrained Systems

Obviously, the multitude of error fixing solutions at the second stage when additional equations need to be added to the system of equations is the bottleneck of the method. At this stage the user must specify in which class the new equation should be inserted. This task can be simplified if certain class components have previously been marked as correct with the help of annotations in other simulation configurations. However, providing the location where a new equation should be inserted and the list of adjacent variables to that equation has turned out to be extremely useful for the user. The implemented debugger prototype uses enhanced variants of Algorithm 6-1 for automatic debugging of under-constrained situations. Algorithm 6-1 has been also extended to handle under-constrained situations with a high degree of under-constraining ($D_u > 1$). Filtering the error fixing solutions based on the equation annotations is also included in the enhanced algorithm used by the prototype debugger.

# Chapter 7

# Structural Consideration of DAE index in Physical System Simulation

***Summary****: This chapter presents structural considerations of the high index problems when modeling with equation-based languages since the notion of index influences the debugging process. There are two definitions of index: the differential index which is the basic definition, and the structural index which is an approximation. The structural index is usually but not always identical to the differential index. An algorithm for computing the structural index based on matchings in bipartite graphs is also given. Current limitations of the graph-based structural approach are briefly presented. For example, the structural index of a linear differential-algebraic equation (DAE) with constant coefficients and with index 1 may be arbitrarily high, a phenomenon which is not always captured by the graph based structural analysis. Due to this phenomenon numerical cancellation may occur because Pantelides' algorithm applied to DAEs of index 1 may perform an arbitrary high number of iterations and differentiations. In such cases a debugger based on graph based structural analysis algorithms might fail to render the correct answer when this situation occurs. Symbolic pre-processing methods for the computation of consistent initial conditions and reduction of higher index DAE problems to an index one problem are also presented here in the context of the proposed debugging framework.*

## 7.1 Introduction

Many physical problems are most naturally described by systems of differential and algebraic equations. Usually, differential equations originate from balances of mass, energy, and momentum, from constitutive equations (equations of state, pressure drops, heat transfer, etc.) and from design and modeling constraints. Many dynamic problem formulations contain path constraints on the state variables, which will inevitably lead to high index problems. Numerical methods applicable for DAE with index 1 might not be useful for high index problems. High index DAE systems require special solvers or symbolic reduction of the problem to index 1. Numerous modeling examples exist that exhibit a high differential index. These cases are generally interesting from the struc-

tural analysis point of view. The debugging of many physical system models is influenced by the notion of index. The following sections present several examples derived directly from simulation models and details of how these situations affect the debugging of such models.

## 7.2 Differential Index

The index of a system of differential algebraic equations is one of the measures of solvability for the numerical problems in physical system simulation when solving differential equations. We should start by giving the most general representation of a DAE system in the nonlinear implicit form $F(u, \dot{y}, y) = 0$, where $y(t)$ represents the unknown state variables and $u(t)$ denotes the known input variables. Sometimes the system is an ODE (ordinary differential equation system), if it only contains ordinary derivatives (of one or more dependent variables) with respect to a single independent variable and it can be rewritten to the explicit form $\dot{y} = F(u, y)$.

**Definition 7.1**: The differential index of a general DAE system is the minimum number of times that all parts of the system must be differentiated with respect to *u* to reduce the system to a set of ODEs for the variable *y* (determine $\dot{y}$ as a continuous function of y and *u*).

$$\begin{aligned} &F(u, y, \dot{y}) = 0 \\ &\frac{dF}{du}(u, y, \ddot{y}) = 0 \\ &\cdots\cdots\cdots\cdots\cdots\cdots \\ &\frac{dF^{d}}{du}(u, y, y^{d+1}) = 0 \end{aligned} \tag{7-1}$$

***Remark:*** The index of an ODE is always 0.

***Example:*** Let us consider the semi-explicit form of a DAE system. In order to determine the index of the DAE system, we need to determine the minimum number of times that $g(x, z, t) = 0$ need to be differentiated with respect to time in order to yield a pure ODE. The general transformation is given below:

$$\begin{cases} \dot{x} = f(x, z, t) \\ g(x, z, t) = 0 \end{cases} \Rightarrow \begin{cases} \dot{x} = f(x, z, t) \\ \dot{z} = s(x, z, t) \end{cases}$$

Now, let us determine the differential index for the following DAE system:

$$\begin{cases} \dot{x} = z + \gamma_1(t) \\ 0 = x + \gamma_2(t) \end{cases}$$

We can start the process of transformation of this DAE equation system into an ODE by differentiating $0 = x + \gamma_2(t)$ and substituting $\dot{x}$ in $\dot{x} = z + \gamma_1(t)$.

$$\begin{cases} \dot{x} = z + \gamma_1(t) \\ 0 = x + \gamma_2(t) \end{cases} \Leftrightarrow \begin{cases} \dot{x} = z + \gamma_1(t) \\ \dot{x} = -\dot{\gamma}_2(t) \end{cases} \Leftrightarrow \begin{cases} -\dot{\gamma}_2(t) = z + \gamma_1(t) \\ \dot{x} = -\dot{\gamma}_2(t) \end{cases} \Leftrightarrow \begin{cases} z = -\dot{\gamma}_2(t) - \gamma_1(t) \\ \dot{x} = -\dot{\gamma}_2(t) \end{cases}$$

We differentiate the first equation $z = -\dot{\gamma}_2(t) - \gamma_1(t)$ again in the obtained system of equations:

$$\begin{cases} z = -\dot{\gamma}_2(t) - \gamma_1(t) \\ \dot{x} = -\dot{\gamma}_2(t) \end{cases} \Leftrightarrow \begin{cases} \dot{z} = -\ddot{\gamma}_2(t) - \dot{\gamma}_1(t) \\ \dot{x} = -\dot{\gamma}_2(t) \end{cases}$$

The index gives a classification of DAEs with respect to their numerical properties and can be seen as a measure of the distance between the DAE and the corresponding ODE (Günther and Feldmann 1995 [51]). When solving an ODE the solution procedure only involves integration. However, when solving a DAE system the solution may involve differentiation as well. For systems with index greater than 1, numerical methods may converge poorly, may converge to the wrong solutions, or may not converge at all. Higher index DAE may have hidden algebraic constraints between the dependent variables and/or derivatives. Hidden algebraic constraints complicate the problem of providing proper initial conditions for the system and also present difficulties for numerical integration methods.

Several approaches for solving high index DAE have been proposed in the literature. These are basically divided into two general approaches (Unger et. al 1995 [118])

- Numerical integration methods for specifically designed higher-index solvers.
- Reduction of the index by symbolic methods and applying a normal DAE index 1 solver for computing the solutions.

## 7.3 Structural Index

The structural index has been proposed as an efficient computational alternative to the differential index. As will be shown in the next section, an efficient computational algorithm for computing the structural index can be implemented within the more general framework for compiling equation-based languages. Computing of the structural index is reduced to the problem of finding maximum weighted matchings of different cardinalities in bipartite graphs that represent the system of differential equations.

### 7.3.1 Definition and Algorithm

The following definition of the structural index is given in (Feehery and Barton 1998 [38]) and is closely related to Pantelides' algorithm (Pantelides 1988 [93]).

**Definition 7.2:** (Structural Index of a DAE): The structural index $i_{str}$ of a DAE is the minimum number of times that any subset of equations in the DAE is differentiated using Pantelides' algorithm.

In order to illustrate the structural index algorithm let us consider the planar pendulum model taken from (Mattsson et. al. 2000 [81]), (Pantelides 1988 [93]) and (Modelica Assoc. 2000 [85]). The pendulum model (see Figure 7-1) is expressed in a Cartesian coordinate system and consists of a heavy body of mass *m* suspended by a rigid rod of length *L*. The air resistance and friction is ignored in this model representation.

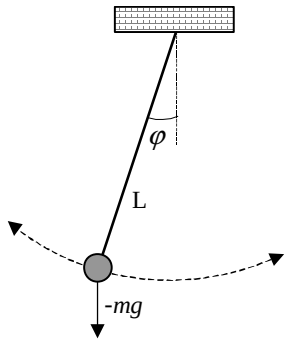



**Figure 7-1.** Planar pendulum model

Applying Newton's second law to the pendulum results in a nonlinear second order differential equation:

$$m\ddot{x} = -\frac{x}{L}F; \quad m\ddot{y} = -\frac{y}{L}F - mg \tag{7-2}$$

where *F* is the force in the rod, *g* is the gravitational acceleration and *x,y* are the coordinates of the mass *m*. In order to fully specify the physical system the following geometrical constraint need to be added:

$$x^2 + y^2 = L^2 \tag{7-3}$$

Combining the differential equations from (7-2) and (7-3) we obtain the DAE system of (7-4) below. It is impossible to directly solve equation (7-4) because the geometric constraint does not contain the higher order derivatives $\ddot{x}, \ddot{y}$ and *F*. The length constraint equation needs to be differentiated, which yields the system of equations presented in (7-5). The equation system obtained still does not contain the higher derivates and therefore needs to be differentiated once more. This operation will result in the system of equations shown in (7-6). In conclusion, the equation (7-4) has been differentiated twice in order to get the system into a suitable form for computing the solutions with a general use index 1 solver. Therefore the differential index of the system is equal to 2.

$$\begin{aligned} m\ddot{x} &= -\frac{x}{L}F \\ m\ddot{y} &= -\frac{x}{L}F - mg \\ x^2 + y^2 &= L^2 \end{aligned} \tag{7-4}$$

$$\begin{aligned} m\ddot{x} &= -\frac{x}{L}F \\ m\ddot{y} &= -\frac{x}{L}F - mg \\ x^2 + y^2 &= L^2 \\ 2x\dot{x} + 2y\dot{y} &= 0 \end{aligned} \tag{7-5}$$

$$\begin{aligned} m\ddot{x} &= -\frac{x}{L}F \\ m\ddot{y} &= -\frac{x}{L}F - mg \\ x^2 + y^2 &= L^2 \\ 2x\dot{x} + 2y\dot{y} &= 0 \\ 2\dot{x}\dot{x} + 2\dot{y}\dot{y} + 2x\ddot{x} + 2y\ddot{y} &= 0 \end{aligned} \tag{7-6}$$

Now, a method that computes or approximates the differential index only based on structural information needs to be provided. The algorithm for computing the structural index starts by constructing the bipartite graph associated to the system of differential

equations. When considering the bipartite graph representation of the DAE system the numerical values of the coefficients are ignored. To each edge a weight is assigned. The value of the assigned weight is equal to the order of the highest derivative of the variable present in the equation. For those edges that do not represent the presence of a derivative in an equation, a weight equal to zero is assigned

As was mentioned in the previous chapters, a matching *M* associated to the bipartite graph *G* is a set of edges from graph *G* where no two edges have a common end vertex. The weight of a bipartite matching is defined as the sum of the weights corresponding to each edge included in the matching. The algorithm for computing the structural index will proceed by computing the weight of the perfect matching associated to the bipartite graph.

$w(M) = \sum_{i=1}^{|M|} w_i$ where $|M|$ is the number of edges contained in the matching *M.*

The algorithm for computing the structural index is presented below:

**Algorithm 7-1: Computing the Structural Index**

**Input Data:** system of differential algebraic equations

**Result:** The numerical value of the structural index $i_{str}(A)$

**begin:**

- Associate the corresponding bipartite graph $G = (V_1, V_2, E)$ to the system of differential equations where $V_1 = \{v_1, v_2, \ldots, v_n\}$ and $V_2 = \{u_1, u_2, \ldots, u_n\}$ are the sets of vertices.
- Assign a weight to each edge $(v_s, u_t) \in E(G)$ where $1 \le s,t \le n$ equal to the order of the highest derivatives of the variable $u_t$ present in the equation $v_s$.
- Find the family of all matchings of size *n* $\Im(\mathbb{M}_n)$ in the bipartite graph $G$.
- **for each** $M$ **in** $\Im(\mathbb{M}_G^n)$ **do**
  - Calculate the weight of each matching $w(M) = \sum_{(i,j)\in M} w_{ij}$
  - **if** $w(M) > maxM_G^n$ **then** $maxM_G^n = w(M)$.

  **end for.**
- Find the family of all matchings of size *n-1* $\Im(\mathbb{M}_G^{n-1})$ in the bipartite graph $G$.
- **for** each $M$ **in** $\Im(\mathbb{M}_G^{n-1})$ **do**
  - Calculate the weight of each matching $w(M) = \sum_{(i,j)\in M} w_{ij}$
  - **if** $w(M) > maxM_G^{n-1}$ **then** $maxM_G^{n-1} = w(M)$.

  **end for.**
- Compute the structural index $i_{str}(G) = maxM_G^{n-1} - maxM_G^n + 1$[5]

**end.**

[5] For a complete demonstration of how this formula is obtained see Murota [28]

One naïve solution to the problem of computing the maximum weighted maximum matching is to compute all the maximum cardinality matchings, calculating the weight of each of these matchings and choose the one with the maximum weight. This method is used in Algorithm 7-1. Efficient implementations of algorithms for directly computing the maximum weighted matchings can be found in (Mehlhorn and Näher 1999 [83]).

Let us now consider the bipartite graph associated to the system of DAEs from (7-4) shown in Figure 7-2 a) below. Since the highest derivatives $\ddot{x}, \ddot{y}$ are present in the first and the second equation we are assigning a weight equal to 2 to the edges *(eq1,x)* and *(eq2,y)*. From the bipartite graph representation the family of all the maximum weighted matchings (family of size 3 in our case), $\Im(\mathbb{M}_G^3) = \{M_{1G}^3, M_{2G}^3\}$ is computed. The weights corresponding to the perfect matching

$$M_{1G}^3 = (\{eq1, eq2, eq3\}, \{x, y, F\}, \{(eq1, F), (eq2, y), (eq3, x)\})$$

is represented in Figure 7-2 b) with thicker lines and is equal to 2 because it includes one edge *(eq2,y)* which has a weight equal to 2, the weight of the other edges being equal to 0. It is computed by the following formula:

$$w(M_{1G}^3) = \sum_{i=1}^{3} w_i = w(eq1, F) + w(eq2, y) + w(eq3, x) = 0 + 2 + 0 = 2$$

Another possible maximum weighted perfect matching of the bipartite graph is $M_{2G}^3 = (\{eq1, eq2, eq3\}, \{x, y, F\}, \{(eq1, x), (eq2, F), (eq3, y)\})$ with the same numerical value for the weight $w(M_{2G}^3) = 2$. In conclusion for computing the maximum weight of the family of perfect matchings we can choose either $M_{1G}^3$ or $M_{2G}^3$.

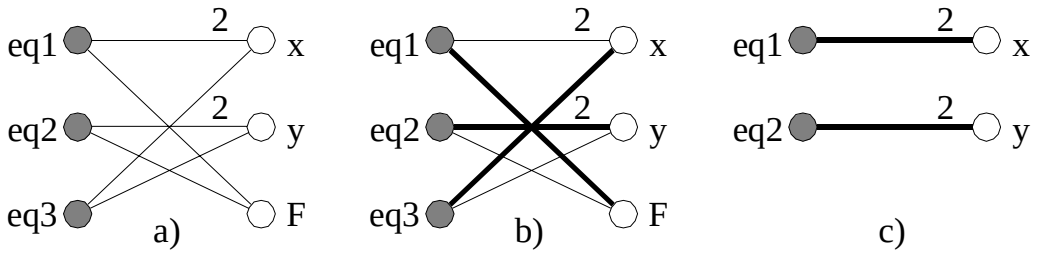



**Figure 7-2. a)** Associated bipartite graph for the DAE system presented in (7-4) **b)** Perfect matching of the bipartite graph **c)** Matching where equation *eq3* and variable *F* have been eliminated from the bipartite graph.

The maximum weighted matching of cardinality equal to 2 is shown in Figure 7-2 c). and has a total weight equal to 4. The structural index is computed by the following formula (Murota 2000 [87]):

$$i_{str}(G) = \max_{M_G^{n-1} \in \Im(\mathbb{M}_G^{n-1})} w(M_G^{n-1}) - \max_{M_G^n \in \Im(\mathbb{M}_G^n)} w(M_G^n) + 1 = 4 - 2 + 1 = 3$$

which in this case corresponds to the value of the differential index.

By differentiating equation (7-3) we can check whether the DAE system structural index is decremented by one. By differentiating equation (7-3) the DAE system (7-5) is obtained which has the associated bipartite graph shown in Figure 7-3. Weights equal to 2 are associated with the edges *(eq1,x)* and *(eq2,y)* and weights equal to 1 are associated with the edges *(eq3,x)* and *(eq3,y).* By differentiating equation (7-3) the first order derivatives of variables `x` and `y` will be present in these equations resulting in the corresponding edges having a weight equal to 1.

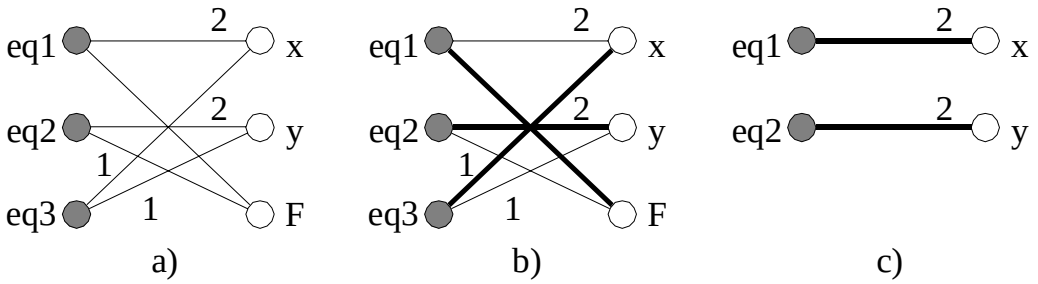



**Figure 7-3. a)** The associated bipartite graph to the system of DAE equations from (7-5). **b)** Perfect matching of the bipartite graph **c)** Maximum lover size matching.

Computing the maximum weighted matchings of size 3 and 2, and computing the structural index:

$$i_{str}(G) = \max_{M_G^{n-1} \in \Im(\mathbb{M}_G^{n-1})} w(M_G^{n-1}) - \max_{M_G^n \in \Im(\mathbb{M}_G^n)} w(M_G^n) + 1 = 4 - 3 + 1 = 2$$

Here we see that the structural index decreased by one when equation (7-3) was differentiated.

### 7.3.2 Index Preserving Differential Equation Rewriting

Since most equation-based modeling languages only support syntax for first order derivatives we rewrite equations (7-2) by introducing $v_x$ and $v_y$ for the velocity components.

$$m\dot{v}_x = -\frac{x}{L}F$$

$$m\dot{v}_y = -\frac{y}{L}F - mg$$

$$x^2 + y^2 = L^2$$

$$\dot{x} = v_x$$

$$\dot{y} = v_y$$

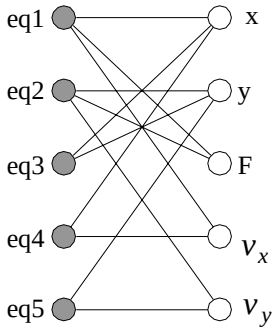



**Figure 7-4.** Pendulum model equations and the corresponding bipartite graph.

Now we are going to demonstrate that the symbolic substitution of the highest derivative and introduction of one extra equation and variable in the system of differential equations will not change the maximum weight of the matching corresponding to the associated bipartite graph. Therefore the value of the computed structural index will not be influenced by equation transformations that introduce implicit definitions of high order derivatives by adding a new algebraic variables. Similar transformations on differential equation systems are employed by certain index reduction algorithms such as the dummy derivatives method (Mattsson and Söderlind [80]). Therefore it is important to demonstrate that the structural index remains unchanged during such equation transformations.

The transformation, at the equation level, consists of introducing a new lower order derivative variable $x^d$ to represent $y^{d+1}$ wherever it appears in the system of differential equations.

For example the differential equation $\frac{dF^d}{dt}(t, y, y^{d+1}) = 0$ from (7-1) will be substituted by the equation $\frac{dF^{d-1}}{dt}(t, y, x^d) = 0$ and a new variable x together with a new equation $y^{d+1} = x^d$ will also be added to the overall equation system in order to avoid introducing an over-constrained situation.

