Debugging Support for an Optimized Modelica Compiler

Master's thesis

by

Kaj Nyström

Reg nr: LiTH-IDA-EX–03/040–SE

30th May 2003

Debugging Support for an Optimized Modelica Compiler

Master's thesis

performed in **Institution for Computer Science**,
Dept. of Computer and Information Science
at **Linköping university**

by **Kaj Nyström**

Reg nr: LiTH-IDA-EX–03/040–SE

Supervisor: **PhD Student Peter Bunus**
Institution for Computer Science
at Linköping University

Examiner: **Professor Peter Fritzson**
Department of Computer and Information Science
at Linköpings universitet

Linköping, 30th May 2003

Abstract

The need for modeling and simulation in engineering is continuously rising as systems become increasingly complex and physical prototyping becomes too expensive. The thesis aims at optimizing the performance of simulation in terms of lowering the time spent solving the equations constituting the model. This is done by decomposing the equation system into sequentially solvable components, making it possible for the solver to work on many smaller components instead of one large component. The system is also transformed in order to be solvable with a less complex, and faster solver through rewriting of differential equations of higher degree to differential equations of index one.

The second purpose of the thesis is to facilitate debugging of faulty models through an integrated debugger. The debugger detects inconsistent models and presents to the user elaborated alternatives to correcting the model. This is done at the original source code level and not in some intermediate language which tend to be difficult to understand for the user.

Results from testing of the implementation shows improvements in performance for several different problems and also enables solving of problems that would normally require a special high index solver. It has also demonstrated its ability to facilitate debugging, primarily as a fault detector but also as a solution provider in a user friendly maner.

The implementation is included into ModSimPack which constitutes the symbolic and numerical engine of the Open Source Modelica Compiler, a compiler for the Modelica Language.

Keywords: Modelica, Optimizer, Debugger, Compiler

Acknowledgments

First, I would like to thank my supervisor Peter Bunus whos research is the basis of this thesis.
Second, a thanks to Peter Fritzon and the Modelica group at PELAB. This thesis would not even exist without their previous, and current work.
Thanks also goes to the MathModelica team at MathCore and PELAB, especially to Eva-Lena Lengquist for her help with construction of the examples.
Finally, I would like to thank Karin Berg for her encouragement and help on the way.
Linköping 30th May 2003
Kaj Nyström

Contents

1 Introduction to Modelica **1**
1.1 Modeling in Modelica 1
1.2 The Modelica Language 3
1.3 Classes and inheritance 3
1.4 Connections 4
1.5 A Simple Modelica Example 6

2 Introduction to Graph Theory **7**
2.1 Basic Definitions 7
2.2 Definitions for bipartite graphs 9
2.3 Graph theory and equations 10
2.4 Matching in a bipartite graph 11
2.4.1 The maximum matching 12
2.4.2 The maximum weighted matching 13

3 The Open Source Modelica Project **15**
3.1 The Compiler Front-end 15
3.2 ModSimPack 16
3.3 ModSimPack overview 17

4 The Optimizer **19**
4.1 Parsing 19
4.2 Dulmage-Mendelsohn Canonical Decomposition 19
4.2.1 Well-Constrained Components 20

4.2.2 Over-Constrained Components 22
4.2.3 Under-Constrained Components 22
4.3 Index Reduction 23
4.3.1 Differential and Structural Index 23
4.3.2 Differential equation rewriting 24

5 The Debugger **28**
5.1 Debugging Over-Constrained systems 28
5.1.1 Assumptions in Debugging of Over-Constrained Systems 28
5.1.2 Debugging Over-Constrained Systems as Bipartite Graphs 29
5.2 Debugging Under-Constrained systems 36
5.3 Debugging Simultaneous Over and Under-Constrained systems 36

6 Implementation **38**
6.1 Reused Libraries 38
6.2 Architecture and language 39
6.3 Limitations in this implementation 39

7 Conclusions and Future Work **40**
7.1 The Debugger 40
7.2 The Optimizer 40
7.3 Future Work 41

8 Related Work **43**

List of Figures

1.1 Modelica code for the Resistor model 2
1.2 Graphical representation for a simple circuit. 2
1.3 Modelica code for the Pin model 5
1.4 Modelica code for the TwoPin model 5
1.5 Modelica code for the Circuit example 6

2.1 A directed graph. 8
2.2 A subgraph of $\overline{D}$. 9
2.3 Translation of equations into a bipartite graph 11
2.4 Equation system of differential equations transformed to a weighted bipartite graph 12
2.5 Example of a perfect matching in a bipartite graph. 13
2.6 Transformation of Figure 2.5 into a directed graph 13