In the corresponding bipartite representation this will transform an edge into an alternating path. The weight assigned to the substituted edge is equal to the highest derivative present in the equation to be substituted. A two-step edge substitution is presented below together with the corresponding graph transformation:

$$\left\{ \dddot{x} = \cdots \right. \Rightarrow \left\{ \begin{array}{l} \dot{s} = \cdots \\ \ddot{x} = s \end{array} \right. \Rightarrow \left\{ \begin{array}{l} \dot{s} = \cdots \\ \dot{t} = s \\ \dot{x} = t \end{array} \right.$$

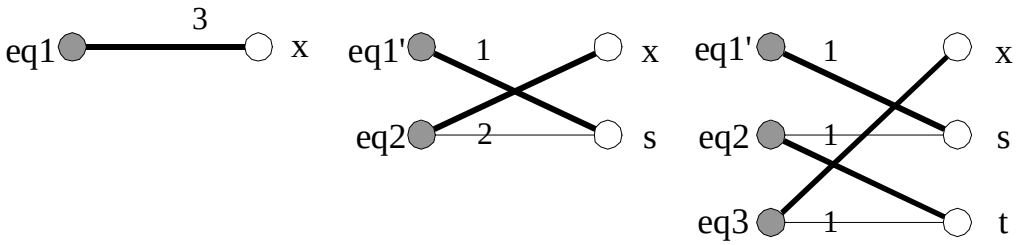



**Figure 7-5.** Transformation of an equation containing a third order derivative into three equations containing first order derivatives.

**Theorem 7-1:** The structural index of a DAE is not changed by simple symbolic derivative rewriting transformations.

**Proof**: Let us consider a balanced bipartite graph $G = (V_1, V_2, E)$, which has a corresponding perfect matching $M$. Let $V_1 = \{v_1, v_2, \ldots, v_n\}$ and $V_2 = \{u_1, u_2, \ldots, u_n\}$. We shall show that:

1. The graph $G'$ obtained by exchanging an edge $(v_s, u_t) \in E(G)$ where $1 \leq s,t \leq n$, with a sequence of edges $\{(v'_s, u_{n+1}), (u_{n+1}, v_{n+1}), (v_{n+1}, u_t)\}$ is bipartite and admits a perfect matching $M'$.
2. If $w((v_s, u_t)) = i$ where $(v_s, u_t) \in E(G)$ and $w((v'_s, u_{n+1})) = 1$, $w((u_{n+1}, v_{n+1})) = 0$ and $w((v_{n+1}, u_t)) = i - 1$ where $(v'_s, u_{n+1}), (u_{n+1}, v_{n+1}), (v_{n+1}, u_t) \in E(G')$ then $w(M) = w(M')$.

**1:** The demonstration that $G'$ is bipartite is obvious. It is clear that $V_1$ and $V_2$ are two distinct sets such that no two vertices from the same set are adjacent because the graph $G$ is considered to be a bipartite graph. The equation node $v_{n+1}$ is adjacent to $u_t \in V_2$ and $u_{n+1}$. The variable node $u_{n+1}$ is adjacent to $v_s \in V_1$ and $v_{n+1}$. The introduced new vertices $v_{n+1}$ and $u_{n+1}$ can be added to $V_1$ and to $V_2$ respectively without having vertices that are adjacent to the same set. Then it follows that $G'$ only has edges joining vertices from $V'_1 = V_1 + \{v_{n+1}\}$ and $V'_2 = V_2 + \{u_{n+1}\}$ and therefore that $G'$ is bipartite. That $G'$ admits a perfect matching is a consequence of *Hall's Theorem* (see Chapter 4, Theorem 4-1).

**2:** Clearly it suffices to consider the case where the only weighted edges in the corresponding bipartite graphs are $(v_s, u_t) \in E(G)$ and $(v_s, u_{n+1}), (v_{n+1}, u_t) \in E(G')$. If $(v_s, u_t) \in M(G)$ then

$$w(M) = \sum_{(i,j) \in M} w_{ij} = w(v_s, u_t) = i .$$

Eliminating the edge $(v_s, u_t)$ means that $v_s, u_t$ are edges which are not covered any more by a matching and therefore $M'$ will contain two extra matching edges $\overline{(v_s, u_{n+1})}$ and $\overline{(v_{n+1}, u_t)}$ because the vertices $v_s$ and $u_t$ need to be covered by matching edges. The weight of $M'$ is given by the following formula:

$$w(M') = \sum_{(i,j) \in M'} w_{ij} = w(v_s, u_{n+1}) + w(v_{n+1}, u_t) = (1 + i) - 1 = i .$$

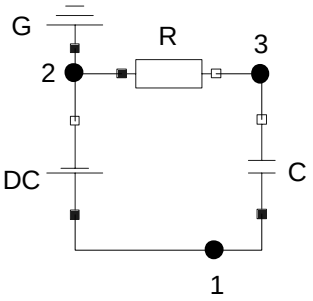
If $(v_s, u_t) \notin M(G)$ then $w(M) = 0$. Eliminating the edge $(v_s, u_t)$ from $G$ will not change the structure of the associated matching $M$ and the vertices $v_s$ and $u_t$ are covered. Therefore any edge incident with these vertices does not need to be a matching edge in a new matching. In conclusion only $\overline{(u_{n+1}, v_{n+1})}$ can be a matching edge in the formed new matching $M'$. Since $w(\overline{u_{n+1}, v_{n+1}}) = 0$ then $w(M')$ is also equal to 0.

## 7.4 Limitations of Graph Based Structural Analysis

The current limitations of the graph-based structural approach are presented in the following subsections. First an example taken from (Reissig et. al. 2000 [100]) is presented where a system of DAEs with differential index 1 may have an arbitrarily high structural index. This phenomenon is not always captured by a structural approach and therefore numerical cancellation may occur later at the numerical solving phase of the equations. Another "embarrassing" phenomenon is when the structural index is less than the differential index of the associated differential equations, which also may pose difficulties to index reduction algorithms and to numerical solvers.

### 7.4.1 Structural Index Higher than the Differential Index

Let us consider the example of a simple electrical circuit taken from (Reissig et. al. 2000 [100]). An electrical resistor `R` is connected in series to a constant voltage source `DC` and a capacitor `C` as shown in Figure 7-6. If node 2 is chosen as the ground node the system of equations (7-7) is derived from the circuit.




$$
\begin{aligned}
& v_2 = 0 \\
& v_3 - v_2 = R * i_R \Leftrightarrow v_3 = R * i_R \\
& C\frac{dv}{dt} = i \Leftrightarrow C(\dot{v}_1 - \dot{v}_3) = i \Leftrightarrow C\dot{v}_1 - C\dot{v}_3 - i = 0 \\
& v_2 - v_1 = v(t) \Leftrightarrow -v_1 = v(t) \\
& i_R = i_C \Leftrightarrow \frac{v_3}{R} = C(\dot{v}_1 - \dot{v}_3) \Leftrightarrow -C\dot{v}_1 + C\dot{v}_3 + \frac{v_3}{R} = 0
\end{aligned}
\tag{7-7}
$$

**Figure 7-6**. A `Resistor-Capacitor` circuit and the associated system of equation

Now we will compute the structural index of the equation system from Figure 7-7. During the first steps the bipartite graph associated to the DAE system is generated. A weight is assigned to each edge, which represents the existence of a higher order derivative in the equation tied to the edge.

$$
\begin{aligned}
& C\dot{v}_1 - C\dot{v}_3 - i = 0 \\
& -C\dot{v}_1 + C\dot{v}_3 + \frac{v_3}{R} \\
& -v_1 = v(t)
\end{aligned}
$$

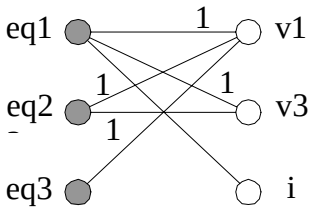



**Figure 7-7.** The system of equations for which the structural index is computed and the associated weighted bipartite graph.

According to the weight assignment rule the following relation holds since the variables $\dot{v}_1$ and $\dot{v}_3$ are present in equations *eq1* and *eq2*:

$$w(eq1, v_1) = w(eq1, v_3) = w(eq2, v_1) = w(eq2, v_3) = 1$$

In the next step the algorithm will try to find a maximum weighted perfect matching in the bipartite graph. The family of corresponding perfect matchings includes only one perfect matching: $\Im(\mathbb{M}_{1G}^3) = \{M_{1G}^3\}$, where:

$$M_{1G}^3 = \{\{eq1, eq2, eq3\}\{v_1, v_3, i\}, \{(eq1, i), (eq2, v_3), (eq3, v_1)\}\}.$$

Since there is only one perfect matching in the family of all perfect matchings there is no need to search for another maximum weighted matching. Therefore, the weight of $M_{1G}^3$ can be calculated directly:

$$w(M_{1G}^3) = \sum_{(i,j)\in M} w_{ij} = w(eq1, i) + w(eq2, v_3) + w(eq3, v_1) = 0 + 1 + 0 = 1.$$

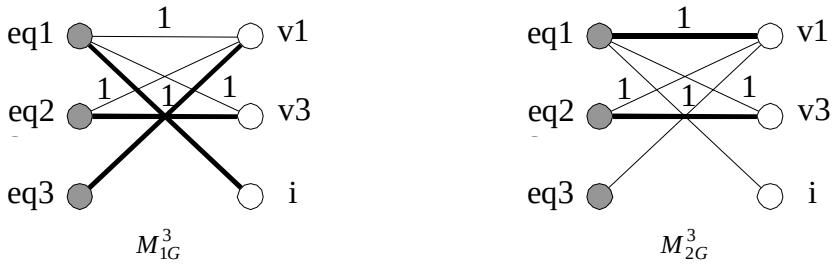



**Figure 7-8.** Maximum weighted matchings corresponding to the bipartite graph.

Comuting the family of the matchings of size 2 we obtain five different matchings in the family $\Im(\mathbb{M}_G^2) = \{M_{1G}^2, M_{2G}^2, M_{3G}^2, M_{4G}^2, M_{5G}^2\}$ where:

| | | |
|---|---|---|
| $M_{1G}^2 = \{\{eq1, eq2\}, \{v_1, v_3\}\{(eq1, v_1), (eq2, v_3)\}\}$ | | $w(M_{1G}^2) = 2$ |
| $M_{2G}^2 = \{\{eq1, eq2\}, \{v_1, v_3\}\{(eq1, v_3), (eq2, v_1)\}\}$ | with the | $w(M_{2G}^2) = 2$ |
| $M_{3G}^2 = \{\{eq2, eq3\}, \{v_1, v_3\}\{(eq2, v_3), (eq3, v_1)\}\}$ | corresponding | $w(M_{3G}^2) = 1$ |
| $M_{4G}^2 = \{\{eq1, eq3\}, \{v_1, i\}\{(eq1, i), (eq3, v_1)\}\}$ | weights | $w(M_{4G}^2) = 0$ |
| $M_{5G}^2 = \{\{eq1, eq3\}, \{v_1, v_3\}\{(eq1, v_3), (eq3, v_1)\}\}$ | | $w(M_{5G}^2) = 1$ |

We should note that the two matchings $M_{1G}^2$, $M_{2G}^2$ respectively have their maximum weight equal to 2.

Applying the structural index formula we obtain a structural index equal to 2 where the underlying DAE system of equations clearly has a differential index equal to 1.

$$i_{str}(G) = \max_{M_G^2 \in \Im(\mathbb{M}_G^2)} w(M_G^2) - \max_{M_G^3 \in \Im(\mathbb{M}_G^3)} w(M_G^3) + 1 = w(M_{1G}^2) - w(M_{1G}^3) = 2 - 1 + 1 = 2$$

If we choose node 1 or node 3 as the ground node, both the differential index and the structural index will be equal to 1. In Figure 7-9 we illustrate the associated bipartite graph and the two maximum weighted matchings for the same electrical circuit presented above but this time the node 3 is chosen to be ground node.

$$C\dot{v}_1 - i = 0$$
$$C\dot{v}_1 - \frac{v_2}{R} = 0 \quad (6.9)$$
$$v_1 - v_2 = v(t)$$

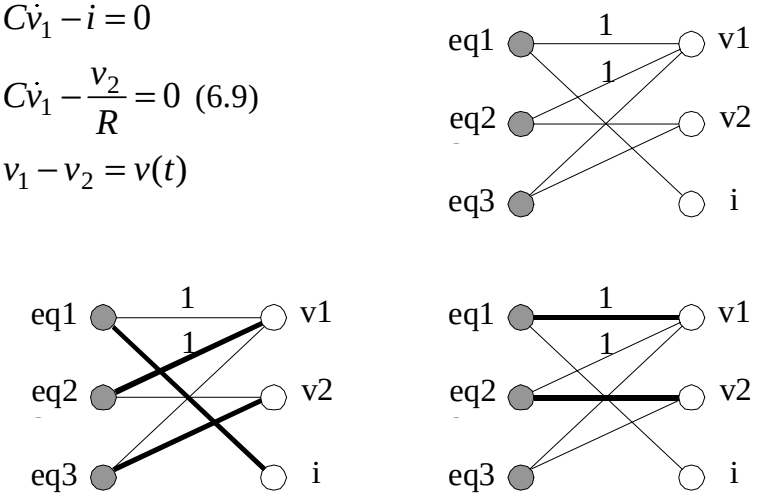



**Figure 7-9.** Electrical circuit with ground located in node 3.

### 7.4.2 The "Embarrassing Phenomenon" Revisited

Let us consider the simple electrical circuit presented in (Murota 2000 [87]) and (Cellier 1991 [24]) This circuit consists of two resistors `R1` and `R2`, a voltage source `VA` an inductor `L` and a capacitor `C`, connected together as illustrated in Figure 7-10. By analyzing the example from Figure 7-10, we are going to demonstrate that the graph theoretical structural approach for computing the structural index of the physical circuit model is only valid to a certain extent.

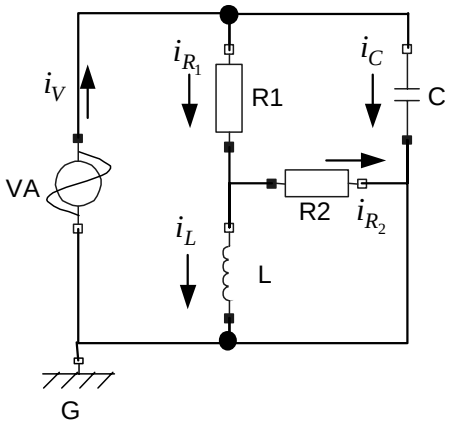
The equations generated by the component connections describe the relations among the voltages and currents respectively of the interconnected electrical components when applying the two laws of Kirchhoff. *Kirchhoff 's Current Law* states that the sum of the currents along any cut set of the network graph is zero. *Kirchhoff 's Voltage Law* states that the sum of the voltages along any circuit of the network graph is zero.

Considering the equations representing the conservation of the voltage along the loops R1-R2-C, R2-L and VA-C we obtain the directed bipartite graph as presented in Figure 7-11 a). The edges $(eq10, \text{var}10)$ and $(eq9, \text{var}4)$ are marked with thicker lines because they represent the presence of a differentiated variable in an equation and therefore will be weighted equal to 1. It should be noted that the edge $(eq9, \text{var}4)$ is also a matching edge. The edge $(eq10, \text{var}10)$ can never be part of a perfect matching corresponding to the bipartite graph because it is not included in any cycle. When calculating the weight of the perfect matching we obtain 1 which is also a maximal weight for the family of perfect matchings since no other perfect matching include more

for the family of perfect matchings since no other perfect matching include more weighted edges. The path

$$\{(eq1, \text{var}\,5), \overline{(\text{var}\,5, eq10)}, (eq10, \text{var}\,10), \overline{(\text{var}\,10, eq3)}, (eq3, \text{var}\,6), \overline{(\text{var}\,6, eq6)}\}$$

is a feasible path. Therefore we can exchange the matching edges with non-matching edges and obtain another matching with a lower cardinality.




$$eq1 \quad i_V - i_{R_1} - i_C = 0$$
$$eq2 \quad -i_{R_1} + i_{R_2} + i_L + i_C = 0$$
$$eq3 \quad -V_V - V_C = 0$$
$$eq4 \quad V_{R_1} + V_{R_2} - V_C = 0$$
$$eq5 \quad -V_{R_2} + V_L = 0$$
$$eq6 \quad -V_V = 0$$
$$eq7 \quad R_1 i_{R_1} - V_{R_1} = 0$$
$$eq8 \quad R_2 i_{R_2} - V_{R_2} = 0$$
$$eq9 \quad \frac{di_L}{dt} L - V_L = 0$$
$$eq10 \quad -i_c + C\frac{dV_c}{dt} = 0$$

$$\text{var}\,1 \quad i_V$$
$$\text{var}\,2 \quad i_{R_1}$$
$$\text{var}\,3 \quad i_{R_2}$$
$$\text{var}\,4 \quad i_C$$
$$\text{var}\,5 \quad i_L$$
$$\text{var}\,6 \quad V_V$$
$$\text{var}\,7 \quad V_{R_1}$$
$$\text{var}\,8 \quad V_{R_2}$$
$$\text{var}\,9 \quad V_L$$
$$\text{var}\,10 \quad V_C$$

**Figure 7-10.** A simple electrical circuit model together with corresponding equations and variables.

After exchanging the edges we obtain the following path:

$$\{\overline{(eq1, \text{var}\,5)}, (\text{var}\,5, eq10), \overline{(eq10, \text{var}\,10)}, (\text{var}\,10, eq3), \overline{(eq3, \text{var}\,6)}, (\text{var}\,6, eq6)\}\,.$$

This time, the edge $(eq10, \text{var}\,10)$ is a matching edge and therefore will add its weight when computing the weight of the corresponding matching. Calculating the weight of the matching of size 9 obtained by exchanging matching edges from the above mentioned path we obtain the value 2, which is also a maximal value because it includes both weighted matchings. Therefore there is no need to iterate to all the possible matchings of size 9. Now we can easily calculate the structural index of the equation system:

$$i_{str}(G) = \max_{M_G^9 \in \Im(\mathbb{M}_G^9)} w(M_G^9) - \max_{M_G^{10} \in \Im(\mathbb{M}_G^{10})} w(M_G^{10}) + 1 = 2 - 1 + 1 = 2$$

which is equal to the differential index.

Now we describe the circuit from Figure 7-10 considering the equations representing the conservation of the voltage along the loop VA-R1-L instead of the loop VA-C. The corresponding directed bipartite graph with an associated perfect matching is presented in Figure 7-11 b). It should be noted that we can easily find a perfect matching which includes both weighted edges because edge $(eq9, \text{var}\,4)$ is already included in the perfect matching and the edge $(eq10, \text{var}\,5)$ is included in an alternating cycle

$\{eq2, \text{var}3), (\overline{\text{var}3, eq8}), (eq8, \text{var}8), (\overline{\text{var}8, eq4}), (eq4, \text{var}10), (\overline{\text{var}10, eq10}),$
$(eq10, \text{var}5)(\text{var}5, eq2)\}$

where we can easily exchange the matching edges with non-matching edges and obtain another matching with the same cardinality which includes the weighted edge. Computing the structural index:

$i_{str}(G) = \max_{M_G^9 \in \Im(\mathbb{M}_G^9)} w(M_G^9) - \max_{M_G^{10} \in \Im(\mathbb{M}_G^{10})} w(M_G^{10}) + 1 = 2 - 2 + 1 = 1$ we obtain a value which is less than the corresponding differential index associated with the DAE system.

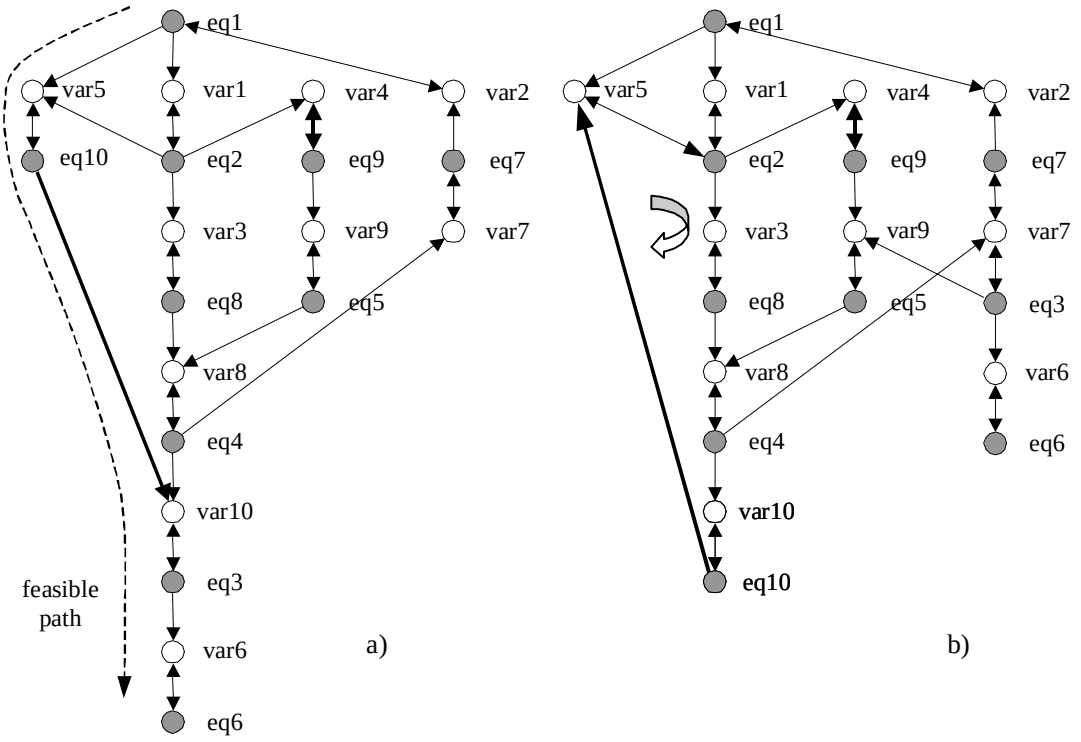



**Figure 7-11. a)** The directed bipartite graph for the simple electrical circuit when the Kirchoff's Voltage Laws along `R2-C`, `R2-L` and `VA-C` were considered. **b)** The corresponding directed bipartite graph for the simple electrical circuit when the Kirchoff's Voltage Laws along `R2-C`, `R2-L` and `VA-R1-L` were considered.

Different views of the same problem can lead to different models. Empirical comparison might be worthwhile. This phenomenon is mentioned in the literature as an “embarrassing” phenomenon (Murota 2000 [87]) and several practical examples were identified such as the above mentioned electrical circuit (Murota 2000 [87]) or analysis problems arising from distillation columns in chemical engineering (Unger et. al. 1995 [118]). This situation is unlikely to appear when the circuit model is specified with the help of a modeling language such as Modelica, where the same equation system is obtained from the same model topology.

## 7.5 Conclusions Regarding the Structural Index

The structural index is not uniquely determined for a physical engineering system, as has been demonstrated by several examples. It depends on the system description and therefore is dependent on modeling choices made by the user. However, structural considerations and the computation of the structural index provides a meaningful and computationally efficient method to characterize large and/or highly nonlinear DAEs where numerical and symbolic methods are extremely expensive or are not even possible.

The first structural algorithm for computing the structural index is due to (Duff and Gear 1986 [31]). This algorithm answers the question whether the index of DAE system exceeds the value of two. For certain problems the algorithm exhibits non-polynomial computational complexity. Fortunately Algorithm 7-1 has a wider applicability since it can be applied to DAE systems of any index and does not require that the equations should be written in a special form. For this reason, the computation of a structural index associated with DAE system constitutes the first step in reducing the index of higher index problems (Pantelides 1988 [93]) as will be presented in the next section.

It should also be noted that the algorithm for computing the structural index can easily be integrated into a general debugging and compilation framework of equation-based languages. The perfect matching of the bipartite graph associated to a DAE system of equations was already computed by the Dulmage-Mendehlson canonical decomposition algorithm and can be reused here. Furthermore the family of all perfect matchings is also computed as a preliminary step for computing the strongly connected components of the associated bipartite graph. Only the computation of the family of lower cardinality matchings introduce an extra computational burden.

## 7.6 Index Reduction Algorithms

The term high-index DAE is commonly used to refer to equation systems where the differential index $i_d \geq 2$ (Brenan et. al. 1989 [22]) Standard numerical methods such as BDF or implicit Runge-Kutta applied to some form of higher-index DAE experience difficulties and need to be adapted to special categories of problems. There are only a few higher-index solvers and only special kinds of higher-index problems may be solved directly at present. Therefore index reduction techniques are necessary to reduce higher-index problems to index 1 problems, which can successfully be solved with general purpose numerical DAE solvers such as DASSL (Differential Algebraic System Solver) (Petzold 1982 [96]).

### 7.6.1 A Higher Index Problem from Two Rigidly Connected Masses

In the following we will try to analyze a modeling example of two rigidly connected masses where the index of the underlying DAE system is greater than 1. Then Pantelides' index reduction algorithm is applied to the simulation model in order to

show the transformation performed on equations during the optimization phase of the compiler.

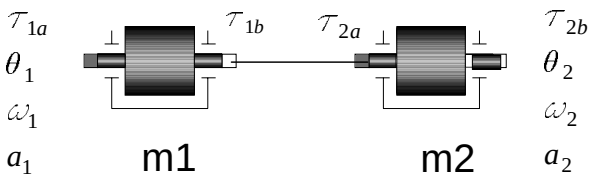



**Figure 7-12.** Two rigidly connected masses

The corresponding Modelica source code is given below:

```
connector Flange_a "1D rotational flange "
  Real phi "Absolute rotation angle of flange";
  flow Real tau "Cut torque in the flange";
end Flange_a;

connector Flange_b "1D rotational flange "
  Real phi "Absolute rotation angle of flange";
  flow Real tau "Cut torque in the flange";
end Flange_b;

partial model Rigid
    "Rigid connection of 2 rotational flanges"
  Real phi "Absolute rotation angle of component
  Flange_a flange_a "(left) driving flange";
  Flange_b flange_b "(right) driven flange";
equation
  flange_a.phi = phi;
  flange_b.phi = phi;
end Rigid;

model Inertia "1D-rotational component with inertia"
  extends Rigid;
  parameter Real J=1 "Moment of inertia";
  Real w "Absolute angular velocity of component";
  Real a "Absolute angular acceleration of component";
equation
  w = der(phi);
  a = der(w);
  J*a = flange_a.tau + flange_b.tau;
end Inertia;

model RigidlyConnectedMasses
   Inertia m1;Inertia m2;
equation
   connect(m1.flange_b,m2.flange_a);
end RigidlyConnectedMasses;
```

When translating the `RigidlyConnectedMasses` model we obtain the following set of flattened equations and variables (shown in Table 7-1).

**Table 7-1.** The flattened set of equations and variables at the intermediate code level corresponding to the rigidly connected masses model.

| | | | |
|---|---|---|---|
| ***eq1*** | `m1.flange_a.phi = m1.phi` | ***var1*** | `m1.phi` |
| ***eq2*** | `m1.flange_b.phi = m1.phi` | ***var2*** | `m1.flange_a.phi` |
| ***eq3*** | `m1.w = (m1.phi)'` | ***var3*** | `m1.flange_a.tau` |
| ***eq4*** | `m1.a = (m1.w)'` | ***var4*** | `m1.flange_b.phi` |
| ***eq5*** | `m1.a * m1.J= m1.flange_a.tau + m1.flange_b.tau` | ***var5*** | `m1.flange_b.tau` |
| ***eq6*** | `m2.flange_a.phi = m2.phi` | ***var6*** | `m1.w` |
| ***eq7*** | `m2.flange_b.phi = m2.phi` | ***var7*** | `m1.a` |
| ***eq8*** | `m2.w = (m2.phi)'` | ***var8*** | `m2.phi` |
| ***eq9*** | `m2.a = (m2.w)'` | ***var9*** | `m2.flange_a.phi` |
| ***eq10*** | `m2.a m2.J = m2.flange_a.tau + m2.flange_b.tau` | ***var10*** | `m2.flange_a.tau` |
| ***eq11*** | `m1.flange_b.phi = m2.flange_a.phi` | ***var11*** | `m2.flange_b.phi` |
| ***eq12*** | `m1.flange_b.tau + m2.flange_a.tau = 0` | ***var12*** | `m2.flange_b.tau` |
| ***eq13*** | `m1.flange_a.tau = 0` | ***var13*** | `m2.w` |
| ***eq14*** | `m2.flange_b.tau = 0` | ***var14*** | `m2.a` |

By transforming the flattened equations into a corresponding bipartite graph representation and applying a strongly connected component algorithm on the directed graph we obtain the following block lower triangular form of the equations:

```
Strongly connected components: 5
STRONGLY CONNECTED COMPONENTS
[0] m1.flange_a.tau = 0        m1.flange_a.tau
[1] m2.flange_b.tau = 0       m2.flange_b.tau
[2] m1.flange_b.phi = m1.phi
    m1.w = (m1.phi)'
    m1.a = (m1.w)'
    m1.a * m1.J = m1.flange_a.tau + m1.flange_b.tau
    m2.flange_a.phi = m2.phi
    m2.w = (m2.phi)'
    m2.a = (m2.w)'
    m2.a m2.J = m2.flange_a.tau + m2.flange_b.tau
    m1.flange_b.phi = m2.flange_a.phi
    m1.flange_b.tau + m2.flange_a.tau = 0

      m1.phi m1.flange_b.phi m1.flange_b.tau m1.w m1.a m2.phi
      m2.flange_a.phi m2.flange_a.tau m2.w m2.a
[3] m1.flange_a.phi = m1.phi  m1.flange_a.phi
[4] m2.flange_b.phi = m2.phi  m2.flange_b.phi
```

**Figure 7-13.** Presentation of the strongly connected components corresponding to the model of rigidly connected masses.

We discard blocks [0][1] and [3][4] which only contain one equation each, have index 0, and do not influence the index of the total system. We concentrate our analysis only on block [2] on which an index reduction algorithm needs to be applied. During the first step we need to compute the structural index of the system of equations. Applying Algorithm 7-1 on the weighted bipartite matching we obtain the structural index value 3.

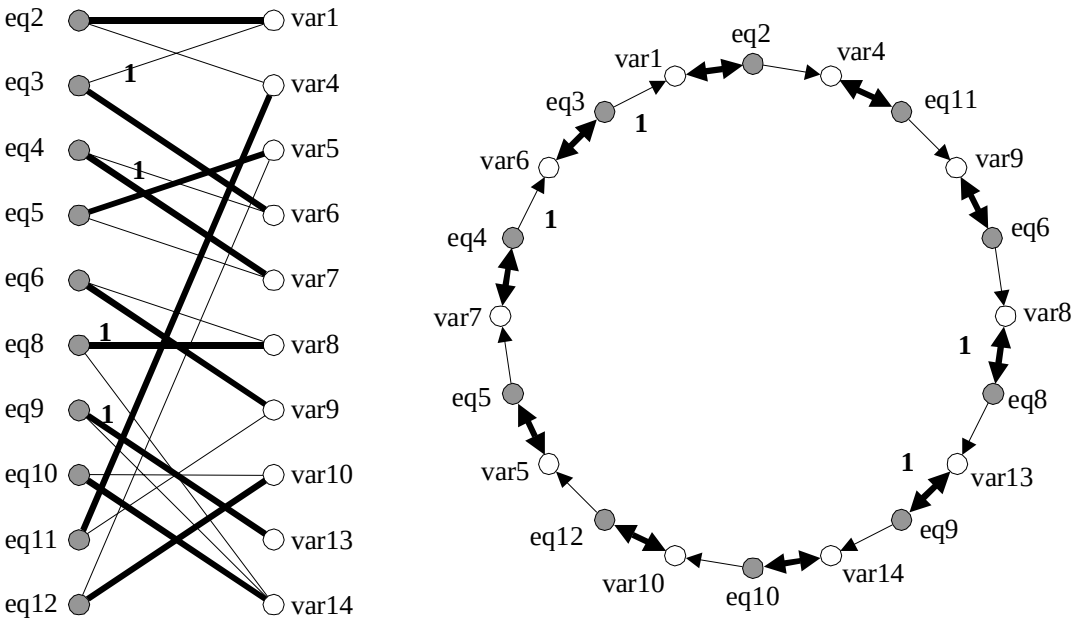



**Figure 7-14.** The bipartite graph and the weighted directed bipartite graph corresponding to block [2] of the rigidly connected masses.

Obviously, being a strongly connected component, block [2] obtained after performing the block lower triangular form decomposition is a cycle as shown in the right part of Figure 7-14. Before performing the structural index calculation, in order to limit our analysis and make the example more illustrative we perform some symbolic variable substitutions. Simple equality expressions such as $x_i = x_j$ can be recognized and eliminated from analyzed equations as already was illustrated in Chapter 3. In our example the variable `m1.flange_a.phi` from the equation `m1.flange_a.phi = m1.phi` can be substituted by the variable `m1.phi` in all the equations in which it appears. Applying this substitution, equation `m1.flange_b.phi = m2.flange_a.phi` becomes `m1.phi = m2.flange_a.phi`. Now we can repeat the same process for the variable `m2.flange_a.phi` and substitute it in equation `m2.flange_a.phi = m2.phi` that becomes `m1.phi = m2.phi` as illustrated below:

```
m1.flange_b.phi = m1.phi
m1.flange_b.phi = m2.flange_a.phi
m2.flange_a.phi = m2.phi

m1.flange_b.phi = m1.phi
m1.phi = m2.flange_a.phi
m2.flange_a.phi = m2.phi

m1.flange_b.phi = m1.phi
m1.phi = m2.flange_a.phi
m1.phi = m2.phi
```

In conclusion: *eq6* (`m2.flange_a.phi = m2.phi`) is transformed into *eq6'* (`m1.phi = m2.phi`). The variables present in equations *eq2* and *eq11* can be computed later based on the value of the variable *var1*. The same symbolic substitution of variables can be done for variables *var14*, *var10*, *var5*, and *var7* and transform the equation *eq4*: m1.a = (m1.w)' into the equation *eq4'*: m2.a = (m1.w)' by substituting `m1.J = 1,`

`m1.flange_a.tau = 0` from solving the previous block and from `m1.flange_b.tau = m2.a` and `m1.a = m2.a`.

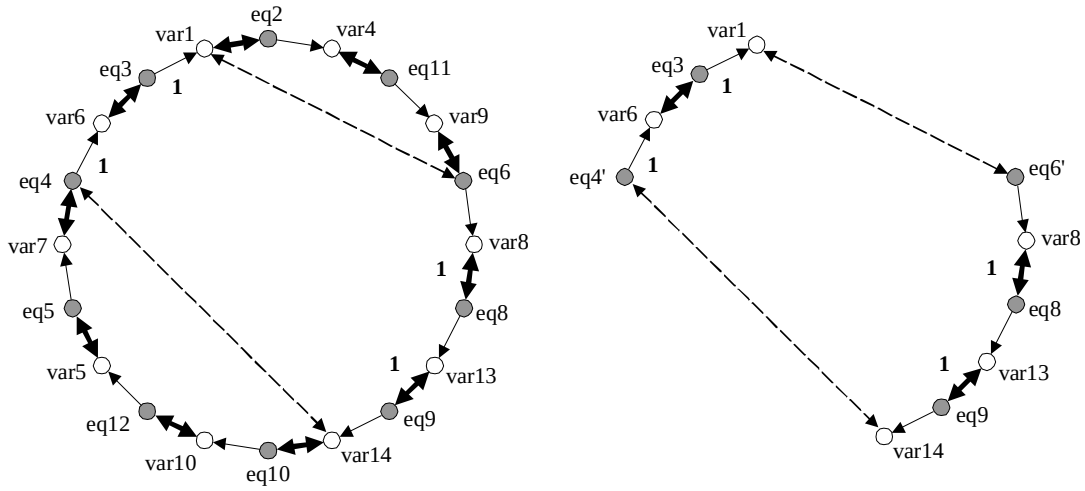



**Figure 7-15.** Symbolic substitution of the variables.

After the symbolic substitutions of the variables involved in simple variable equality equations we obtain the reduced system of equations from block [2] that need to be solved:

```
eq3          m1.w = (m1.phi)'
eq4'         m2.a = (m1.w)'
eq8          m2.w = (m2.phi)'                    (7-8)
eq4          m2.a = (m2.w)'
eq6'         m1.phi = m2.phi
```

## 7.7 Index Reduction Algorithm

In this section we perform Pantelides' index reduction algorithm (Pantelides 1988 [93]) on the set of equation (7-8) resulting from the symbolic simplification of block [2] from the BLT form. For readability purposes we rewrite the equation system expressed in the intermediate flattened code form into a more mathematical notation. We are also relabel the equations as follows:

| | | | |
|---|---|---|---|
| *eq3* | `m1.w = (m1.phi)'` | eq1 | $\omega_1 = \dot{\Phi}_1$ |
| *eq4'* | `m2.a = (m1.w)'` | eq2 | $a_2 = \dot{\omega}_1$ |
| *eq8* | `m2.w = (m2.phi)'` | eq3 | $\omega_2 = \dot{\Phi}_2$ |
| *eq4* | `m2.a = (m2.w)'` | eq4 | $a_2 = \dot{\omega}_2$ |
| *eq6'* | `m1.phi = m2.phi` | eq5 | $\Phi_1 = \Phi_2$ |

(7-9)

The first step in Pantelides' algorithm is to compute the structural index based on the weighted matching of the directed graph. Therefore we construct the weighted bipartite graph

corresponding to the system of differential algebraic equations from (7-9) (see the left part of Figure 7-16.

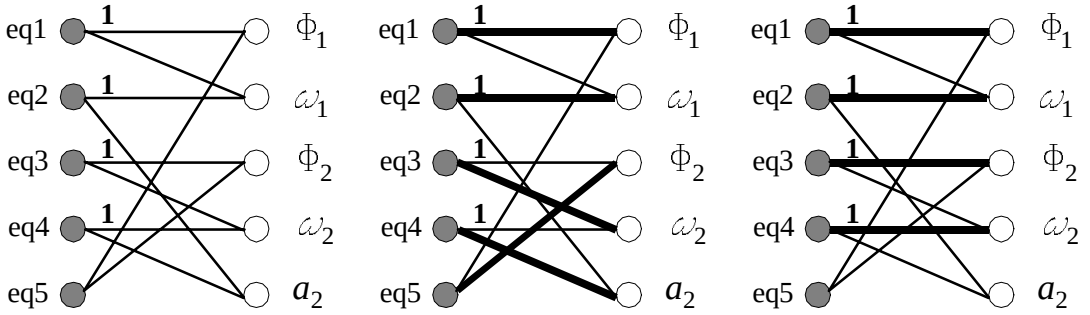



**Figure 7-16.** Corresponding weighted bipartite graph, perfect matching and n-1 size maximum cardinality matching corresponding to the system of equation (7-9).

The bipartite graph shown in the middle of Figure 7-16 represents a possible perfect matching. For computing the structural index we need to find a maximum weighted perfect matching. The total weight of the chosen perfect matching is $w(M_{1G}^5) = 2$. In order to be sure that no other perfect matching associated with the bipartite graph has a higher total weight, we construct the directed bipartite graph corresponding to the graph, together with all possible perfect matchings, see Figure 7-17.

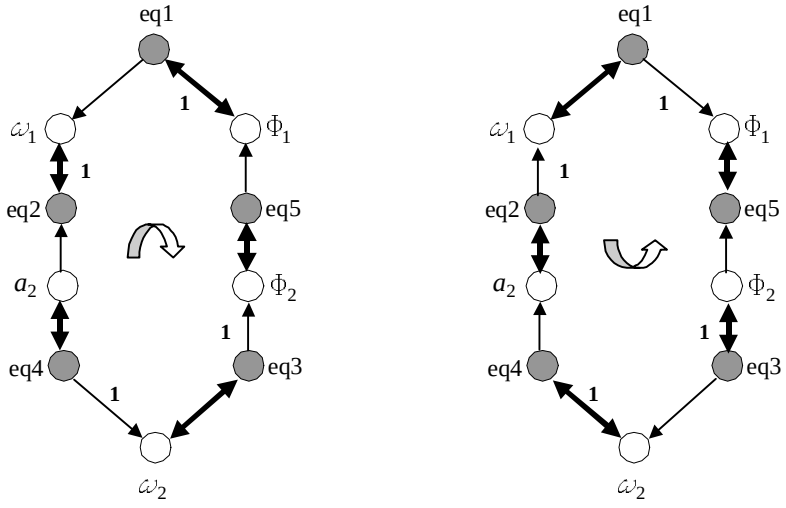



**Figure 7-17.** All possible perfect matching of the bipartite graph associated to the system of equations (7-9).

The other possible perfect matching present for the system also has its total weight equal to 2, $w(M_{2G}^5) = 2$, as illustrated in the right part of Figure 7-17. The maximum weight *n-1* cardinality matching has a weight equal to 4, which is illustrated in in the right part of Figure 7-16. We apply Algorithm 7-1 to the weighted bipartite graph to compute the structural index $i_{str}(G) = maxM_G^{n-1} - maxM_G^n + 1 = 4 - 2 + 1 = 3$.