3.1 From .mof-file to result 17
3.2 The back-end of the compiler - ModSimPack 18

4.1 Dulmage and Mendelsohn canonical decomposition with only well-constrained components 21
4.2 Dulmage and Mendelsohn canonical decomposition with over- and under-constrained components 22
4.3 The planar pendulum 24
4.4 The bipartite graph representing equations 4.4, 4.5 and 4.6 25
4.5 A maximum weighted perfect matching for the pendulum example 26

4.6 The maximum weighted matching with cardinality 2 26
4.7 Bipartite graph for the pendulum example with equation three differentiated once 27

5.1 A model with two resistors and an AC-source 30
5.2 Modelica code for the Resistor model with an extra equation 31
5.3 Over-constrained bipartite graph 32
5.4 A maximum cardinality matching of an over-constrained graph 32
5.5 Well-constrained subgraph 33
5.6 Over-constrained subgraphs 34
5.7 Equationsets to be considered for removal 35

7.1 Performance when solving optimized problem compared to solving same problem without optimization 41

Chapter 1

Introduction to Modelica

This section will give a short introduction to Modelica and simulation in general.

1.1 Modeling in Modelica

Modeling has been used for centuries in one form or another. Before the invention of computers, engineers built physical scaled prototypes to test the principles of their construction before they built a full scale version. Modern modeling software provides even better ways of testing and simulating complex processes spanning multiple physical domains (electrical, mechanical, biological etc.). Computerized modeling makes it possible to mathematically solve the complex construction related problems, which is cheaper and less time consuming than physical modeling.

The first step in simulating a system is to construct a model of the system. Most modern modeling systems includes a graphical model editor in which the user can place different types of predefined components such as resistors, bearings, inertial masses and PID-regulators. All of these components are actually collections of equations, variables and associated graphical representations. In this thesis the Modelica language is used for describing this representation. Furthermore, focus is primarily on the

MathModelica simulation environment from MathCore when we talk about simulation environments.

The common resistor in the MathModelica simulation environment will be our first example. It consists of one variable and one equation as shown in Figure 1.1 and the graphical representation, shown in Figure 1.2 inserted into a small circuit.

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

Figure 1.1: Modelica code for the Resistor model

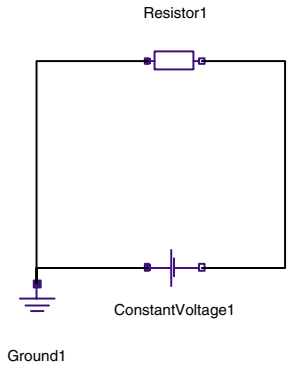

Figure 1.2: Graphical representation for a simple circuit.

Obviously, the different components have to be connected so that the equations they represent can be related to one another. This is done by using connectors, graphically represented as lines. More about these in Section 1.4.

Instead of, or in addition to using a model editor, it is also frequently

possible to define the components and connectors directly into the modeling systems own language.

After having defined our model and given it some parameters, for example the resistance R of our component, we want to get some results from our simulation. The model is now submitted to a compiler which compiles and executes the resulting simulation. A result is produced in which values for all variables are available distributed along a time axis in a diagram. In this thesis, the focus is on the compilation process and in particular the optimization and debugging processes in the the Open Source Modelica compiler. A description of this compiler can be found in [Fritzon et al., 1998].

1.2 The Modelica Language

The aim of modeling languages, such as the Modelica language is to express properties of objects and the relations between them. Modelica uses an object oriented approach to this problem in the same way that other object oriented languages like C++ and Java do. However, Modelica is different from most well-known object oriented languages in some aspects. Modelica is declarative instead of imperative, which implies that the language semantics is different from the semantics.

Modelica is also acausal, meaning that the computation does not depend on the order in which the equation are stated. This gives Modelica components great potential for reusability since they come with a well-defined interface (connectors) towards other components. No data-flow considerations has to be taken when integrating a new component into an existing model.

1.3 Classes and inheritance

Just as Java and C++, Modelica uses classes to define the structure of objects. A Modelica class consists of definition block where variables and parameters are defined and inheritance is specified. Also included in a Modelica class is a block of equations to define the behavior of this compo-

nent. An example of a Modelica class is found in Figure 1.1 and another one in Figure 1.4.

Modelica implements inheritance in a fairly straightforward way. Especially Java programmers can easily recognize the `extends TwoPin` statement in the `Resistor` example in figure 1.1. This means that `Resistor` is an extension of the `TwoPin` model. The Resistor class thus inherits all of `TwoPin`'s equations and variables; they can now be considered to be equations and variables of the `Resistor` model.

The similar concept of subtyping is also important in Modelica. If class A is a subtype of class B, this means that class A contains all public variables and equations of class B. A more thorough description of subtyping can be found in [Abadi and Cardelli, 1996].