In order to reduce the computed structural index the total weight of the perfect matching need to be increased without increasing the weight of the lower cardinality (n-

1) matching. This can be achieved if we can assign a weight equal to 1 to the edges $(eq5, \Phi_1)$ or $(eq5, \Phi_2)$, which is equivalent to differentiation of *eq5*. Differentiating *eq5*, the maximum weighted perfect matching and the maximum weighted lower cardinality matching are recomputed based on the updated weighted bipartite graphs as shown in Figure 7-18.

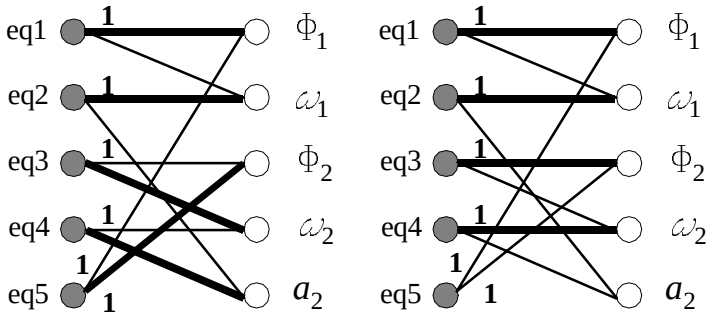



**Figure 7-18.** The updated weighted bipartite graph with the associated maximum weighted perfect matching and the maximum weighted lower cardinality matching obtained after differentiating *eq5*.

The new structural index becomes $i_{str}(G) = maxM_G^{n-1} - maxM_G^n + 1 = 4 - 3 + 1 = 2$. Differentiating *eq5* one more time the structural index can further be reduced. A weight equal to 2 will be assigned to the edges $(eq5, \Phi_1)$ and $(eq5, \Phi_2)$. The structural index is recomputed once more $i_{str}(G) = maxM_G^{n-1} - maxM_G^n + 1 = 4 - 4 + 1 = 1$ and is equal to 1, which means that no further differentiation is necessary. By differentiating *eq5* twice the system of equations from (7-9) becomes:

```
m1.w = (m1.phi)'
m2.a = (m1.w)'
m2.w = (m2.phi)'
m2.a = (m2.w)'                    (7-10)
m1.phi = m2.phi
m1.phi' = m2.phi'
m1.phi'' = m2.phi''
```

Equation (7-10) contains 7 equations and 9 variables (`m2.a, m2.w, m1.w, m1.phi, (m1.phi)', (m1.phi)'', m2.phi, (m2.phi)', (m2.phi)''`). Thus two variables out of the variable set can be given an arbitrary initial value in order to make the system well-constrained.

## 7.8 Conclusions

We have presented a graph theoretic approach based on Pantelides' algorithm (Pantelides 1988 [93]). This algorithm is used to find consistent initial values for high index DAEs. The algorithm can also be used for symbolically reducing the index of a DAE. The main conclusions of this chapter is that the abstractions used for symboli-

cally reducing the index and providing consistent initial values for DAEs to a very large extent are similar to the abstractions used for debugging equation-based languages and presented in the previous chapters. Therefore the compiler and the debugger can easily be built based on the same graph theoretical abstraction. Moreover, the same abstraction can be used to specify the three main components of the structural analysis of DAEs:

- Detecting over-constraining equations.
- Detecting under-constraining equations.
- Consistent initialization of DAEs.

The last component is already provided by any equation-based language compiler that employs DAEs. Therefore it is important for the two components of a debugger that handles over- and under-constraining equations to be tightly integrated with the compiler.

# Chapter 8

# Debugging Environments for Declarative Equation-Based Languages

***Summary****: 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 automatic or semi-automatic debugging tools. Another important factor that must to be taken into account is the integration of the debugger into an existing language compiler. To support effective debuggers, static analyzers need to be added to the existing simulation environments that employ equation-based languages. In this chapter, we outline the architecture and organization of the two implemented debugging tools. One of the tools has been attached to the MathModelica simulation environments that employs the Modelica language.*

## 8.1 Overview of the Algorithmic Automatic Debugging Framework

At this stage we are able to provide an overview of the proposed framework developed for the Modelica language and Modelica-based simulation environments. Even if we have limited our prototype implementations to the Modelica language, the developed debugging kernels can easily be adapted to handle other object-oriented equation-based languages. In the present work we have tried to deal only with the static aspects of the debugging process, namely the detection and correction of over and under-constrained error situations. The next step should be to improve the already developed techniques by providing better user interaction and extending the debugger to handle dynamic error situations as well, e.g. to support locating and fixing numerical failures and inconsistencies. It is important to note that the proposed debugging framework can easily be integrated into the existing Modelica compilers.

In the following subsections details about our two prototype debugging systems will be presented:

- The MathModelica debugging kernel prototype.
- The AMOEBA debugging kernel for the Open Source Modelica System.

## 8.2 MathModelica Based Debugging Kernel

Our first prototype debugger was built and attached to the *MathModelica* simulation environment as a testbed for evaluating the usability of the previously presented graph decomposition techniques for debugging declarative equation-based languages. The implemented debugger is just a research prototype and is not part of the *MathModelica* commercial product.

*MathModelica* is an integrated problem-solving environment (PSE) for full system modeling and simulation (Jirstrand 2000 [68]) (Jirstrand et. al 1999 [69]) (Fritzson et. al. 2002 [45]). The environment integrates Modelica-based modeling and simulation with graphic design, advanced scripting facilities, integration of code and documentation, and symbolic formula manipulation provided via *Mathematica* (Wolfram 1996 [126]). Import and export of Modelica code between internal structured and external textual representation is supported by *MathModelica*. The environment extensively supports the principle of literate programming and integrates most activities needed in simulation design: modeling, documentation, symbolic processing, transformation and formula manipulation, input and output data visualization.

The model corrections presented by the debugger will of course lead to a mathematically sound system of equations. However some of the solutions might not be acceptable from the modeling language point of view or from the physical system model perspective. The debugger focuses on those static model errors whose identification does not require the solution of the underlying system of equations.

As indicated previously, it is necessary for the compiler to annotate the underlying equations to help identify the equations at the intermediate code level and to help the user to choose the right solution. Accordingly, we have modified the front end of the compiler to annotate the intermediate representation of the source code where equations are involved. The annotations are propagated appropriately through the various phases of the compiler. When an error is detected, the debugger uses the annotations to eliminate some of the error fixing solutions and to identify the problematic equations. Then, error messages, consistent with the user perception of the source code of the simulation model, are generated automatically.

The implemented debugger has been successfully tested on declarative models involving several hundred differential algebraic equations.

The general architecture of the implemented debugger attached to the *MathModelica* environment is presented in Figure 8-1. The debugging algorithm proceeds as follows: based on the original object-oriented equation-based source code an intermediate representation is generated. From the intermediate representation the overall system of equations is extracted and transformed into bipartite graph form. The associated bipartite graph is canonically decomposed and error-fixing strategies are applied if the decomposition leads to over- or under-constrained components. The debugger will try to solve the errors automatically without the explicit intervention of the user. If automatic error fixing is not possible due to missing information, the user will be consulted regarding

the repair strategy. When the user is interrogated, all the valid options that will lead to a numerically sound system of equations are presented.

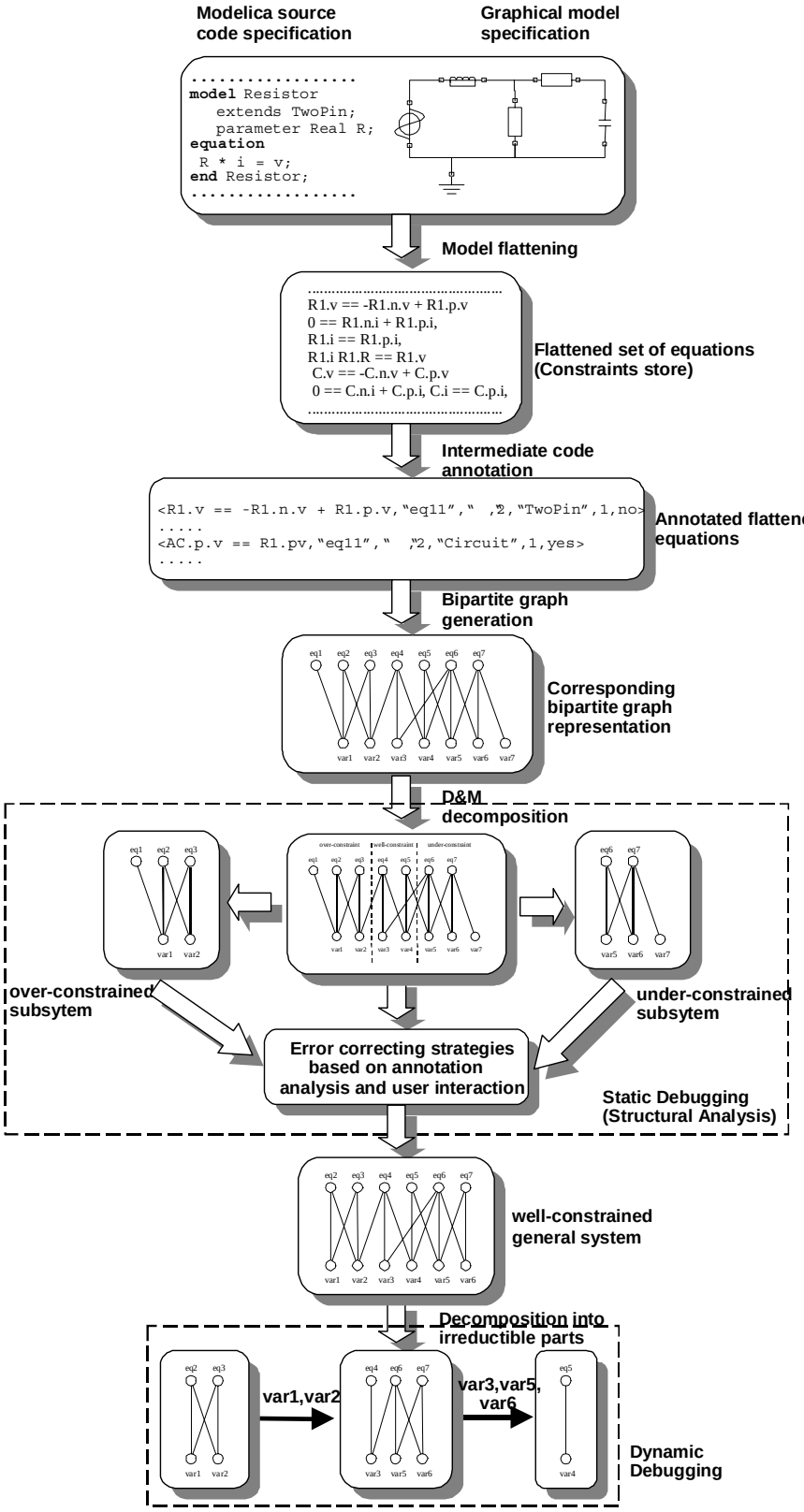



**Figure 8-1.** The debugging framework

The main advantage of using the *MathModelica* debugger is that the intermediate flattened form of the equations can be visualized together with the corresponding bipartite graph representation as shown in Figure 8-2. Of course, this facility is useful only for small simulation models. However, in order to address different categories of users, the *MathModelica* debugging kernel has been designed to allow multiple debugger windows. Each of these windows is associated to the original file that contains the model that needs to be debugged. One debugging window similar to that shown in Figure 8-2 can be generated with detailed information regarding the translation process, the intermediate flattened equations from, the bipartite graph representation together with the associated matchings, and the description of the steps performed by the debugger.

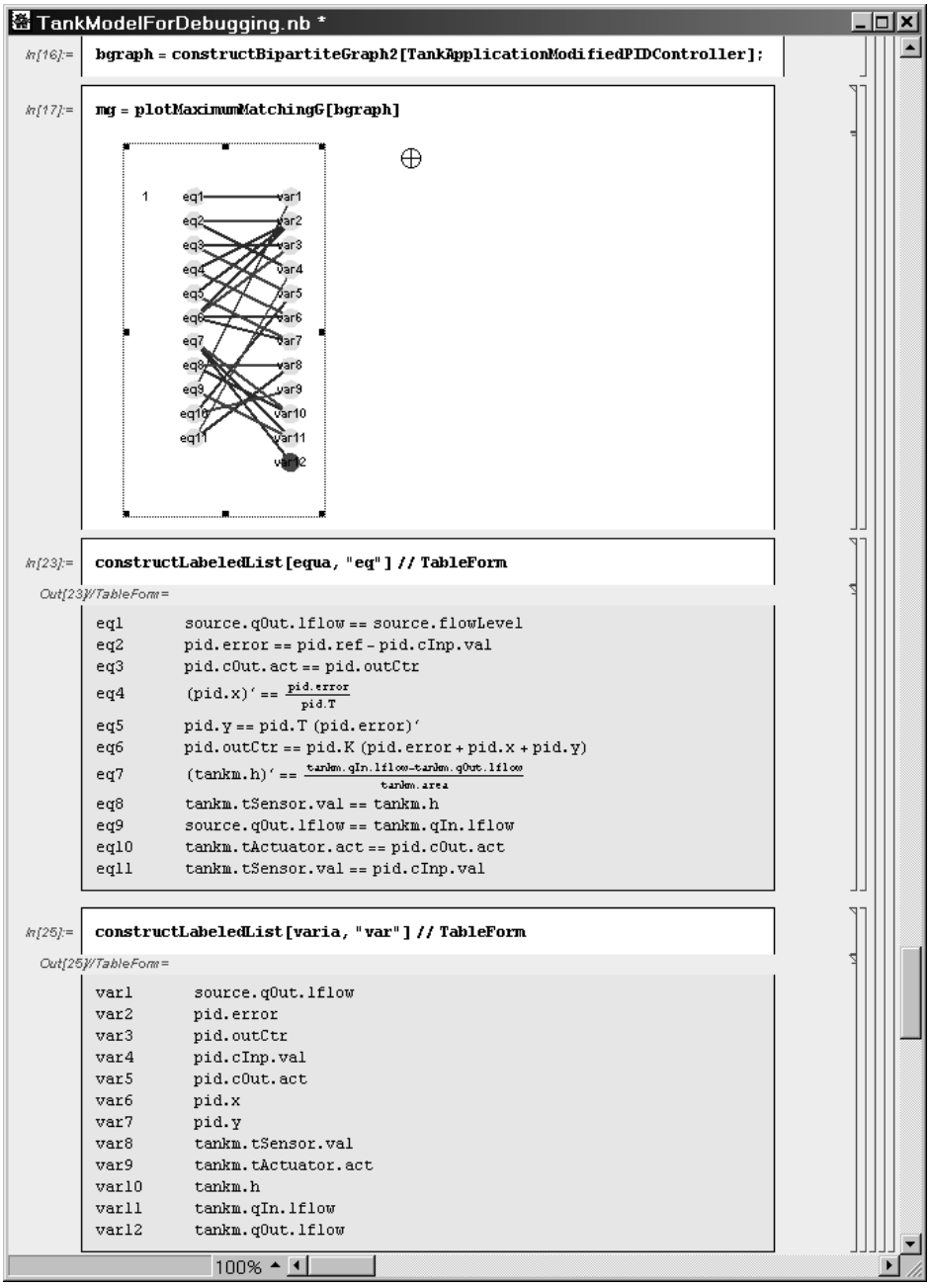



**Figure 8-2.** Screen shot of the MathModelica debugging kernel visualization of the intermediate flattened form of the equations and the corresponding bipartite graphs.

The second debugging window provides error-fixing solutions similar to those presented in Chapter 6, Figure 6-14 and Figure 6-15. Based on the information provided in this window the necessary modifications can be made in the original electronic document that contains the simulation source code.

## 8.3 AMOEBA – Automatic MOdelica Equation Based Analyzer

AMOEBA (Automatic MOdelica Equation-Based Analyzer) is the second debugging kernel that we have designed and implemented in order to attach it to the Open Source Modelica Environment. The tool is able to successfully detect and provide error-fixing solutions for typical over and under-constrained situations, which might appear during the modeling stage using Modelica. The debugging kernel is in a prototype stage and is intended to be integrated fully with the Open Source Modelica compiler (Fritzson et. al. 2002 [44]), (Aronsson et. al. 2002 [6]). At this stage the flattened form of the equations are generated from the *MathModelica* environment.

Below we present each phase of the debugging process with the corresponding module.

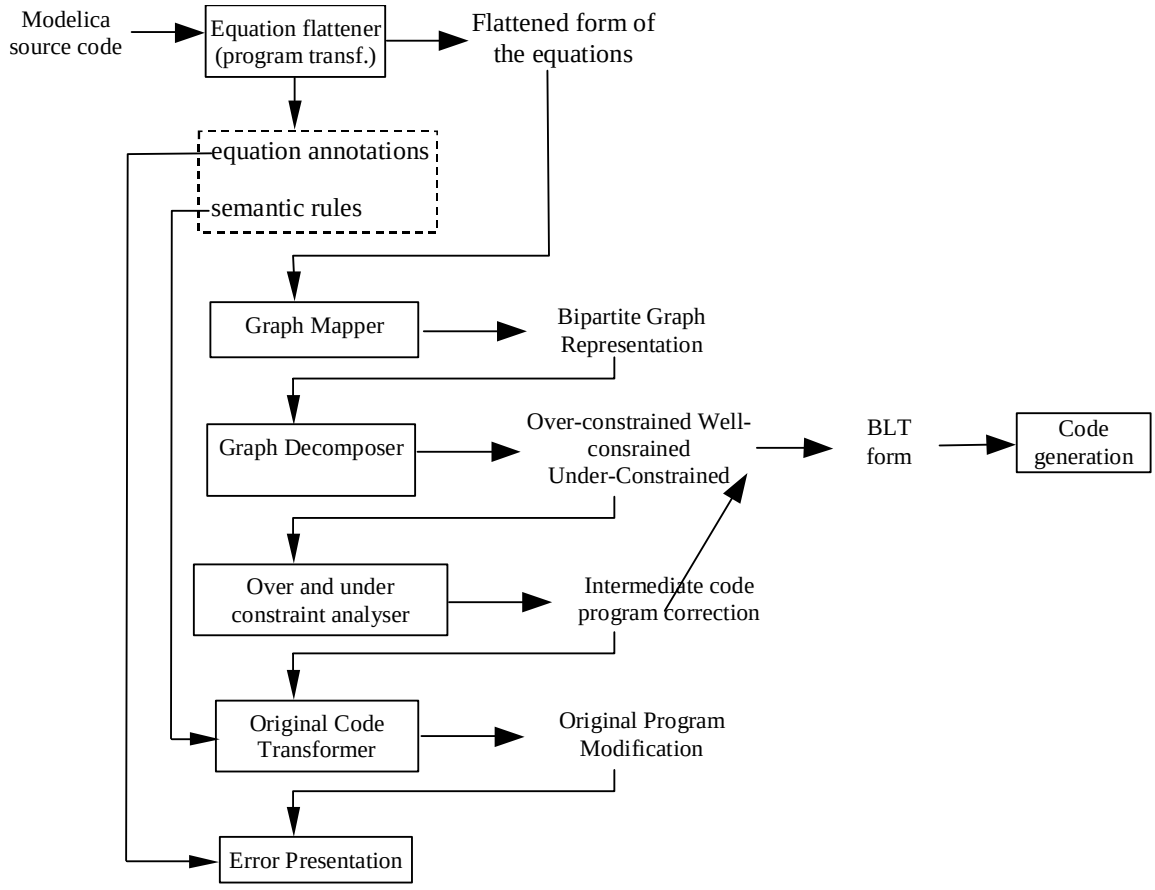



**Figure 8-3.** Overview of the debugging kernel architecture.

AMOEBA has been implemented in C++ using LEDA (Mehlhorn and Näher 1999 [83]), a library of efficient data types and algorithms for graphs and combinatorial computing. As output and presented results, AMOEBA produces error fixing messages for erroneous models and a block lower triangular form for models that compile correctly and do not contain any over- or under-constrained parts. However, the code generation module has not been implemented yet and constitutes our next obvious step for the open source Modelica compiler in order to be able to simulate corrected models.