1.4 Connections

Connections between components in Modelica is expressed with special `connect` equations, as couplings between variables. The statement "`connect`(v1,v2)" connects variable v1 to variable v2. The variable v1 and v2 are called connectors. The connectors should contain everything needed to describe the interaction with another connector. Let us consider an example: The connectors in the `TwoPin` model are of the type Pin which are defined as shown in Figure 1.3. The `TwoPin` component uses the connector Pin to be able to model any electrical component with two pins (see Figure 1.4). The `TwoPin` model has two connectors of type Pin, a voltage over its pins and a current going through it. The equations in the model are easily derived from Ohm's and Kirchoff's laws:

- The difference in voltage between the two pins is the voltage over the component.

- The current coming in on one pin is the same as the current going out of the other pin.

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

Figure 1.3: Modelica code for the Pin model

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

Figure 1.4: Modelica code for the TwoPin model

1.5 A Simple Modelica Example

To conclude this short introduction to Modelica, we show in Figure 1.5 the Modelica code for a simple example with our previously defined resistor connected to an AC voltage source. The graphical representation is found in Figure 1.2

```
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 1.5: Modelica code for the Circuit example

Chapter 2

Introduction to Graph Theory

In this section, a short introduction to general graph theory, upon which this thesis relies is given. We also define general concepts and describe some algorithms of great importance to the Open Source Modelica Compiler.

2.1 Basic Definitions

Definition 1 *An undirected graph $G = \{V, E\}$ consists of a set of vertices (or nodes) V, and a set of unordered pairs of vertices E called undirected edges.*

Definition 2 *A directed graph $D = \{V, E\}$ consists of a set of vertices (or nodes) V, and a set of ordered pairs of vertices E called directed edges. It is normally denoted $\overline{D} = \{V, \overline{E}\}$.*

In Figure 2.1 we show a simple example of the directed graph: $\overline{D} = \{\{1, 2, 3\}, \{\{1, 2\}, \{1, 3\}\{3, 2\}\}\}$

Definition 3 *Let G be a graph $G = \{V, E\}$ and $u, v \in V$. u and v are adjacent if they are joined by an edge $e \in E$. Furthermore, both u and v*

are incident with e and e is incident with u and v.

Definition 4 *A path of length q (with cardinality q) is a sequence of edges such that for every consecutive pair of edges* $\{e_1 = \{v_0, v_1\}, e_2 = \{v_2, v_3\}\}$ *it holds that* $v_1 = v_2$*. A path from vertex u to vertex v is denoted* $u \xrightarrow{*} v$*.*

In the graph in Figure 2.1, the path $\{1, 2\}, \{2, 4\}$ is a path with cardinality two.

Definition 5 *A graph is called connected if, for all pairs of vertices* $\{i, j\}$*, there exists a path from i to j.*

Definition 6 *A circuit is a path whose endpoints coincide.*

Definition 7 *A subgraph* $S = \{V_1, E_1\}$ *of the graph* $G = \{V, E\}$ *is a graph whose vertices* V_1 *are elements of V and whose arcs* E_1 *are those arcs of G which have two endpoints in* V_1*.*

In Figure 2.2 we show a simple example of the directed subgraph D_{sub} of $\overline{D}$: $\overline{D_{sub}} = \{\{1, 2, 3, 4\}, \{\{1, 2\}, \{1, 3\}, \{2, 4\}, \{3, 2\}, \{4, 3\}\}\}$

Definition 8 *Let* $G = \{V, E\}$ *be an undirected graph where* $V = \{v_0, v_1, ...v_p\}$ *and* $E = \{e_0, e_1, ..., e_q\}$*. The incidence matrix of G is a* $p \times q$ *matrix with* $(v, e) = 1$ *iff vertex v is incident upon edge e and 0 otherwise.*

The incidence matrix of $\overline{D}$ is

$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

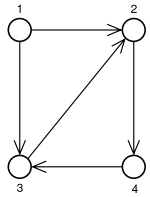

Figure 2.1: A directed graph.

2.2 Definitions for bipartite graphs

Definition 9 *A matching M_G^k for a graph G is a set of k edges from G where no two edges have a common vertex.*

An example matching M for the directed graph $\overline{D}$ is the edge $\{3, 4\}$.

Definition 10 *A maximum cardinality matching (sometimes refered to only as a maximum matching) M_G^{max} for a graph G is a matching with the maximum number of edges.*

Definition 11 *A vertex v is said to be covered by a matching M if some edge in M is incident with v. An uncovered vertex is called a free vertex.*

The vertices 1 and 2 are free in the matching M for $\overline{D}$.

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

Definition 13 *A perfect matching for G is a matching for a graph G that covers all vertices of G.*

Definition 14 *An alternating path P in a graph G with a matching M_G is a path that contains alternating free and covered edges.*

Definition 15 *A feasible path is an alternating path whose end vertices are not covered by matching edges outside the end vertices.*

The path $\{1, 2\}, \{2, 4\}, \{4, 3\}$ is a feasible path for M over $\overline{D}$.

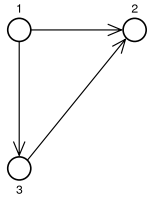

Figure 2.2: A subgraph of $\overline{D}$.

Definition 16 *A partial order relation between two directed graphs G_1, G_2 is defined as $G_1 \prec G_2 \Leftrightarrow v_1 \xrightarrow{*} v_2$ for some $v_1 \in \overline{G_1}$ and $v_2 \in \overline{G_2}$.*

Definition 17 *A directed graph $D = \{V, E\}$ is strongly connected if $u \xrightarrow{*} v$ and $v \xrightarrow{*} u$.*

Definition 18 *A bipartite graph G is an ordered set $G = \{V_1, V_2, E\}$ such that all elements of V_1 and V_2 are vertices of G, $V_1 \cap V_2 = \emptyset$ and $E = \{\{x, y\}; x \in V_1, y \in V_1\}$ and the elements of E are edges of G.*

2.3 Graph theory and equations

We will now explain the relation between the graph theory presented earlier in this chapter, with emphasis on bipartite graphs, and the set of equations produced by the front-end of the compiler that we work with in this thesis.

The main idea as presented in [Bunus, 2002] is, somewhat simplified that symbolic manipulation of equations is not easily done in most programming languages. Especially not in those suited for compiler construction. Graphs and matrices on the other hand are a lot easier to both examine and manipulate and there are plenty of code libraries that can be reused for this purpose.

The principle when converting a set of equations to a bipartite graph is to let each equation in the set represent a node in one of the two sets of nodes, according to Definition 18. The other set of nodes will represent the variables present in the set of equations. Let us look at a simple example: The equations below will be translated into the graph in Figure 2.3.

$$\text{eq1:} x + y = 3 \tag{2.1}$$
$$\text{eq2:} x - y = 1 \tag{2.2}$$
$$\text{eq3:} y + z = 5 \tag{2.3}$$

A few observations can be made. If the two sets of vertices in the graph consists of an equal number of vertices, there are just as many equations as there are variables in the equation set. If there also exists a perfect matching for the graph, this implies that the system is structurally non-singular. If no perfect matching exists, a problem will arise. This typically

means that something is wrong in the equation system. We shall investigate this further in Chapter 5.

For reasons that will become apparent in Chapter 5, we need to treat differential equations somewhat differently from ordinary equations. Depending on the degree of the differentiation of each variable in each equation, we shall assign weights to the edges in the graph. If an equation contains for example the variable a'' (a differentiated two times), an edge should be created from that equation node to the variable node for a and assign to the edge the weight 2. An example equation system is shown below and the corresponding graph is depicted in Figure 2.4.

$$a'' + b' - c = 0 \tag{2.4}$$
$$c' - d = 5 \tag{2.5}$$
$$b + d'' = -2 \tag{2.6}$$
$$-c' + d = 1 \tag{2.7}$$

2.4 Matching in a bipartite graph