### 8.3.1 Equation Flattener

The first stage when analyzing Modelica programs is to flatten the model classes into a system of equations without object-oriented structure in order to obtain the overall system of equations that needs to be solved. The *Equation Flattener* accepts as input the equation-based model code and outputs a set of flattened equations and variables. For debugging purposes we have enhanced this module, by adding a set of annotations to each flattened equation. The annotations help the debugger to later prioritize the error fixing solutions. In this way, for example, the flexibility level attributes to the equations can be provided. Furthermore, many of the existing library models have been well tested in different simulation configurations, validated, and can be trusted. Tested library components are usually assigned a low flexibility level. They can even be locked to prevent modification. The debugger will later automatically eliminate all those error fixing solutions that require modification of the behavior of such components. The correspondence between the original source code statements and the statements in the intermediate code is also preserved by the equation flattener.

### 8.3.2 Graph Mapping

The flattened equations are transformed into the bipartite graph representation by a *Graph Mapping* module. Modeling the intermediate flattened code with the help of bipartite graphs gives a good insight into the structure of the equations. Even information about the solvability or insolvability of the equation system can be derived as has already been demonstrated in the previous chapters.

For testing purposes the *Graph Mapping* module can accept as input a general graph file format such as GML (Himsolt 1996 [59]), (Himsolt 1997 [58]). GML supports attaching arbitrary information to graphs, nodes and edges. Therefore equation annotations can easily be specified. The *Graph Mapping* module can also save the flattened system of equations in an external file in GML format. At this level the user has the possibility of accessing and visualizing all the three representations of the flattened system of equations (flattened form, bipartite graph form, and GML form.). In Figure 8-4 below we show the three different representations of the overall system of equations at the intermediate code level.

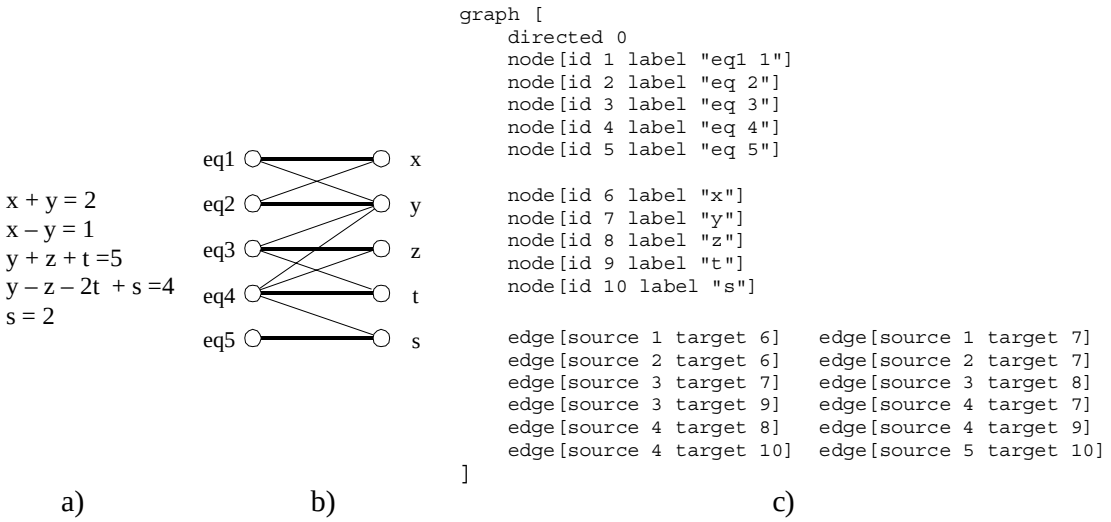



**Figure 8-4.** Various representations of the system of equations at the *Graph Mapping* module level: **a)** flattened form **b)** bipartite graph form **c)** GML format

### 8.3.3 Graph Decomposer

The canonical decomposition algorithm applied by the *Graph Decomposer* module splits the graph into three distinct subgraphs corresponding to an over-constrained system of equations (too many equations are present), an under-constrained system (too few equations or too many variables are present in the system) and a well-constrained system of equations (the number of variables is equal to the number of equations). A simple heuristic filtering rule assumes that the well-constrained part obtained after decomposition will lead to a solvable system of equations and therefore need not be included in any repair strategy. If under- or over-constrained situations are detected, this means that there are some inconsistencies in the model. One should note that if no inconsistencies are detected in the model, the compilation process follows the path from the original declarative code to the transformed procedural code through the *Equation Flattener*, *Graph Mapping*, *Graph Decomposer*, and *Code Generation* modules.

### 8.3.4 Constraint Analyser

The *Constraint Analyzer* takes the over- or under-constrained graphs obtained from the previous phase and performs a graph transformation in order to transform these graphs into a well-constrained graph and then generates a program modification at the intermediate source code level based on the transformed computed graph. Different graph transformations are applied for over- and under-constrained components. Details on how these transformations are performed have already been discussed in Chapter 5 and Chapter 6. Based on structural properties of the underlying over- and under-constrained subgraphs intermediate code program modifications is performed by the *Constraint*

*Analyser*. The program modifications always lead to well-constrained systems of equations and thereby eliminate the symptom of the bug.

### 8.3.5 Code Transformer

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

### 8.3.6 Error Output

This 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 prioritized based on the annotations provided by the *Equation Flattener*. 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 bug at the intermediate code level. The corrections that most closely correspond to the programmer's view of the model behavior should be selected.

For the implemented debugger the user has the possibility of specifying different levels of filtering the error messages. First, we have provided two general levels of debugging:

- *General Level 1*: Generate error messages that exclude error-fixing modifications for those components and library models that previously have been marked as "*locked*". Usually well-tested libraries such as those included in the Modelica Basic Library are safe and can be market as "*locked*". Error fixing solutions that include modification of the "*locked"* library models are not taken into account. Recently developed models can be "*locked*" by the user through the selection facilities provided in the class browser associated to the source code editor or even through the graphical model editor.
- *General Level 2*: Generate error messages that include modifications for any components.

The debugging of over-constrained systems includes two secondary levels:

- *Secondary Level 1*: Present corrections, not including those equations that disconnect the underlying bipartite graph corresponding to the equation system.
- *Secondary Level 2*: Present all the possible corrections that might involve the elimination of equations including those that disconnect the underlying bipartite graph. However in these cases a reachability analysis is performed in order to find possibilities of reconnecting the bipartite graph.

For debugging under-constrained models an additional four secondary levels of debugging might be present which are suboptions to the two general levels:

- *Secondary Level 1*: Take into account those corrections that imply the removal of free variables from the under-constrained system. However do not use those variable removals that will disconnect the associated graph.
- *Secondary Level 2*: Remove extra variables and provide additional linking information to those cases where the removal of a variable will disconnect the underlying bipartite graph. This means that a reachability analysis is performed on the system.
- *Secondary Level 3*: Instead of only removing free variables, the addition of extra equations is also taken into account. Reachability analysis is then performed on the added equations in order to show the user the possible presence of other variables in the equations.
- *Secondary Level 4*: Present all possible error fixing solutions for the user.

Our experience of using the debugger has shown that the most efficient settings is when the general and the secondary level are set to *level1*, for models that contains small deviations such as accidentally removed equations, duplicate equations, uninitialised variables, and wrong connections among components.

The remaining debugger level combinations are more useful in detecting subtle bugs, especially in those cases where the overall model contains many missing equations. The remaining levels are also useful in the design phase by providing additional information regarding possible model combinations and possible presence of variables in different equations.

## 8.4 Implementation Status

Both debuggers are in a prototype stage. The debugger attached to the MathModelica environment has been developed to test the debugging hypotheses presented in this thesis. First debugging experiments conducted in the MathModelica environment have yielded encouraging results that started the development of the second debugging kernel designed for the Open Source Modelica compiler. The experience with the prototype has shown that the proposed debugging scheme presented in this thesis provides a low-cost, practical approach to statically debug Modelica programs. Moreover, a Modelica debugger and compiler can be built using the same graph theoretical abstraction providing in this way a uniform framework.

We are currently working on expanding the scope and functionality of the AMOEBA debugger and integrate it in the Open Source Modelica project. Obviously the scope of the debugger needs to be extended with dynamic debugging capabilities. Currently, AMOEBA's functionality is limited mostly due to our inability to provide automatic equation annotations. For this reason only a limited number of examples with limited size and complexity have been tested. At the current implementation stage AMOEBA is able to decompose the flattened system of equations based on Dulmage-Mendelsohn algorithm and provide error-fixing messages for the over- and under-constrained components. Moreover, the debugger is also able to decompose the system into a Block Lower Triangular form, reduce the index of differential algebraic equations, and provide consistent initialisation of DAEs

# Chapter 9

# Related Work

***Summary****: This chapter surveys the work which is most closely related to ours. The related work overview can clearly be divided into three distinct parts: one part is related to constraint satisfaction approaches, the second part surveys related work done in the area of structural analysis, and the third part deals with failures in object-oriented modeling and simulation systems as well as diagnosability of such systems.*

## 9.1 Introduction

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

Nowadays, complex 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 mathematical characterization of the modeled systems. Obviously, a connection between structural analysis and programming languages needs to be established in order 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. For this reason, most of the related work presented below uses a graph-based abstract representation in order to provide a more intuitive depiction of the computation process. The user is provided with different views at different levels of detail, which allows an easy understanding of the whole modeling process when using equation-based languages.

## 9.2 Constraint Satisfaction Approaches

Our debugging approach follows the same philosophy as the method for reduction of constraint systems used for geometric modeling in (Ait-Aoudia et. al. 1993 [3]). In this system the algebraic equations resulting from a geometric modeling by constraints problem are decomposed into well-constrained, over- and under-constrained subsystems for debugging purposes. The well-constrained systems are further decomposed into irreducible subsystems. This speeds up the resolution in case of reducible systems which are decomposed into smaller reducible subsytems. The methods developed in the paper help to gain knowledge on the combinatorial structure of the underlying equation systems to geometric constraints and provide valuable information for debugging such systems.

In (Bliek et. al 1998 [19]) attention is paid to the well-constrained part of a system of equations by proposing new algorithms for solving the structurally well-constrained problems by combining the use of constraint solvers with intelligent backtracking techniques. The backtracking of the ordered blocks is performed when a block has no solution. This approach deals mostly with numerical problems, which of course are due to erroneous modeling and wrong declarative specification of the problem, and requires the use of constraint solvers. In (Bliek et. al 1998 [19]) an algorithm for under-constrained problems is presented as well, which deals with the problem of selecting the input parameters that lead to a good decomposition.

In (Bakker et. al. 1993 [9]) a method called DOC (Diagnosis of Over-determined Constraint satisfaction problems) is described that solves over-determined constraint satisfaction problems by generating a set of constraints that should be relaxed in order of increasing cost. The method first identifies the set of least important constraints that should be relaxed to solve the remaining constraint satisfaction problem. If the solution is not acceptable for the user, the next-best set of least important constraint is scheduled by the method until an acceptable solution from the user point of view is found and validated. The classification mechanism of DOC is based on weighting the underlying set of constraints.

(Müller 2000 [86]) proposes an interactive tool called Constraint Investigator for debugging constraint-based application models. The tool is targeted at problems like wrong, void, or partial solutions, and is not restricted to any specific constraint system. Internally the debugging tool uses a graph-based representation that makes it possible to efficiently emphasize different aspects of the constraints in the constraint solver.

## 9.3 Structural Analysis Techniques

(Murota 2000 [87]), (Murota 1987 [88]) presents matroid-theoretic combinatorial methods for equation system analysis. In Murota's approach equation systems are described by mixed polynomial matrices. Mathematically, the analysis of mixed polynomial matrices is based on the combinatorial canonical form, which generalizes the Dulmage-Mendelsohn's decomposition. Structural solvability criteria are also formulated based on graphical and matroidal conditions.

In (Leitold and Hangos 2001 [75]) the variable structure of dynamic models is represented by a directed graph for the purpose of solvability analysis. The graph representation is suitable for computing the structural differential index of lumped dynamic process models. A structural decomposition is derived as well. The paper proposes a graph theoretical method for index analysis and also develops index reduction methods. The ODEs and DAEs of a dynamic simulation model are represented by dynamic graphs (a sequence of static graphs corresponding to each step of numeric integration).

(Gilli and Garbely 1996 [49]) presents structural analysis techniques for diagnosing macro-economic models. By means of graph theoretic approaches the normalization of equations is computed as a necessary condition for the nonsingularity of the Jacobian matrix. The paper analyses, in a systematic way, those situations in which the model does not admit normalization and therefore the local uniqueness of the solution of a system of equations is no longer guaranteed. In this case, the Jacobian matrix is reordered based on the minimum cover of the bipartite graph associated to the system of equations. The reordering permits isolation of the error and clearly indicates where the modifications of equations should occur in order to obtain a normalization.

The Abacuss II environment (Tolsma et. al. [117]) provides a debugging module for structural analysis of the underlying systems of equations. The environment gives the user the possibility to output a file which lists all the equations, initial conditions, variable and parameters. The equations are presented in flattened form. Starting from the Dulmage & Mendelsohn canonical decomposition of the incidence matrix, Abacuss II also has the possibility to visualize the over- and under-constrained components (Barton 2000 [15]) (Barton 1995 [14]). All the variables and equations of the under- and over-constrained equation system blocks can be inspected. If a block is sufficiently small, simple inspection of the equations provides valuable information for the modeler. The debugging capabilities of Abacuss II are completed by the Block Solver Diagnosis module that permits the visualization of the equations and variables that have failed during the solving process. The bounds on the variables and the current values of the variables can also be visualized by the Block Solver Diagnosis module. In this way the source of runtime errors resulting from the failure of the attached numerical solver to compute a solution is reduced to a small block of equations.

## 9.4 Diagnosability of Modeling Systems

(Ramos et. al. 1998 [99]) propose a methodology which adds a layer of physical knowledge in the analysis of simulation models described with object-oriented equation-based modeling languages such as Dymola or Omola. The topology of submodel interconnections is analyzed and well known algebraic problems such as systems of simultaneous equations and high index DAEs are identified based on the extracted information. Some numerical problems are avoided by exchanging submodels with equivalent submodels with better numerical properties. The authors have developed a symbolic formula manipulation kernel in Maple in order to obtain the mathematical representation of the equivalent model behavior.

Regarding systems modeled with non-linear and linear differential algebraic equations, (Krysander and Nyberg 2002 [74]) present an algorithm for finding a small set of

submodels, which can be used to derive consistency relations with the highest possible diagnosis capability. The fault diagnosability of the system is characterized by the minimal structurally singular (MSS) set of equations. The proposed algorithm operates by first detecting the over-constrained submodels; subsequently the submodels are transformed into consistency relations. The MSS set of equation is computed by means of matchings in bipartite graphs.

In (Wani and Gandhi 2000 [121]) a method developed on the basis of relationships between system performance parameters and physical objects for mechanical and hydraulical systems for assessment of diagnosability is presented. The relationships are modeled in terms of a bipartite graph called the Diagnosability Bipartite Graph (DBG) and the associated matrix to the bipartite graph called Diagnosability Matrix (DM). Various diagnostic parameters are obtained from these representations, which aid the designer in evaluating and comparing the diagnosability of various design variants of the system. The diagnosability of the system is based on proposed parameters such as: maximum number of set conflicts (MNS), maximum number of parameters in a set conflict (MNCS), diagnosability effort and cost (DEC), and average merit of diagnosability.

# Chapter 10

# Evaluation Based on Usability Criteria

***Summary**: This section presents our evaluation of our debugging framework based on existing usability criteria developed for algorithmic automated debugging.*

## 10.1 Evaluation of the Debugging Framework.

The objective of this research is to bring the latest advances from the areas of debugging and program analysis to develop a powerful framework that will permit the integration of debugging tools into simulation environments that employ declarative equation-based languages. These debugging tools should be easy to use by a variety of simulation system end users with varying mathematical and computational expertise ranging from normal users to expert library developers. In this section we analyze and evaluate our debugging framework based on the usability criteria for automated algorithmic debuggers presented in (Shahmehri et al. 1995 [109]) and in relation to the automated debugging techniques presented in (Ducassé [29])

### 10.1.1 Usability Criteria Based Evaluation

*Generality.* At this stage our framework successfully handles two classes of bugs: over-constrained and under-constrained situations. These kinds of bugs are usually the most frequent error situations encountered when programming with declarative equation-based languages. Most of the erroneous specifications at the equation level will fall into one of these categories. Our approach successfully handles over-constrained situations when multiple equations overconstrain the problem, under-constrained problems with missing equations and/or many variables, as well as combined errors when over and under-constrained situations are simultaneously present in the model specification.

*Cognitive plausibility.* Bug localization is performed at the intermediate code level because only at this level can the structural dysfunctionality of the system of equations be

detected. In order to provide meaningful error messages and error fixing solutions the trace of transformations from the original code to the intermediate code is saved. On the other hand, these transformations are used to generate powerful filtering rules for the combinatorial explosion at the level of the intermediate code.

Each time an error is detected in the intermediate code and the error fixing solution is elaborated, the debugger queries for the original source code statement before presenting any information to the user. Moreover, the algorithmic part of the debugging session, when the user interacts with the debugger and tries to supplement the missing formal specification, is performed at the original source code level. Furthermore, if possible the interaction is performed at the graphical editor level although these situations do not occur very common. For example, error-fixing solutions that involve the removal of connectors or model components from the whole simulation model can be presented at the graphical level. This facility is useful for end-users who interact with the system at the graphical model editing level, but has turned out to be counterproductive for library developers.

Most of the bugs that would usually require a third-level problem solving cognitive process from the user, (Hale et. al. 1999 [55]) (Hale and Haworth 1991 [56]) have been automatically moved by the debugger and transformed into problems that require first-level problem solving cognitive processes which only involve the invocation of simple atomic code modification rules to directly eliminate the program bug.

*Degree of automation.* In our prototype system a high degree of automation of the debugging process has been achieved, reducing the user interaction to a minimum. Only at the end of the debugging process is the user interrogated concerning the correct error fixing solution that needs to be applied. The structural analysis and filtering algorithms are fully automated, the annotation of the equations is automatically generated and the filtering rules are derived from the language semantics and saved during the translation phase of the compiler.

*Appreciation of the user's expertise*. In our system the user-interaction is reduced to "yes" and " no" answers. In the debugging tool, we have provided the possibility of selecting different levels of debugging, especially for under-constrained systems, as presented in Chapter 6. The users interested in the detailed translation and equation optimization process have the possibility of saving and visualizing the bipartite graph that represents the flattened intermediate form. The Block Lower Triangular form obtained after equation sorting can also be visualized by the debugger for program understanding purposes.

We can summarize the evaluation of our system in the following table:

**Table 10-1.** Evaluation of the debugging framework based on usability criteria

| Generality | Cognitive Plausibility | Degree of Automation | Appreciation of User Expertise |
|---|---|---|---|
| Yes | High | Semiautomatic<br>Full (when applicable) otherwise partly manual | Yes |

### 10.1.2 Strategies for Automated Debugging

*Verification with respect to a specification:* Usually this technique is not so useful for large programs if the complete specification is missing or in the absence of an automated tool that is able to extract the specifications (Ammons et. al. [4]). In our case it turns out that purely mathematical specifications encoded as graph matching abstractions are sufficient to provide an acceptable formal specification of the intermediate flattened code. After applying different filtering techniques the intended program is represented by a modified program expressed in the intermediate code obtained from program modifications. Since the error location problem is handled at the modified intermediate code level, the transformation traces saved during the compilation process are used to map the error back to the original source code.

*Slicing.* This technique in the case of imperative programming languages is a data flow dependency analysis triggered by "wrong value output variable" or "wrong control sequence". In our debugger, the slicing is used to flatten the `if` statements in the intermediate code and select the set of equations on which the canonical decomposition is going to be applied.

*Heuristic Filtering.* The main heuristic filtering rules of our debugging framework are mostly based on structural analysis performed on the underlying systems of equations and from graph connectivity considerations. At the first stage the equations and variables of the well-constrained part of the D&M canonical decomposition are eliminated from any further consideration. Only the remaining equations of the under- and over-constrained parts are kept by the debugger. The second main filtering rule is only based on structural consideration of the bipartite graphs the system of equations. For example, as shown in Chapter 5 all the equation nodes that being eliminated disconnect the underlying bipartite graph are eliminated from any error fixing strategies implying over-constrained systems.