Matching is, as hinted above, essential in order to determine some properties of a bipartite graph and thus also of the equation system. Especially the maximum matching can provide us with some interesting information. The maximum cardinality or perfect matchings are not unique but it is not important which maximum cardinality matching we choose to work with.

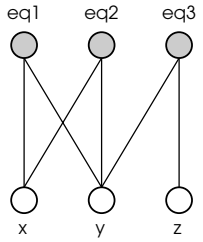

Figure 2.3: Translation of equations into a bipartite graph

2.4.1 The maximum matching

In order to find a maximum matching, we use a simplified, but still as fast algorithm as the one proposed by [Hopcroft and Karp, 1973] which takes $O(n^{5/2})$ time and $O(nm)$ memory, where n is the number of of vertices and m is the number of edges in the graph. The algorithm works by finding feasible paths in the graph and changing the matching status of them as follows:

1. Find a feasible path in the directed graph.
2. If an edge in a feasible path is not in the matching, it is added to the matching.
3. If an edge in a feasible path is in the matching, it is removed from the matching.

This is repeated until no more feasible paths can be found. For every feasible path found, the cardinality of the matching will increase by one. When the maximal matching has been found, there can be no more feasible paths in the graph.

An example of a maximum matching which is also perfect matching is shown in Figure 2.5. Of central interest in this thesis is a special transformation of a perfect matching M for an undirected graph G into a directed graph D. This transformation is done by exchanging all edges in M with

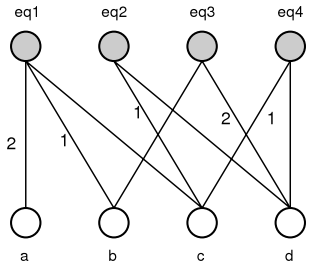

Figure 2.4: Equation system of differential equations transformed to a weighted bipartite graph

bidirectional edges in D. All other edges in G are replaced with edges in D, oriented from the equations to the variables. Transformation of the graph in Figure 2.5 will result in the graph shown in Figure 2.6. This kind of transformed directed graph is the starting point of the Dulmage and Mendelsohn canonical decomposition described in Section 4.2.

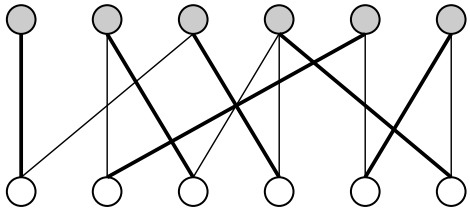

Figure 2.5: Example of a perfect matching in a bipartite graph.

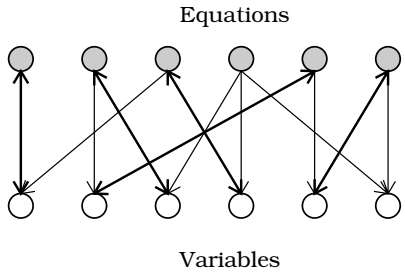

Figure 2.6: Transformation of Figure 2.5 into a directed graph

2.4.2 The maximum weighted matching

Not only do we need to find a maximum matching but we also, at one point in the optimization, need to find the maximum weighted matching of a weighted graph. A weighted graph is a graph where every edge has been assigned a number called weight. The maximum weighted matching of a graph is the matching where the sum of all weights of edges in the matching has its maximum value.

There exists good algorithms for finding all maximum matchings of a graph, for example by [Fukuda and Matsui, 1994] whose algorithm takes $O(n^{1/2}m + mN)$ time and by [Uno, 2001] who has managed to reduce the complexity to $O(n)$ in time and $O(\log n)$ per perfect matching.

Chapter 3

The Open Source Modelica Project

This section will give an overview of the compilation process and explain how the equations which we will process later are extracted from the Modelica code and also how they are treated after our manipulation.

3.1 The Compiler Front-end

This thesis focuses on the back-end of the compiler since that is where the optimizer belongs. It is however necessary to get to know the front-end in order to put the back-end into perspective and to understand some of the problems we will encounter later. Readers who want a more thorough description are encouraged to read [Fritzon et al., 1998].

Most of the front-end is specified in RML [Pettersson, 1995], a language that provides a simple and straightforward way to specify rules and data types in natural semantics. This specification in RML can be compiled into an executable that can interpret or translate any file in the described language. The language is in this case the Modelica language. The RML specification for the Modelica Language is described in [Kågedal and Fritzon, 1998].

After the initial parsing, various operations are performed. The classes are instantiated, creating the desired objects, typechecking is performed and algorithms in the Modelica code are interpreted. An Abstract Syntax Tree (AST) is also created from from the code. This is because it is more convenient to perform necessary operations on the AST than directly on the Modelica source code.

After these operations are performed, the AST is passed through a module that translates it into equations. These equations put together constitutes the Modelica flattened form, described in [Kågedal and Fritzon, 1998]. The main objective of the front-end is to dump them into a file denoted as a ".mof-file". This is where the back-end comes into play.

3.2 ModSimPack

This description will go into more detail than the description of the front-end since this is where the optimizer resides. The back-end, the compilers symbolic and numerical engine has been given its own name: ModSimPack. This is because it is not only a part of the Open Source Modelica Compiler. Due to its highly modular design, it can also be used as a stand-alone application for solving a great variety of mathematical problems. ModSimPack can also be used as a symbolical and numerical API with other applications.

The purpose of the front-end is primarily to produce a set of flattened equations and put them in a .mof-file. In short, these equations are treated by the back-end of the compiler and a file with c-code is generated. This code can then be compiled with a traditional c-compiler such as for instance gcc. The executable generated by the c-compiler can then be executed and results from the simulation will be produced.

An overview of ModSimPack is presented in Figure 3.2. The main objective of the back-end is thus to produce c-code. This c-code generation is done in a series of steps described in the following sections.

3.3 ModSimPack overview

ModSimPack can receive input in two formats. The primary is the Modelica Flattened Form ([Kågedal and Fritzon, 1998]), which is the format produced by the front-end of the compiler. ModSimPack can however also accept input in the GML file format ([Himsolt, 1996a] and [Himsolt, 1996b]), normally used to define graphs.

The parsing of a mof-file is conducted in a fairly straightforward way, inserting the equations into an internal representation. This internal representation is then used by the optimizer and debugger to perform its various tasks, described in Chapter 5 and 4. The actions taken are pretty much the same if the input is in GML format. The exception is that the input does not have to be translated into a graph since it is already a graph.

After the equation system has been optimized and, if necessary debugged it is passed on to a solver. These solvers are external programs, written in Fortran or C which produce results for requested variables. Depending on the type of equations different solvers are used. Examples of these solvers are LAPACK [Anderson et al., 1990], ODEPACK [Hindmarsh, 1983] and DASSL [Petzold, 1983]. For a general description of these solvers the reader is refered to [Health, 2002].

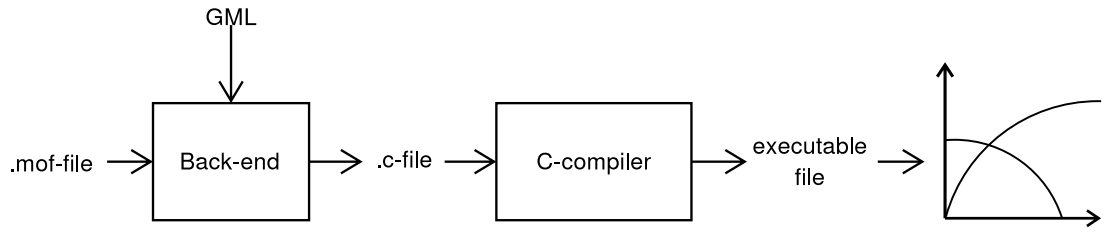

Figure 3.1: From .mof-file to result

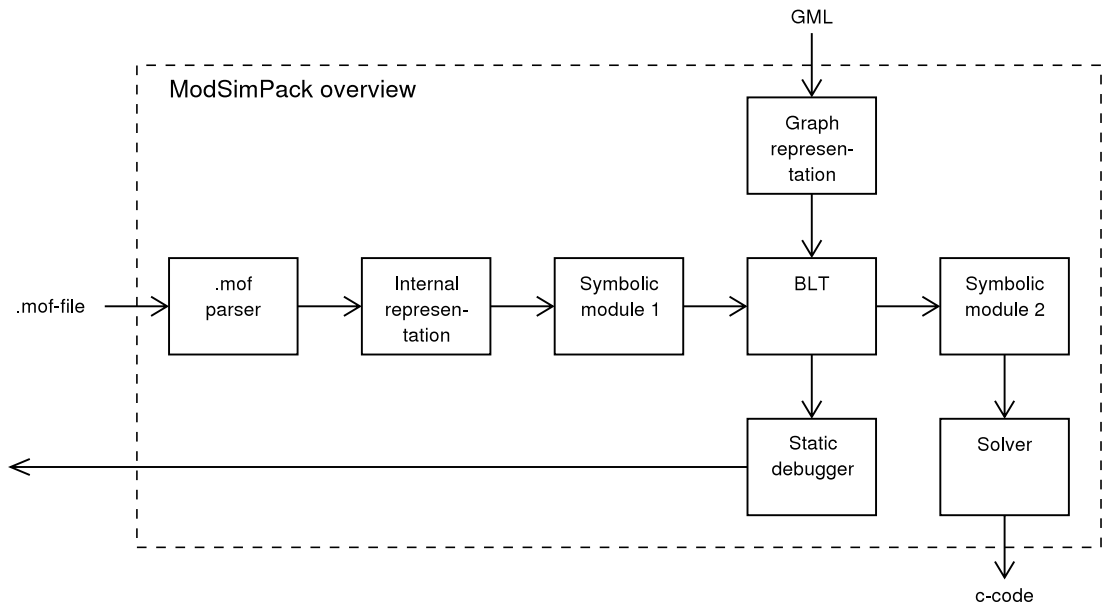

Figure 3.2: The back-end of the compiler - ModSimPack

Chapter 4

The Optimizer

Although the debugger is integrated with the optimizer, we will here focus on the optimizer parts. The debugging parts are described in chapter 5.

4.1 Parsing

The input to the optimizer is, as has been stated in Section 3.1 the flattened form of the Modelica source code. This flattened form consists of variable-, parameter- and constant declarations and a set of equations. This form is easily parsed into an intermediate representation of the model. This model is designed to ensure easy symbolic manipulation of the equations, which is needed later.

4.2 Dulmage-Mendelsohn Canonical Decomposition

The decomposition of the system is the key to both the debugging and to the optimizing of the system. The algorithm is due to