*Program Corrections.* After applying the heuristic filtering the debugger has all the necessary information to create program corrections (also called mutations by some authors) of the intermediate source code that will cause the symptoms of the bugs to disappear. In most of the cases a combinatorial explosion is experienced at this stage where many possible error fixing solutions might be feasible.

*Program Transformation.* At this stage the original program needs to be modified to generate a corrected intermediate program. These modifications were defined at the previous step where mutated versions of the intermediate code were created. Several powerful filtering rules derived from the semantics of the language will guide the transformations of the original program, eliminating most of the mutated intermediate code variants.

*Algorithmic behaviour verification.* This stage is necessary to help the user to select the error-fixing solutions that will yield the correct behaviour of the modelled systems. A strategy for error fixing, based on equation annotations, has already been presented by the debugger but usually the heuristics involved in the annotation process are quite weak and the user validation is necessary at this step. It is worth noting that this stage is performed at the original source code level and not on the flat intermediate form. This

was facilitated by saving the translation traces for each original statement to the intermediate code during the compilation phase.

## 10.2 The Debugging Process from the Automated Debugging Perspective

Now, let us summarize the debugging process from the automated debugging perspective, by showing the order in which each of the above mentioned techniques are used. The debugging process starts by first filtering the problem at the intermediate code level by verifying each valid "*slice*" of the program.

Then based on the mathematical formal specifications concerning the solvability of the system of equations coded as matching algorithms, sets of equations are isolated by performing canonical decomposition on the corresponding graph. 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.

Based on structural properties of the underlying over and under-constrained graph, *program correction* is used to create a model program at the intermediate code level. Then *program transformation* is used to validate the *corrections* (sometimes called mutants) and provide error messages in terms of the original program. When several valid program transformations have been selected *algorithmic behaviour verification* is necessary: the user acts like an "*oracle*" completing the missing formal specifications and selecting the right behaviour of the problem. The algorithmic debugger traverses the equation dependency tree for over-constrained systems and a multilevel decision tree for under-constrained systems, asking the user about the removal of an equation or the removal of a variable or new equations additions.

When multiple errors are present in the program, the debugger will try to locate all of them at the same time. Repeating the whole process for an over-constrained and an under constrained situation simultaneously present in the model is not necessary, both errors can be fixed in the same debugging session.

# Chapter 11

# Conclusions and Future Work

***Summary**: This presents our conclusions to the work presented in this thesis. We also provide an insight into future research related to debugging of equation-based declarative languages. While some more experimental work remains ahead of us, initial experience is promising.*

## 11.1 Discussion and Comparison

Our work and the previously presented related work share the common goal of providing users with an effective way of debugging constraint or equation-based problems. However, none of the environments presented in the related work section maintains a mapping between the source text, the original intermediate representation, and the optimized intermediate representation. To our knowledge, no other existing simulation system which employs an equation-based modeling language performs a mapping between information obtained from graph decomposition techniques and the original program source code.

In particular, the Abacuss debugger (Tolsma et. al. [117]), (Barton 1995 [14]) has some similarities to our approach and shares a common goal. The Abacuss debugger is also based on the Dulmage-Mendehlson decomposition. However, no filtering based on rules derived from the semantics of the original language or from structural information is provided. The Abacuss II environment only visualizes the over- and under-constrained components. The user can isolate the over-constraining equation or the free variable by visually examining these components. For very small systems the error can easily be found but for larger systems this quickly becomes an extremely difficult and error-prone task. The same technique is employed in (Ait-Aoudia et. al. 1993 [3]) for debugging constraint systems used for geometric modeling. We believe that just isolating and showing the user the over and under-constrained components is not enough in the context of debugging equation-based languages. Therefore, our approach differs in a number of ways, even though the first step also uses the D&M canonical decomposition.

We have presented derivations of the analysis of over and under constrained components obtained after the Dulmage-Mendehlson decomposition that are more extensible than previous approaches and more effective for program debugging purposes. The *main contribution* of this thesis is to apply *semantic filtering rules* on the over and un-

der-constrained components. Moreover, filtering rules obtained from structural properties of the over- and under-constrained graphs contribute to a substantial reduction in number of error messages. We argue that simple elimination of equation or variable nodes that disconnect the underlying bipartite graph cannot be considered as valid solutions from the modeling point of view. Our system takes such considerations into account. In this way efficient error fixing messages can be presented to the programmer in a natural and intuitive manner and only those solutions are validated that can be obtained by source code manipulations.

Compared to other approaches of dealing with under-constrained systems that only consider the elimination of the free variables from the system, we have *extended our approach* by also considering the *addition of new equations* into the system. In this context the reachability analysis is proposed to assist the user to provide the right form of the introduced new equation. This technique supervises the debugging process by interactively providing hints and immediately checking the correctness of a modification.

The debugging framework, presented in this thesis, also allows the analysis of *complex erroneous situations* when *over and under-constrained appear simultaneously*. We have pointed out that these situations cannot be treated by considering the over and under-constrained problems in isolation. They need to be debugged simultaneously. It remains an open question whether there is a general algorithm that will be able to automatically handle such situations.

We have also explored the possibility of *annotating the equations* to narrow down the number of error fixing choices and to prioritize the solutions. In that way we can take advantage of library models that are fully tested and eliminate them from any error fixing solutions. This also provides a means to use and attach domain related information to the equations and to elaborate domain specific filtering strategies on top of the semantic and structural filtering rules.

Our debugging framework is enhanced by incorporating ordinary and differential algebraic equations (DAEs) into the static analysis by manipulating the system of equations to achieve an acceptable index and by providing consistent initial conditions (Pantelides 1988 [93]). The debugging framework tries *to keep a mapping* between the source text, the original intermediate representation and the symbolically optimized intermediate representation.

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. Details on how this goal has been achieved is given in Chapter 8.

We claim that the techniques developed and proposed in this thesis are suitable for a wide range of equation-based languages and can easily be adapted to the specifics of a particular simulation environments. Our claim is based on the *close integration of the developed debugging techniques and the compilation process*. System decomposition, partial ordering of equation blocks into BLT form, symbolic manipulation, index reduction, tearing, and providing consistent initial conditions can be done using the same graph theoretical abstractions based on the bipartite graphs.

## 11.2 Prospects for Future Improvements

While some more experimental work remains ahead of us, initial experience is promising. In the work presented we have tried to deal with the static aspects of the debugging process, namely the detection and fixing of over and under-constrained situations. Obviously the next step would be to improve these techniques by providing better user interaction and extending the debugger to handle dynamic situations as well.

The following specific issues are planned to be addressed in our future work concerning the debugging of equation base languages:

- The ideas presented in the thesis have been tried only on a very limited scale of examples, even through the principles, especially those handling over-constrained system situations have been developed in detail. The implemented debugger kernel will be useful when it can handle large system models. The restrictions were imposed mostly by our inability to provide automatic equation annotations and to save the transformation rules during the previous phases of the debugging. The parser and the modules involved with the transformation of the language need to be modified and extended in order to provide better support for such annotations.
- To extend the algorithms that are concerned with over- and under-constrained situations when they appear simultaneously in the same model. Usually the error fixing solutions are much more complex than separately handling each situation. These situations involve a complex interaction, as was shown in the simple example from Chapter 6, section 6.5.
- Development of an integrated debugging kernel tightly integrated with the core of the optimizer module of the compiler for supporting Modelica environments. We intended to couple the debugging and the optimization kernel into one module that should be integrated into a free Modelica compiler (Fritzson et al. [73]). Nevertheless, most of our results developed during this research should be also applicable for other equation-based modeling languages. We aim to develop a language independent debugging and optimization kernel.
- Development of static system diagnosis modules with enhanced domain knowledge information. These modules will be responsible for mapping the error conditions to specific domain patterns taken from physical system domain knowledge databases. This will contribute to an improved diagnosability and advanced failure analysis of complex physical systems.
- Development of dynamic diagnosers which will perform localization of program errors from symptoms found at the numerical solver level. Such numerical singularities cannot be detected by static analyzers.
- Extension of the debugging kernel for model-based debugging in order to locate faulty components in a given physical system simulation model. The model-based diagnosis is performed by having a model of a system under consideration, a logical representation of the correct behavior, and a set of observations (Wieland 2001 [123]).

- Extend the approach to environments that employ constraint satisfaction solvers and adapt the developed techniques for equation-based languages to partial constraint-satisfaction problems.

## 11.3 Final Words

We expect that this work will considerably improve the debugging technology for system modeling using declarative equation-based languages. Such progress in the development of debugging environments for these languages might increase the acceptance of a new language such as Modelica in the engineering communities that it targets. The general debugging framework developed during the project can also serve as a basis for compiler construction research regarding equation-based modeling languages as well as providing valuable information on how to design efficient optimizing compilers for equation-based languages.

# References


[1] Abadi M., and Cardelli, L. *A Theory of Objects*. Springer Verlag, 1996.

[2] Aggoun A., F. Bueno, M. Carro, P. Deransart, M. Fabris, W. Drabent, G. Ferrand, M. Hermenegildo, C. Lai, J. Lloyd, J. Maluszynski, G. Puebla, A. Tessier. "CP Debugging Needs and Tools" In *Proceedings of the Third International Workshop on Automatic Debugging AADEBUG '97*. (Linköping May 26-27 1997).

[3] Ait-Aoudia, S.; Jegou, R. and Michelucci, D. “Reduction of Constraint Systems.” In Compugraphic, pp 83--92, Alvor, Portugal, 1993.

[4] Ammons Glenn, Rastislav Bodik and James Larus. "Mining Specification." In the *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon, January 2002).

[5] Andersson Mats. "Combined object-oriented modelling in Omola." In Stephenson, Ed., *Proceedings of the 1992 European Simulation Multiconference* ESM '92, York, UK, Society for Computer Simulation International, June 1992

[6] Aronsson Peter, Peter Fritzson, Levon Saldamli, Peter Bunus. "Incremental declaration handling in Open Source Modelica." In *Proceedings of 43rd the Scandinavian Conference on Simulation and Modeling* (SIMS 2002), (September 26-25 Oulu, Finland, 2002)

[7] Asratian A.S., T. Denley and R. Häggkvist. *Bipartite Graphs and their Applications*. Cambridge University Press 1998.

[8] Baecker Ron, Chris DiGiano and Aaron Marcus. *Software Visualization for Debugging*. Communications of the ACM, Vol. 40, No. 4, April 1997.

[9] Bakker R.R., F. Dikker, F. Tempelman, P.M Wognum. "Diagnosing and Solving Over-Determined Constraint Satisfaction Problems" In *Proceedings of the 13th International Joint Conference on Artificial Intelligence* (IJCAI' 93) pp. 276-281, Chambery France, Morgan Kaufmann, 1993.

[10] Ball Thomas, Sriram K. Rajamani. "Automatically Validating Temporal Safety Properties of Interfaces." In *Proceedings of, Workshop on Model Checking of Software, SPIN 2001*, LNCS 2057, pp. 103-122, May 2001.

[11] Ball Thomas, Sriram K. Rajamani. "The SLAM Project: Debugging System Software via Static Analysis." In the *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon USA). pp 1-3 January 2002.

[12] Barton Paul Inigo and C.C. Pantelides, *The Modelling of Combined Discrete/Continuous Processes*. AIChemE Journal 40, pp. 996-979 (1994).

[13] Barton Paul Inigo and C.C. Pantelides. "gPROMS - A Combined Discrete/Continuous Modelling Environment for Chemical Processing Systems." In *Proceedings of International. Conf. Simul. Engg. Educ., The Society for Computer Simulation, Simulation Series*, 25, 25-34 (1993)

[14] Barton Paul Inigo. *Structural Analysis of Systems of Equations*. Massachusetts Institute of Technology Chemical Engineering System Research Group. July 1995 Available from http://yoric.mit.edu/abacuss2/Lecture2.pdf

[15] Barton Paul Inigo. *The Equation Oriented Strategy for Process Flowsheeting*. Massachusetts Institute of Technology Chemical Engineering System Research Group. March 2000. Available from http://yoric.mit.edu/abacuss2/Lecture1.pdf

[16] Barton Paul Inigo. *The Modelling and Simulation of Combined Discrete/Continuous Processes*. PhD. Thesis, Department of Chemical Engineering, Imperial College of Science, Technology and Medicine, University of London, 1992.

[17] Beek van D.A. "Variables and Equations in Hybrid Systems with Structural Changes." In *Proceedings of the 13th European Simulation Symposium*, Marseille, 2001, pp 30-34.

[18] Benson Bjorn Freeman and Alan Borning. "Integrating Constraints with an Object-Oriented Language." In *Proceedings of the 1992 European Conference on Object-Oriented Programming*, pp 268-286, June 1992

[19] Bliek, C., B. Neveu, and G. Trombettoni "Using Graph Decomposition for Solving Continuous CSPs", *Principles and Practice of Constraint Programming*, CP'98, Springer LNCS 1520, pp 102-116, Pisa, Italy, November 1998.

[20] Borning A. "Classes Versus Prototypes in Object-Oriented Languages." In *Proceedings of the ACM/IEEE Fall Joint Computer Conference*, pp 36-40 (Dallas, Texas, 1986)

[21] Borning Alan H. *Thinglab: A Constraint-Oriented Simulation Laboratory*. PhD thesis, Stanford University, Stanford, California, July 1979.

[22] Brenan K., S. Campbell, and L. Petzold. *Numerical solution of initial-value problems in ordinary differential-algebraic equations*. North Holland Publishing Co., New York, 1989.

[23] Breuer, P.T., N.M. Madrid, J.P. Bowen, R. France, M.L Petrie, C.D. Kloos. "Reasoning about VHDL and VHDL-AMS using denotational semantics." In *Proceedings of Design, Automation and Test in Europe Conference and Exhibition*. pp 346 -352, 1999.

[24] Cellier Francois. E. *Continuous System Modelling*, Springer Verlag, Berlin, 1991.

[25] Christen, E. and K. Bakalar. *VHDL-AMS-a hardware description language for analog and mixed-signal applications*. IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing. Volume: 46, Issue: 10, pp 1263 - 1272, Oct. 1999

[26] Cleve Holger, Andreas Zeller. "Finding Failure Causes through Automated Testing." In *Proceedings of the Fourth International Workshop on Automated Debugging*, Munich, Germany, 28-30 August 2000.

[27] Dolan A. and Aldous J. *Networks and algorithms – An introductory approach*. John Wiley & Sons England. 1993.

[28] Drager, S.L., H.W. Carter, H.L Hirsch. "A VHDL-AMS mixed-signal, mixed-technology design tool." In *Proceedings of the IEEE 1998 National Aerospace and Electronics Conference*, 1998. NAECON (13-17 July 1998, Dayton, OH, USA) pp 552 - 556, 1998.

[29] Ducassé Mireille. "A pragmatic survey of automated debugging." In Proceedings of *1st International Workshop on Automated and Algorithmic Debugging*, LNCS 749, Springer Verlag, 1993.

[30] Ducassé Mireille and J. Noyé. *Logic Programming Environments*: *Dynamic Program Analysis and Debugging*, Journal of Logic Programming, Vol.19/20, pp.351-384, 1994.

[31] Duff Iain. and C. W. Gear. *Computing the Structural Index*, SIAM J. Alg. Discrete Methods, 7 (1986), 594--603.

[32] Duff Iain S., Albert M. Erisman, and John K. Reid. *Direct Methods for Sparse Matrices.* Clarendon Press, Oxford, 1986.

[33] Dulmage, A.L., N.S. Mendelsohn. *Coverings of bipartite graphs*, Canadian J. Math. 10, pp 517-534. 1963.

[34] Elmqvist Hilding and Martin Otter. "Methods for tearing systems of equations in object oriented modeling." In *Proceedings of the European Simulation Multiconference*, ESM'94, pp. 326--334, Barcelona, Spain, June 1994. SCS, The Society for Computer Simulation.

[35] Elmqvist Hilding, Sven Erik Mattsson and Martin Otter. "Modelica - A Language for Physical System Modeling, Visualization and Interaction." In *Proceedings of the 1999 IEEE Symposium on Computer-Aided Control System Design* (Hawaii, Aug. 22-27) 1999.

[36] Elmqvist Hilding. *A Structured Model Language for Large Continuous Systems*. PhD thesis TFRT-1015, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden. 1978.

[37] Fábian Georgina. *A language and simulator for hybrid systems*. Phd Thesis 1999. Technische Universiteit Eindhoven, the Netherlands.

[38] Feehery W., and P. Barton. *Dynamic optimization with state variable path constraints*. Computers in Chemical Engineering, v. 22, no. 9, pp. 1241-1256, 1998.

[39] Flannery, L. M. and A. J. Gonzalez. *Detecting Anomalies in Constraint-based Systems*, Engineering Applications of Artificial Intelligence, Vol. 10, No. 3, June 1997, pages. 257-268. 1997.

[40] Frey, P. K. Nellayappan, V. Shanmugasundaram, R.S. Mayiladuthurai, C.L. Chandrashekar, H.W. Carter. "SEAMS: simulation environment for VHDL-AMS." In *Proceedings of Winter Simulation Conference* (13-16 Dec. 1998, Washington, DC, USA). Vol.1. pp 539 –546, 1998.

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

[42] Fritzson Peter, and Vadim Engelson. "Modelica - A Unified Object-Oriented Language for System Modeling and Simulation." In *Proceedings of the 12th European Conference on Object-Oriented Programming* (ECOOP'98, Brussels, Belgium, Jul. 20-24), 1998.

[43] Fritzson Peter. *Introduction to Modelica*. Book draft. 2002. First chapter available at https://www.ida.liu.se/~pelab/rooms/

[44] Fritzson Peter, Peter Aronsson, Peter Bunus, Vadim Engelson, Henrik Johansson, Andreas Karström, Levon Saldamli. "The Open Source Modelica Project." In *Proceedings of the 2nd International Modelica Conference* (18-19 March, Munich Germany). 2002.

[45] Fritzson Peter, Mats Jirstrand, Johan Gunnarsson. "MathModelica - An Extensible Modeling and Simulation Environment with Integrated Graphics and Literate Programming". *In Proceedings of the 2nd International Modelica Conference* (18-19 March, Munich Germany, 2002)

[46] Fritzson Peter, Nahid Shahmehri, Mariam Kamkar. and Tibor Gyimothy. *Generalized algorithmic debugging and testing*. ACM Letters on Programming Languages and Systems, 1(4):303--322, December 1992.

[47] Fukuda, K., T. Matsui. *Finding All The Perfect Matchings in Bipartite, Appl. Math. Lett*. Vol. 7, No. 1, pp 15-18. 1994.

[48] Gibbons, A. *Algorithmic Graph Theory*. Cambridge University Press 1985

[49] Gilli Manfred and Myriam Garbely. *Matchings, covers, and Jacobian matrices*. Journal of Economic Dynamics and Control. 20 (1996) 1541-1556.

[50] Govindarajan K., B. Jayaraman, and S. Mantha. "Optimization and Relaxation in Constraint Logic Languages." In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pp 91--103, 1996.

[51] Günther M., U. Feldmann. *The DAE-index in electric circuit simulation*. Mathematics and Computers in Simulation 39, 573-582 (1995).

[52] Hakman Mikael and Torgny Groth. *Object-oriented biomedical system modelling - the language*. Computer Methods and Programs in Biomedicine, Volume 60, Issue 3, pp 153-181, November 1999.

[53] Hakman Mikael and Torgny Groth. *Object-oriented biomedical system modelling -The Rationale*. Computer Methods and Programs in Biomedicine, Volume 59, Issue 1, pp 1-17, April 1999.

[54] Hakman Mikael. *Methods and tools for object-oriented modeling and knowledge-based simulation of complex biomedical systems*. PhD Thesis – Uppsala University, 2000.

[55] Hale E. Joanne, Shane Sharpe, David P. Hale. *An Evaluation of the Cognitive Processes of Programmers Engaged in Software Debugging*. Journal of Software Maintenance. Vol. 11. No. 2, pp. 73-91, March-April 1999.

[56] Hale P. David and D.W. Haworth. *Toward A Model of Programmers' Cognitive Processes in Software Maintenance: A Structural Learning Theory Approach for Debugging*. Journal of Software Maintenance. Vol. 3, No. 2, pp. 85-106, Spring 1991.

[57] Harrold, M.J and G. Rothermel. “A Coherent Family of Analyzable Graphical Representations for Object-Oriented Software.” *Technical Report* OSU-CISRC-11/96-TR60, November, 1996, Department of Computer and Information Science Ohio State University. 1996.

[58] Himsolt Michael "GML: A portable Graph File Format". Technical Report Universität Passau 1997.

[59] Himsolt Michael. GML: Graph Modeling Language. 1996. Available at http://www.infosun.fmi.uni-passau.de/Graphlet/GML/

[60] Hopcroft, J.E. and R.M. Karp. *An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs*. SIAM Journal of Computing, 2(4): pp 225--231, December 1973.

[61] IEEE Std 1076.1-1999. IEEE Computer Society Design Automation Standards Committee, USA. *IEEE standard VHDL analog and mixed-signal extensions*. 23 Dec. 1999.

[62] IEEE Std 610.12-1990. *IEEE Standard Glossary of Software Engineering Terminology*. 1990

[63] Jaramillo C., R. Gupta and M.L Soffa. "FULLDOC: A Full Reporting Debugger for Optimized Code". In *Proceedings of International Static Analysis Symposium*, LNCS, Springer Verlag, Santa Barbara, CA, July, 2000.

[64] Jaramillo, C., Gupta, R., and Soffa, M.L. "Comparison Checking: An Approach to Avoid Debugging of Optimized Code". In *Proceeding of ACM SIGSOFT Symposium on Foundations of Software Engineering and European Software Engineering Conference*, pages 268--284, September 1999.

[65] Jayaraman B. and P. Tambay. "Compositional Semantics for Diagrams using Constrained Objects" In *Proceedings of the Second International Conference on Theory and Application of Diagrams* (April 18-20, Georgia USA, 2002). LNAI 2317. Springer Verlag 2002.

[66] Jayaraman B. and P. Tambay. "Modeling Engineering Structures with Constrained Objects". In *Proceedings of International Symposium on Practical Applications of Declarative Languages* (Portland, OR, pp. 28-46) LNCS 2257 Springer-Verlag, January 2002.

[67] Jayaraman B. and Tambay P. "Semantics and Applications of Constrained Objects" Technical Report 2001-15 at Department of Computer Science and Engendering, State University of New York at Buffalo. October 12, 2001.

[68] Jirstrand, M. "MathModelica – A Full System Simulation Tool". In *Proceedings of Modelica Workshop 2000* (Lund, Sweden, Oct. 23-24), 2000.

[69] Jirstrand Mats, Johan Gunnarsson and Peter Fritzson "MathModelica – a New Modeling and Simulation Environment for Modelica." In *Proceedings of the Third International Mathematica Symposium* (IMS'99, Linz, Austria, Aug), 1999.

[70] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. *The CLP(R) language and system*. ACM Transactions on Programming Languages and Systems (TOPLAS), 14(3): 339--395, July 1992.

[71] Kågedal David and Peter Fritzson. "Generating a Modelica Compiler from Natural Semantics Specifications". In *Proceedings of the Summer Computer Simulation Conference* '98, Reno, Nevada, USA, July 19-22, 1998.

[72] Kahn Gilles. "Natural Semantics." In *Proc. of the Symposium on Theoretical Aspects on Computer Science, STACS'87*, LNCS 247, pp 22-39. Springer-Verlag, 1987.

[73] Kamkar Mariam. *Interprocedural Dynamic Slicing with Applications to Debugging and Testing*. Ph.D. Thesis, Department of Computer Science, Linköping University, Linköping, Sweden, 1993.

[74] Krysander M. and M. Nyberg. "Structural Analysis for Fault Diagnosis of DAE Systems Utilizing Graph Theory and MSS Sets." Technical Report LiTH-R-2410, Linköping University, SE-581 83 Linköping, Sweden, Department of Electrical Engineering, 2002.

[75] Leitold Adrien and Hangos M. Katalin. *Structural Solvability of dynamic process models*. Journals of Computers and Chemical Engineering 25 (2001) pp 1633-1646.

[76] Lencevicius Raimondas. *Advanced Debugging Methods*, Kluwer Academic Publishers, August 2000.

[77] Liebermann H. *The Debugging Scandal and What To Do About It*. Communications of the ACM, 40(4), pp 27-29, 1997.

[78] Maffezzoni C., Girelli R. and Lluka P. *Generating efficient computational procedures from declarative models*. Simulation Practice and Theory 4, pp 303-317. 1996.

[79] Mah S.H. Richard. *Chemical Process Structures and Information Flows*. Butterworths Series in Chemical Engineering. Buttherworts 1990.

[80] Mattsson S.E, G. Söderlind. *Index Reduction in Differential-Algebraic Equations Using Dummy Derivatives*. SIAM Journal on Scientific Computing. Vol. 14, 1993, pp. 677-692.

[81] Mattsson S.E, H. Olsson, Elmqvist Hilding. “Dynamic Selection of States in Dymola”. In *Proceedings of Modelica Workshop 2000* (Lund, Sweden, Oct. 23-24), 2000.

[82] Mattsson Sven Erik and Mats Andersson: "The ideas behind Omola." In *Proceedings of the 1992 IEEE Symposium on Computer-Aided Control System Design*, CADCS '92, Napa, California, March 1992.

[83] Mehlhorn K. and S. Näher. *LEDA: A Platform of Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, England, 1999.

[84] Modelica Association. *Modelica – A Unified Object-Oriented Language for Physical Systems Modeling - Tutorial and Design Rationale Version 2.0* (July 10, 2002). Available at http://www.modelica.org

[85] Modelica Association. *Modelica – A Unified Object-Oriented Language for Physical Systems Modeling – Language Specification Version 1.4*. (December 15, 2000). Available at http://www.modelica.org

[86] Müller Tobias. "Practical Investigation of Constraints with Graph Views". In *Proceedings of the Sixth International Conference on Principles and Practice of Constraint Programming* (CP 2000, September 18 - 22, 2000. Singapore), LNCS 1894 Springer-Verlag.

[87] Murota Kazuo. *Matrices and Matroids for System Analysis*, Springer Verlag, Berlin, 2000.

[88] Murota Kazuo. *System Analysis by Graphs and Matroids - Structural Solvability and Controllability*, Springer Verlag, Berlin, 1987

[89] Myers G. *The Art of Software Testing*. John Wiley and Sons, New York NY, 1979.

[90] Nilsson Henrik. *Declarative Debugging for Lazy Functional Languages*. PhD thesis, Department of Computer and Information Science, Linköping University 1998.

[91] Nuutila Esko and Eljas Soisalon-Soininen. *On Finding the Strongly Connected Components in a Directed Graph*. Information Processing Letters 49 (1993) 9-14

[92] Oh M. and C. C. Pantelides. *A Modelling and Simulation Language for Combined Lumped and Distributed Parameter System*. Computers & Chemical Engineering, Vol. 20, Issues 6-7, pp 611-633, June 1996.

[93] Pantelides, C. *The consistent initialization of differential algebraic systems*. SIAM Journal on Scientific and Statistical Computing, 9(2):213--231, March 1988.

[94] Parasoft. “It's a Bug-free World After All.” Technical Paper PS-9710-DV3. 1997.

[95] Petersson Mikael. *Compiling Natural Semantics*. Lecture Notes in Computer Science LNCS 1549. Springer Verlag 1999.

[96] Petzold L. R. A Description of DASSL: A Differential/Algebraic System Solver. In Proceedings of the 10th IMACS World Congress, August 8-13, Montreal, 1982

[97] Pope Alan. *The CORBA Reference Guide: Understanding the Common Object Request Broker Architecture*. Addison Wesley Professional 1997

[98] Ramirez Vicente Rico. *Representation, Analysis and Solution of Conditional Models in an Equation-Based Environment*. PhD Thesis, Carnegie Melon University, Pittsburgh, Pennsylvania, August 1998.

[99] Ramos Juan José, Piera Miquel Àngel and Serra Ignasi. *The use of physical knowledge to guide formula manipulation in system modeling*. Simulation Practice and Theory, Volume 6, Issue 3, 15 March 1998, Pages 243-254.

[100] Reissig Günter, Wade Martinson, Paul Barton. *DAE of index 1 might have an arbitrary high structural index*. SIAM J. Sci. Comput. Vol 21, No.6, pp 1987-1990. 2000.

[101] Robson D., K. H. Nenett, B.J. Cornelius and M. Munro *Approaches to program comprehension*. The Journal of Systems Software. 14(2), pp 79-84, 1991.

[102] Rosenberg, J.B., *How Debuggers Work. Algorithms, Data Structures and Architecture.* Wiley Computer Publishing, John Wiley & Sons Inc. 1996.

[103] Ryder G. Barbara and Frank Tip. "Change Impact Analysis for Object-Oriented Programs." In *Proceeding of ACM Workshop on Program Analysis for Software Tools and Engineering* (PASTE'01), June 18-19, Snowbird, Utah, USA. 2001

[104] Sahlin P. *Modelling and Simulation Methods for Modular Continuous Systems in Buildings.* PhD Thesis. Department of Building Science, Royal Institute of Technology Stockholm, Sweden, May 1996.

[105] Sahlin Per. *NMF Handbook - an Introduction to the Neutral Model Format*. Dept. of Building Sciences, KTH, Stockholm, June 1996.

[106] Sahlin Per., A. Bring, K. Kolsaker. "Future Trends of the Neutral Model Format". In *Proceedings of the Conference on Building Simulation, IBPSA* (Madison, WI, USA, Aug 1995).

[107] Saldamli Levon, Peter Fritzson and Bernhard. Bachmann. “Extending Modelica for Partial Differential Equations”. In *Proceedings of the 2nd International Modelica Conference* (Munich, Gernany, March. 18-29), 2002.

[108] Schneider T., J. Mades, M. Glesner, A. Windisch, W. Ecker. "An Open VHDL-AMS Simulation Framework." In *Proceedings of 2000 IEEE/ACM International Workshop on Behavioral Modeling and Simulation*, (19-20 Oct. 2000, Orlando, FL, USA) pp 89 –94, 2000.

[109] Shahmehri Nahid, Mariam Kamkar and Peter Fritzson. *Usability Criteria for Automated Debugging System*', Journal of Systems and Software, pp 55-70; October 1995.

[110] Shapiro Ehud Y. *Algorithmic Program Debugging*. MIT Press (May). 1982.

[111] Snelting Gregor. and Frank Tip. Understanding Class Hierarchies Using Concept Analysis". ACM Transactions on Programming Languages and Systems, Vol. 22, No.3, May 2000, pp 540-582.

[112] Szilágyi Gyöngyi, Tibor Gyimóthy, Jan Małuszyński. *Static and Dynamic Slicing of Constraint Logic Programs.* The International Journal on Automated Software Engineering. Vol 9, Issue 1, January 2002.

[113] Tarjan, R.E. *Depth First Search and Linear Graph Algorithms.* SIAM Journal of Computing, 1, pp 146-160, 1972.

[114] Tiller Michael. *Introduction to Physical Modeling with Modelica.* Kluwer Academic Publisher 2001.

[115] Tolsma J. and P. Barton. *DAEPACK: An Open Modeling Environment for Legacy Model*. Journal of Industrial & Engineering Chemistry Research, 39(6): 1826-1839, (2000).

[116] Tolsma John E, Jerry Clabaugh, Paul I. Barton. *Abacuss II, Advanced Modeling Environment and Embedded Simulator, Abacuss II Syntax Manual*. Massachusetts Institute

of Technology Chemical Engineering System Research Group. 2002.
Available at http://yoric.mit.edu/abacuss2/syntax.html..

[117] Tolsma John E. Jerry Clabaugh and Paul I. Barton. *Abacuss II. Advanced Modeling Environment and Embedded Simulator*. Massachusetts Institute of Technology Chemical Engineering System Research Group. 2002.
Available at http://yoric.mit.edu/abacuss2/abacuss2.html

[118] Unger, J., A. Kröner, and W. Marquardt. *Structural Analysis of Differential Algebraic Equation Systems – Theory and Applications*. Computers chem. Engng. Vol 19. No. 18. pp. 867-882. 1995.

[119] Uno Takeaki, "A Fast Algorithm for Enumerating Bipartite Perfect Matchings," In *Proceeding of Twelfth Annual International Symposium on Algorithms and Computation* (ISAAC2001). LNCS 2223 , Springer-Verlag, pp. 367-379, 2001.

[120] Uno Takeaki. “Algorithms for Enumerating All Perfect, Maximum and Maximal Matchings in Bipartite Graphs.” In *Proceedings of Eighth Annual International Symposium on Algorithms and Computation (ISAAC '97)*, Singapore, December 17--19, 1997.

[121] Wani M.F. and O.P. Gandhi. *Diagnosability Evaluation of Systems Using Bipartite Graph and Matrix Approach*. Journal of Artificial Intelligence for Engineering Design, Analysis and Manufacturig (2000), 14, pp 193-206.

[122] Weiser M. *Programmers Use Slices when Debugging*, Communications of ACM 25, pp 446-452 (1982).

[123] Wieland Dominik. "*Model-Based Debugging of Java Programs Using Dependencies*", PhD Thesis, Technische Universität Wien, Institute of Information Systems, 2001.

[124] Wilson M. A. *Hierarchical Constraint Logic Programming*. PhD thesis, Dept. of Computer Science, University of Washington, May 1993.

[125] Wilson M. and A. Borning. *Hierarchical constraint logic programming*. Journal of Logic Programming, 16(3/4): 277-319, 1993.

[126] Wolfram Stephen. *The Mathematica Book*. Wolfram Media Inc. (February 1996)

[127] Zeller Andreas and Ralf Hildebrandt. *Simplifying and Isolating Failure-Inducing*. IEEE Transactions on Software Engineering 28(2), pp. 183-200, February 2002.

LINKÖPINGS UNIVERSITET

**Avdelning, institution**
Division, department

Institutionen för datavetenskap

Department of Computer and Information Science

**Datum**
Date

2002-09-20

**Språk**
Language

[ ] Svenska/Swedish
[X] Engelska/English

[ ] _____

**Rapporttyp**
Report category

[X] Licentiatavhandling
[ ] Examensarbete
[ ] C-uppsats
[ ] D-uppsats
[ ] Övrig rapport

_____

**URL för elektronisk version**

**ISBN** 91-7373-382-2

**ISRN** LiU-Tek-Lic-2002:37

**Serietitel och serienummer**
Title of series, numbering

**ISSN** 0280-7971

Linköping Studies in Science and Technology

Thesis No. 964

**Titel**
Title

# Debugging and Structural Analysis of Declarative Equation-Based Languages

**Författare**
Author

Peter Bunus

**Sammanfattning**
Abstract

A significant part of the software development effort is spent on detecting deviations between software implementations and specifications, and subsequently locating the sources of such errors. This thesis illustrates that is possible to identify a significant number of errors during static analysis of declarative object-oriented equation-based modeling languages that are typically used for system modeling and simulation. Detecting anomalies in the source code without actually solving the underlying system of equations provides a significant advantage: a modeling error can be corrected before trying to get the model compiled or embarking on a computationally expensive symbolic or numerical solution process. The overall objective of this work is to demonstrate that debugging based on static analysis techniques can considerably improve the error location and error correcting process when modeling with equation-based languages.

A new method is proposed for debugging of over- and under-constrained systems of equations. The improved approach described in this thesis is to perform the debugging process on the flattened intermediate form of the source code and to use filtering criteria generated from program annotations and from the translation rules. Each time when an error is detected in the intermediate code and the error fixing solution is elaborated, the debugger queries for the original source code before presenting any information to the user. In this way, the user is exposed to the original language source code and not burdened with additional information from the translation process or required to inspect the intermediate code.

We present the design and implementation of debugging kernel prototypes, tightly integrated with the core of the optimizer module of a Modelica compiler, including details of the novel framework required for automatic debugging of equation-based languages.

This thesis establishes that structural static analysis performed on the underlying system of equations from object-oriented mathematical models can effectively be used to statically debug real Modelica programs. Most of our conclusions developed in this thesis are also valid for other equation-based modeling languages.

**Nyckelord**
Keywords

Modelica, Automated debugging, Equation-based languages, Over-constrained systems, Under-constrained systems.

Department of Computer and Information Science
Linköpings universitet

## Linköping Studies in Science and Technology
## Faculty of Arts and Sciences - Licentiate Theses

No 17 **Vojin Plavsic:** Interleaved Processing of Non-Numerical Data Stored on a Cyclic Memory. (Available at: FOA, Box 1165, S-581 11 Linköping, Sweden. FOA Report B30062E)

No 28 **Arne Jönsson, Mikael Patel:** An Interactive Flowcharting Technique for Communicating and Realizing Algorithms, 1984.

No 29 **Johnny Eckerland:** Retargeting of an Incremental Code Generator, 1984.

No 48 **Henrik Nordin:** On the Use of Typical Cases for Knowledge-Based Consultation and Teaching, 1985.

No 52 **Zebo Peng:** Steps Towards the Formalization of Designing VLSI Systems, 1985.

No 60 **Johan Fagerström:** Simulation and Evaluation of Architecture based on Asynchronous Processes, 1985.

No 71 **Jalal Maleki:** ICONStraint, A Dependency Directed Constraint Maintenance System, 1987.

No 72 **Tony Larsson:** On the Specification and Verification of VLSI Systems, 1986.

No 73 **Ola Strömfors:** A Structure Editor for Documents and Programs, 1986.

No 74 **Christos Levcopoulos:** New Results about the Approximation Behavior of the Greedy Triangulation, 1986.

No 104 **Shamsul I. Chowdhury: S**tatistical Expert Systems - a Special Application Area for Knowledge-Based Computer Methodology, 1987.

No 108 **Rober Bilos:** Incremental Scanning and Token-Based Editing, 1987.

No 111 **Hans Block:** SPORT-SORT Sorting Algorithms and Sport Tournaments, 1987.

No 113 **Ralph Rönnquist:** Network and Lattice Based Approaches to the Representation of Knowledge, 1987.

No 118 **Mariam Kamkar, Nahid Shahmehri:** Affect-Chaining in Program Flow Analysis Applied to Queries of Programs, 1987.

No 126 **Dan Strömberg:** Transfer and Distribution of Application Programs, 1987.

No 127 **Kristian Sandahl:** Case Studies in Knowledge Acquisition, Migration and User Acceptance of Expert Systems, 1987.

No 139 **Christer Bäckström:** Reasoning about Interdependent Actions, 1988.

No 140 **Mats Wirén:** On Control Strategies and Incrementality in Unification-Based Chart Parsing, 1988.

No 146 **Johan Hultman:** A Software System for Defining and Controlling Actions in a Mechanical System, 1988.

No 150 **Tim Hansen:** Diagnosing Faults using Knowledge about Malfunctioning Behavior, 1988.

No 165 **Jonas Löwgren:** Supporting Design and Management of Expert System User Interfaces, 1989.

No 166 **Ola Petersson:** On Adaptive Sorting in Sequential and Parallel Models, 1989.

No 174 **Yngve Larsson:** Dynamic Configuration in a Distributed Environment, 1989.

No 177 **Peter Åberg:** Design of a Multiple View Presentation and Interaction Manager, 1989.

No 181 **Henrik Eriksson:** A Study in Domain-Oriented Tool Support for Knowledge Acquisition, 1989.

No 184 **Ivan Rankin:** The Deep Generation of Text in Expert Critiquing Systems, 1989.

No 187 **Simin Nadjm-Tehrani:** Contributions to the Declarative Approach to Debugging Prolog Programs, 1989.

No 189 **Magnus Merkel:** Temporal Information in Natural Language, 1989.

No 196 **Ulf Nilsson:** A Systematic Approach to Abstract Interpretation of Logic Programs, 1989.

No 197 **Staffan Bonnier:** Horn Clause Logic with External Procedures: Towards a Theoretical Framework, 1989.

No 203 **Christer Hansson:** A Prototype System for Logical Reasoning about Time and Action, 1990.

No 212 **Björn Fjellborg:** An Approach to Extraction of Pipeline Structures for VLSI High-Level Synthesis, 1990.

No 230 **Patrick Doherty:** A Three-Valued Approach to Non-Monotonic Reasoning, 1990.

No 237 **Tomas Sokolnicki:** Coaching Partial Plans: An Approach to Knowledge-Based Tutoring, 1990.