[Dulmage and Mendelsohn, 1963] and its task is to partition any maximum matching of a graph into three distinct components: an under-constrained

component, a well-constrained component and an over-constrained component. The well-constrained components may in turn consist of many different components of their type.

The main objective of the Dulmage and Mendelsohn canonical decomposition is to partition our problem into many smaller problems. This will give us greater computational efficiency since it is easier to solve for example two equation systems with two equations each than to solve one equation system with four equations in it. The decomposition also aids debugging of the system since it can identify partitions of equations that can cause problems in the system.

It is worth noting that the matching from which the algorithm starts is not generally unique. The canonical decomposition is however. The three different components generated have quite different properties which we will now take a closer look at.

4.2.1 Well-Constrained Components

The well-constrained component determined in the decomposition has an equal number of equations and variables. This means that no error has been detected so far and that the system is structurally sound. This component can now be decomposed further by using the strong components algorithm described in [Tarjan, 1972]. This algorithm aims in our case to find cycles in the bipartite graph, which is equal to finding blocks of equations that can be solved independently. An explanation of the decomposition, including the strong components algorithm is given in [Bunus, 2002].

The Dulmage and Mendelsohn Canonical Decomposition results in the Block Lower Triangular form (BLT form) of the incidence matrix. In Figure 4.1, the decomposition results in only well-constrained components, i.e. square blocks in the BLT form. All the blocks consists of an equal number of variables and equation (all the blocks are squares). We can further say that block one, three and five to ten all consist of one variable and one equation, that block two consists of three variables and equations and finally that block four consists of seven equations and seven variables.

The case when all components are well-constrained is the ideal case, since this means that no problem has been found and that we can proceed to the next step in the compilation.

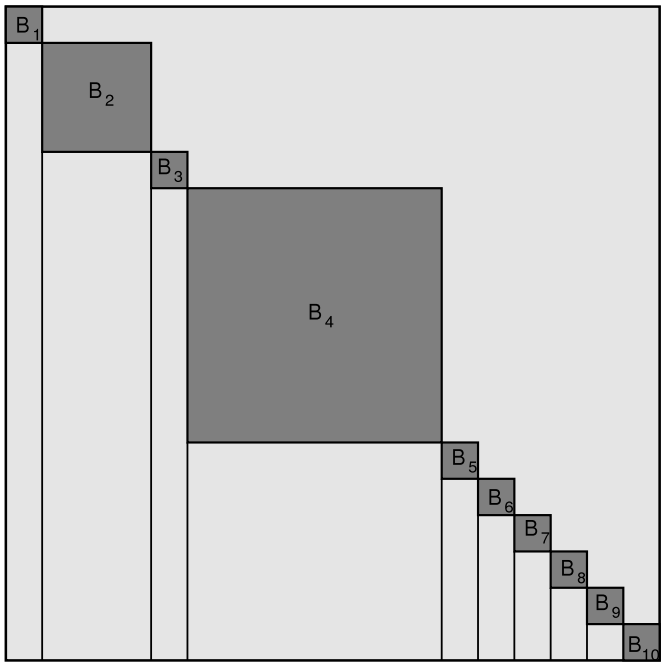

Figure 4.1: Dulmage and Mendelsohn canonical decomposition with only well-constrained components

4.2.2 Over-Constrained Components

The over-constrained component has a greater number of equations than variables. This means that we have redundant or contradictory equations, in which case no solution to the system can be produced, or there is a variable missing in one or more of the equations. The later is however less likely. The optimizer cannot do anything more about this component, it has to be passed to the debugger. The debugger's treatment of over-constrained equations is described in Section 5.1. Figure 4.2 shows the BLT-form for a system with an over-constrained component (component number one).

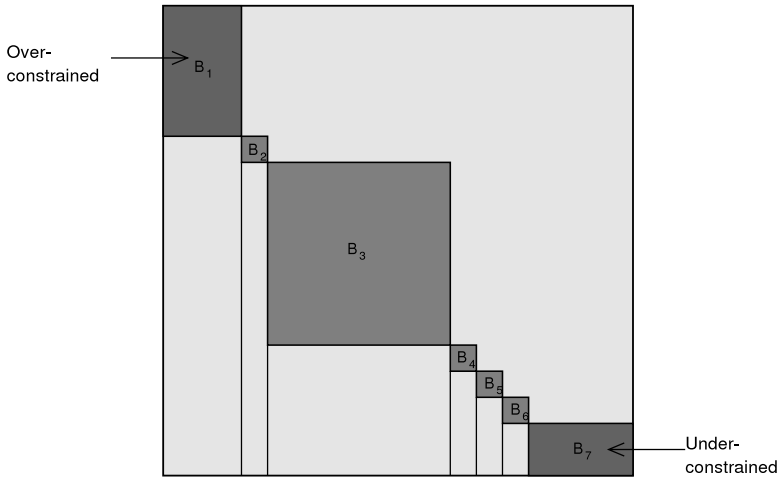

Figure 4.2: Dulmage and Mendelsohn canonical decomposition with over- and under-constrained components

4.2.3 Under-Constrained Components

The under-constrained component on the other hand has a greater number of variables than the number of equations. This also means that there is a problem with the equation system. Typically one equation is missing or an extra variable has been introduced. The debugger's treatment of under-constrained equations is described in Section 5.2. Figure 4.2 shows the

BLT-form for a system with an under-constrained component (component number seven).

4.3 Index Reduction

The next step in the compilation phase is to take care of Differential Algebraic Equations (DAEs) with a high differential index. A high differential index for a system demands specifically designed higher-index solvers which are not as fast as ordinary solvers. To avoid the need for these solvers, we shall instead use symbolic methods to reduce the index and apply a normal DAE index one solver instead.