No 250 **Lars Strömberg:** Postmortem Debugging of Distributed Systems, 1990.

No 253 **Torbjörn Näslund:** SLDFA-Resolution - Computing Answers for Negative Queries, 1990.

No 260 **Peter D. Holmes:** Using Connectivity Graphs to Support Map-Related Reasoning, 1991.

No 283 **Olof Johansson:** Improving Implementation of Graphical User Interfaces for Object-Oriented Knowledge-Bases, 1991.

No 298 **Rolf G Larsson:** Aktivitetsbaserad kalkylering i ett nytt ekonomisystem, 1991.

No 318 **Lena Strömbäck:** Studies in Extended Unification-Based Formalism for Linguistic Description: An Algorithm for Feature Structures with Disjunction and a Proposal for Flexible Systems, 1992.

No 319 **Mikael Pettersson:** DML-A Language and System for the Generation of Efficient Compilers from Denotational Specification, 1992.

No 326 **Andreas Kågedal:** Logic Programming with External Procedures: an Implementation, 1992.

No 328 **Patrick Lambrix:** Aspects of Version Management of Composite Objects, 1992.

No 333 **Xinli Gu:** Testability Analysis and Improvement in High-Level Synthesis Systems, 1992.

No 335 **Torbjörn Näslund:** On the Role of Evaluations in Iterative Development of Managerial Support Sytems, 1992.

No 348 **Ulf Cederling:** Industrial Software Development - a Case Study, 1992.

No 352 **Magnus Morin:** Predictable Cyclic Computations in Autonomous Systems: A Computational Model and Implementation, 1992.

No 371 **Mehran Noghabai:** Evaluation of Strategic Investments in Information Technology, 1993.

No 378 **Mats Larsson:** A Transformational Approach to Formal Digital System Design, 1993.

No 380 **Johan Ringström:** Compiler Generation for Parallel Languages from Denotational Specifications, 1993.

No 381 **Michael Jansson:** Propagation of Change in an Intelligent Information System, 1993.

No 383 **Jonni Harrius:** An Architecture and a Knowledge Representation Model for Expert Critiquing Systems, 1993.

No 386 **Per Österling:** Symbolic Modelling of the Dynamic Environments of Autonomous Agents, 1993.

No 398 **Johan Boye:** Dependency-based Groudness Analysis of Functional Logic Programs, 1993.

No 402 **Lars Degerstedt:** Tabulated Resolution for Well Founded Semantics, 1993.
No 406 **Anna Moberg:** Satellitkontor - en studie av kommunikationsmönster vid arbete på distans, 1993.
No 414 **Peter Carlsson:** Separation av företagsledning och finansiering - fallstudier av företagsledarutköp ur ett agentteoretiskt perspektiv, 1994.
No 417 **Camilla Sjöström:** Revision och lagreglering - ett historiskt perspektiv, 1994.
No 436 **Cecilia Sjöberg:** Voices in Design: Argumentation in Participatory Development, 1994.
No 437 **Lars Viklund:** Contributions to a High-level Programming Environment for a Scientific Computing, 1994.
No 440 **Peter Loborg:** Error Recovery Support in Manufacturing Control Systems, 1994.
FHS 3/94 **Owen Eriksson:** Informationssystem med verksamhetskvalitet - utvärdering baserat på ett verksamhetsinriktat och samskapande perspektiv, 1994.
FHS 4/94 **Karin Pettersson:** Informationssystemstrukturering, ansvarsfördelning och användarinflytande - En komparativ studie med utgångspunkt i två informationssystemstrategier, 1994.
No 441 **Lars Poignant:** Informationsteknologi och företagsetablering - Effekter på produktivitet och region, 1994.
No 446 **Gustav Fahl:** Object Views of Relational Data in Multidatabase Systems, 1994.
No 450 **Henrik Nilsson:** A Declarative Approach to Debugging for Lazy Functional Languages, 1994.
No 451 **Jonas Lind:** Creditor - Firm Relations: an Interdisciplinary Analysis, 1994.
No 452 **Martin Sköld:** Active Rules based on Object Relational Queries - Efficient Change Monitoring Techniques, 1994.
No 455 **Pär Carlshamre:** A Collaborative Approach to Usability Engineering: Technical Communicators and System Developers in Usability-Oriented Systems Development, 1994.
FHS 5/94 **Stefan Cronholm:** Varför CASE-verktyg i systemutveckling? - En motiv- och konsekvensstudie avseende arbetssätt och arbetsformer, 1994.
No 462 **Mikael Lindvall:** A Study of Traceability in Object-Oriented Systems Development, 1994.
No 463 **Fredrik Nilsson:** Strategi och ekonomisk styrning - En studie av Sandviks förvärv av Bahco Verktyg, 1994.
No 464 **Hans Olsén:** Collage Induction: Proving Properties of Logic Programs by Program Synthesis, 1994.
No 469 **Lars Karlsson:** Specification and Synthesis of Plans Using the Features and Fluents Framework, 1995.
No 473 **Ulf Söderman:** On Conceptual Modelling of Mode Switching Systems, 1995.
No 475 **Choong-ho Yi:** Reasoning about Concurrent Actions in the Trajectory Semantics, 1995.
No 476 **Bo Lagerström:** Successiv resultatavräkning av pågående arbeten. - Fallstudier i tre byggföretag, 1995.
No 478 **Peter Jonsson:** Complexity of State-Variable Planning under Structural Restrictions, 1995.
FHS 7/95 **Anders Avdic:** Arbetsintegrerad systemutveckling med kalkylkprogram, 1995.
No 482 **Eva L Ragnemalm:** Towards Student Modelling through Collaborative Dialogue with a Learning Companion, 1995.
No 488 **Eva Toller:** Contributions to Parallel Multiparadigm Languages: Combining Object-Oriented and Rule-Based Programming, 1995.
No 489 **Erik Stoy:** A Petri Net Based Unified Representation for Hardware/Software Co-Design, 1995.
No 497 **Johan Herber:** Environment Support for Building Structured Mathematical Models, 1995.
No 498 **Stefan Svenberg:** Structure-Driven Derivation of Inter-Lingual Functor-Argument Trees for Multi-Lingual Generation, 1995.
No 503 **Hee-Cheol Kim:** Prediction and Postdiction under Uncertainty, 1995.
FHS 8/95 **Dan Fristedt:** Metoder i användning - mot förbättring av systemutveckling genom situationell metodkunskap och metodanalys, 1995.
FHS 9/95 **Malin Bergvall:** Systemförvaltning i praktiken - en kvalitativ studie avseende centrala begrepp, aktiviteter och ansvarsroller, 1995.
No 513 **Joachim Karlsson:** Towards a Strategy for Software Requirements Selection, 1995.
No 517 **Jakob Axelsson:** Schedulability-Driven Partitioning of Heterogeneous Real-Time Systems, 1995.
No 518 **Göran Forslund:** Toward Cooperative Advice-Giving Systems: The Expert Systems Experience, 1995.
No 522 **Jörgen Andersson:** Bilder av småföretagares ekonomistyrning, 1995.
No 538 **Staffan Flodin:** Efficient Management of Object-Oriented Queries with Late Binding, 1996.
No 545 **Vadim Engelson:** An Approach to Automatic Construction of Graphical User Interfaces for Applications in Scientific Computing, 1996.
No 546 **Magnus Werner :** Multidatabase Integration using Polymorphic Queries and Views, 1996.
FiF-a 1/96 **Mikael Lind:** Affärsprocessinriktad förändringsanalys - utveckling och tillämpning av synsätt och metod, 1996.
No 549 **Jonas Hallberg:** High-Level Synthesis under Local Timing Constraints, 1996.
No 550 **Kristina Larsen:** Förutsättningar och begränsningar för arbete på distans - erfarenheter från fyra svenska företag. 1996.
No 557 **Mikael Johansson:** Quality Functions for Requirements Engineering Methods, 1996.
No 558 **Patrik Nordling:** The Simulation of Rolling Bearing Dynamics on Parallel Computers, 1996.
No 561 **Anders Ekman:** Exploration of Polygonal Environments, 1996.
No 563 **Niclas Andersson:** Compilation of Mathematical Models to Parallel Code, 1996.
No 567 **Johan Jenvald:** Simulation and Data Collection in Battle Training, 1996.
No 575 **Niclas Ohlsson:** Software Quality Engineering by Early Identification of Fault-Prone Modules, 1996.
No 576 **Mikael Ericsson:** Commenting Systems as Design Support—A Wizard-of-Oz Study, 1996.
No 587 **Jörgen Lindström:** Chefers användning av kommunikationsteknik, 1996.
No 589 **Esa Falkenroth:** Data Management in Control Applications - A Proposal Based on Active Database Systems, 1996.
No 591 **Niclas Wahllöf:** A Default Extension to Description Logics and its Applications, 1996.
No 595 **Annika Larsson:** Ekonomisk Styrning och Organisatorisk Passion - ett interaktivt perspektiv, 1997.
No 597 **Ling Lin:** A Value-based Indexing Technique for Time Sequences, 1997.

| | |
|---|---|
| No 598 | **Rego Granlund:** $C^3$Fire - A Microworld Supporting Emergency Management Training, 1997. |
| No 599 | **Peter Ingels:** A Robust Text Processing Technique Applied to Lexical Error Recovery, 1997. |
| No 607 | **Per-Arne Persson:** Toward a Grounded Theory for Support of Command and Control in Military Coalitions, 1997. |
| No 609 | **Jonas S Karlsson:** A Scalable Data Structure for a Parallel Data Server, 1997. |
| FiF-a 4 | **Carita Åbom:** Videomötesteknik i olika affärssituationer - möjligheter och hinder, 1997. |
| FiF-a 6 | **Tommy Wedlund**: Att skapa en företagsanpassad systemutvecklingsmodell - genom rekonstruktion, värdering och vidareutveckling i T50-bolag inom ABB, 1997. |
| No 615 | **Silvia Coradeschi**: A Decision-Mechanism for Reactive and Coordinated Agents, 1997. |
| No 623 | **Jan Ollinen:** Det flexibla kontorets utveckling på Digital - Ett stöd för multiflex? 1997. |
| No 626 | **David Byers:** Towards Estimating Software Testability Using Static Analysis, 1997. |
| No 627 | **Fredrik Eklund:** Declarative Error Diagnosis of GAPLog Programs, 1997. |
| No 629 | **Gunilla Ivefors:** Krigsspel coh Informationsteknik inför en oförutsägbar framtid, 1997**.** |
| No 631 | **Jens-Olof Lindh:** Analysing Traffic Safety from a Case-Based Reasoning Perspective, 1997 |
| No 639 | **Jukka Mäki-Turja:**. Smalltalk - a suitable Real-Time Language, 1997. |
| No 640 | **Juha Takkinen**: CAFE: Towards a Conceptual Model for Information Management in Electronic Mail, 1997. |
| No 643 | **Man Lin**: Formal Analysis of Reactive Rule-based Programs, 1997. |
| No 653 | **Mats Gustafsson**: Bringing Role-Based Access Control to Distributed Systems, 1997. |
| FiF-a 13 | **Boris Karlsson:** Metodanalys för förståelse och utveckling av systemutvecklingsverksamhet. Analys och värdering av systemutvecklingsmodeller och dess användning, 1997. |
| No 674 | **Marcus Bjäreland:** Two Aspects of Automating Logics of Action and Change - Regression and Tractability, 1998. |
| No 676 | **Jan Håkegård**: Hiera rchical Test Architecture and Board-Level Test Controller Synthesis, 1998. |
| No 668 | **Per-Ove Zetterlund**: Normering av svensk redovisning - En studie av tillkomsten av Redovisningsrådets rekommendation om koncernredovisning (RR01:91), 1998. |
| No 675 | **Jimmy Tjäder**: Projektledaren & planen - en studie av projektledning i tre installations- och systemutvecklingsprojekt, 1998. |
| FiF-a 14 | **Ulf Melin**: Informationssystem vid ökad affärs- och processorientering - egenskaper, strategier och utveckling, 1998. |
| No 695 | **Tim Heyer**: COMPASS: Introduction of Formal Methods in Code Development and Inspection, 1998. |
| No 700 | **Patrik Hägglund:** Programming Languages for Computer Algebra, 1998. |
| FiF-a 16 | **Marie-Therese Christiansson:** Inter-organistorisk verksamhetsutveckling - metoder som stöd vid utveckling av partnerskap och informationssystem, 1998. |
| No 712 | **Christina Wennestam:** Information om immateriella resurser. Investeringar i forskning och utveckling samt i personal inom skogsindustrin, 1998. |
| No 719 | **Joakim Gustafsson:** Extending Temporal Action Logic for Ramification and Concurrency, 1998. |
| No 723 | **Henrik André-Jönsson:** Indexing time-series data using text indexing methods, 1999. |
| No 725 | **Erik Larsson:** High-Level Testability Analysis and Enhancement Techniques, 1998. |
| No 730 | **Carl-Johan Westin:** Informationsförsörjning: en fråga om ansvar - aktiviteter och uppdrag i fem stora svenska organisationers operativa informationsförsörjning, 1998. |
| No 731 | **Åse Jansson:** Miljöhänsyn - en del i företags styrning, 1998. |
| No 733 | **Thomas Padron-McCarthy:** Performance-Polymorphic Declarative Queries, 1998. |
| No 734 | **Anders Bäckström:** Värdeskapande kreditgivning - Kreditriskhantering ur ett agentteoretiskt perspektiv, 1998. |
| FiF-a 21 | **Ulf Seigerroth:** Integration av förändringsmetoder - en modell för välgrundad metodintegration, 1999. |
| FiF-a 22 | **Fredrik Öberg:** Object-Oriented Frameworks - A New Strategy for Case Tool Development, 1998. |
| No 737 | **Jonas Mellin:** Predictable Event Monitoring, 1998. |
| No 738 | **Joakim Eriksson:** Specifying and Managing Rules in an Active Real-Time Database System, 1998. |
| FiF-a 25 | **Bengt E W Andersson:** Samverkande informationssystem mellan aktörer i offentliga åtaganden - En teori om aktörsarenor i samverkan om utbyte av information, 1998. |
| No 742 | **Pawel Pietrzak:** Static Incorrectness Diagnosis of CLP (FD), 1999. |
| No 748 | **Tobias Ritzau:** Real-Time Reference Counting in RT-Java, 1999. |
| No 751 | **Anders Ferntoft:** Elektronisk affärskommunikation - kontaktkostnader och kontaktprocesser mellan kunder och leverantörer på producentmarknader,1999. |
| No 752 | **Jo Skåmedal:** Arbete på distans och arbetsformens påverkan på resor och resmönster, 1999. |
| No 753 | **Johan Alvehus:** Mötets metaforer. En studie av berättelser om möten, 1999. |
| No 754 | **Magnus Lindahl:** Bankens villkor i låneavtal vid kreditgivning till högt belånade företagsförvärv: En studie ur ett agentteoretiskt perspektiv, 2000. |
| No 766 | **Martin V. Howard:** Designing dynamic visualizations of temporal data, 1999. |
| No 769 | **Jesper Andersson:** Towards Reactive Software Architectures, 1999. |
| No 775 | **Anders Henriksson:** Unique kernel diagnosis, 1999. |
| FiF-a 30 | **Pär J. Ågerfalk:** Pragmatization of Information Systems - A Theoretical and Methodological Outline, 1999. |
| No 787 | **Charlotte Björkegren:** Learning for the next project - Bearers and barriers in knowledge transfer within an organisation, 1999. |
| No 788 | **Håkan Nilsson:** Informationsteknik som drivkraft i granskningsprocessen - En studie av fyra revisionsbyråer, 2000. |
| No 790 | **Erik Berglund:** Use-Oriented Documentation in Software Development, 1999. |
| No 791 | **Klas Gäre:** Verksamhetsförändringar i samband med IS-införande, 1999. |
| No 800 | **Anders Subotic:** Software Quality Inspection, 1999. |
| No 807 | **Svein Bergum**: Managerial communication in telework, 2000. |

| | |
|---|---|
| No 809 | **Flavius Gruian:** Energy-Aware Design of Digital Systems, 2000. |
| FiF-a 32 | **Karin Hedström:** Kunskapsanvändning och kunskapsutveckling hos verksamhetskonsulter - Erfarenheter från ett FOU-samarbete, 2000. |
| No 808 | **Linda Askenäs:** Affärssystemet - En studie om teknikens aktiva och passiva roll i en organisation, 2000. |
| No 820 | **Jean Paul Meynard:** Control of industrial robots through high-level task programming, 2000. |
| No 823 | **Lars Hult:** Publika Gränsytor - ett designexempel, 2000. |
| No 832 | **Paul Pop:** Scheduling and Communication Synthesis for Distributed Real-Time Systems, 2000. |
| FiF-a 34 | **Göran Hultgren:** Nätverksinriktad Förändringsanalys - perspektiv och metoder som stöd för förståelse och utveckling av affärsrelationer och informationssystem, 2000. |
| No 842 | **Magnus Kald:** The role of management control systems in strategic business units, 2000. |
| No 844 | **Mikael Cäker:** Vad kostar kunden? Modeller för intern redovisning, 2000. |
| FiF-a 37 | **Ewa Braf**: Organisationers kunskapsverksamheter - en kritisk studie av ”knowledge management”, 2000. |
| FiF-a 40 | **Henrik Lindberg:** Webbaserade affärsprocesser - Möjligheter och begränsningar, 2000. |
| FiF-a 41 | **Benneth Christiansson:** Att komponentbasera informationssystem - Vad säger teori och praktik?, 2000. |
| No. 854 | **Ola Pettersson:** Deliberation in a Mobile Robot, 2000. |
| No 863 | **Dan Lawesson**: Towards Behavioral Model Fault Isolation for Object Oriented Control Systems, 2000. |
| No 881 | **Johan Moe:** Execution Tracing of Large Distributed Systems, 2001. |
| No 882 | **Yuxiao Zhao:** XML-based Frameworks for Internet Commerce and an Implementation of B2B e-procurement, 2001. |
| No 890 | **Annika Flycht-Eriksson:** Domain Knowledge Management inInformation-providing Dialogue systems, 2001. |
| Fif-a 47 | **Per-Arne Segerkvis**t: Webbaserade imaginära organisationers samverkansformer, 2001. |
| No 894 | **Stefan Svarén:** Styrning av investeringar i divisionaliserade företag - Ett koncernperspektiv, 2001. |
| No 906 | **Lin Han**: Secure and Scalable E-Service Software Delivery, 2001. |
| No 917 | **Emma Hansson:** Optionsprogram för anställda - en studie av svenska börsföretag, 2001. |
| No 916 | **Susanne Odar:** IT som stöd för strategiska beslut, en studie av datorimplementerade modeller av verksamhet som stöd för beslut om anskaffning av JAS 1982, 2002. |
| Fif-a-49 | **Stefan Holgersson:** IT-system och filtrering av verksamhetskunskap - kvalitetsproblem vid analyser och beslutsfattande som bygger på uppgifter hämtade från polisens IT-system, 2001. |
| Fif-a-51 | **Per Oscarsson:**Informationssäkerhet i verksamheter - begrepp och modeller som stöd för förståelse av informationssäkerhet och dess hantering, 2001. |
| No 919 | **Luis Alejandro Cortes:** A Petri Net Based Modeling and Verification Technique for Real-Time Embedded Systems, 2001. |
| No 915 | **Niklas Sandell:** Redovisning i skuggan av en bankkris - Värdering av fastigheter. 2001. |
| No 931 | **Fredrik Elg:** Ett dynamiskt perspektiv på individuella skillnader av heuristisk kompetens, intelligens, mentala modeller, mål och konfidens i kontroll av mikrovärlden Moro, 2002. |
| No 933 | **Peter Aronsson:** Automatic Parallelization of Simulation Code from Equation Based Simulation Languages, 2002. |
| No 938 | **Bourhane Kadmiry**: Fuzzy Control of Unmanned Helicopter, 2002. |
| No 942 | **Patrik Haslum**: Prediction as a Knowledge Representation Problem: A Case Study in Model Design, 2002. |
| No 956 | **Robert Sevenius:** On the instruments of governance - A law & economics study of capital instruments in limited liability companies, 2002. |
| FiF-a 58 | **Johan Petersson:** Lokala elektroniska marknadsplatser - informationssystem för platsbundna affärer, 2002. |
| No 964) | **Peter Bunus:** Debugging and Structural Analysis of Declarative Equation-Based Languages, 2002. |