4.3.1 Differential and Structural Index

The differential index for an equation system is defined as follows:

Definition 19 *The differential index of a system of DAEs 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.*

$$\begin{aligned} F(u, y, \dot{y}) &= 0 && (4.1)\\ \frac{dF}{du}(u, y, \ddot{y}) &= 0 && (4.2)\\ &\Downarrow \\ \frac{dF^d}{du}(u, y, y^{d+1}) &= 0 && (4.3) \end{aligned}$$

The differential index for a given system is not easy to compute. This is why an approximation, called the structural index is used instead. Below is a definition from [Fegery and Barton, 1998], using the algorithm described in [Pantelides, 1988].

Definition 20 *The structural index i_{str} is the minimum number of times that any subset of equations in the DAE is differentiated using Pantelides algorithm.*

According to the algorithm in [Murota, 2000], the structural index can be computed as the difference between the sum of weights in the maximum-minus-one weighted matching and the sum of weights in the maximum matching, plus one. In mathemethical notation:
$i_{str}(G) = maxM_G^{n-1} - maxM_G^n + 1$

An explained example of computation of the structural index is found in Section 4.3.2. Computation of the structural index can be reduced to finding the maximum weighted matching with maximum cardinality and the maximum weighted matching with maximum-1 cardinality in the bipartite graph representing the system of differential equations, as described in [Bunus, 2002].

4.3.2 Differential equation rewriting

Once the structural index is computed, rewriting of the equation to an index of one is quite easy, using diffrentiation rules. We shall here demonstrate both computation of the structural index and the rewriting with the classical pendulum as an example. The example is described in [Pantelides, 1988] and also in [Bunus, 2002]. The system consists of a planar pendulum with a body, suspended by a rigid rod as pictured in Figure 4.3.

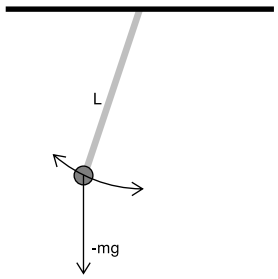

Figure 4.3: The planar pendulum

Newtons second law will give us the differential equations below, which

along with a geometrical constraint will give us the equations for the system.

$$m\ddot{x} = -\frac{x}{L}F \tag{4.4}$$

$$m\ddot{y} = -\frac{x}{L}F - mg \tag{4.5}$$

$$x^2 + y^2 = L^2 \tag{4.6}$$

Direct solving with an index one solver is not possible since equation 4.4 and 4.5 contain high order derivates and equation 4.6 does not. Differentiating equation 4.6 two times however will give us the system below which is suitable for an index one solver. Therefore the differential index of the system is two.

$$m\ddot{x} = -\frac{x}{L}F \tag{4.7}$$

$$m\ddot{y} = -\frac{x}{L}F - mg \tag{4.8}$$

$$x^2 + y^2 = L^2 \tag{4.9}$$

$$2x\dot{x} + 2y\dot{y} = 0 \tag{4.10}$$

$$2\dot{x}\dot{x} + 2x\ddot{x} + 2y\dot{y}\dot{y} + 2y\ddot{y} = 0 \tag{4.11}$$

Structural index computation will also tell us which equation to differentiate in the following manor. The equation system transformed into its bipartite graph form is pictured in Figure 4.4. The maximum weighted

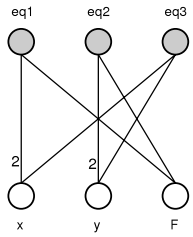

Figure 4.4: The bipartite graph representing equations 4.4, 4.5 and 4.6

perfect matching, pictured in Figure 4.5 has a value of 2. The maximum

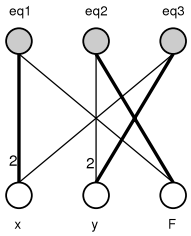

Figure 4.5: A maximum weighted perfect matching for the pendulum example

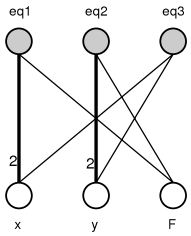

Figure 4.6: The maximum weighted matching with cardinality 2

weighted matching with cardinality 2, shown in Figure 4.6 has a value of 4 which gives us a structural index of $4 - 2 + 1 = 3$ which corresponds to the differential index. By differentiating equation three once, we obtain the graph in Figure 4.7 which has structural index of two. Differentiating once more will give us a structural index of one, which is exactly what is intended.

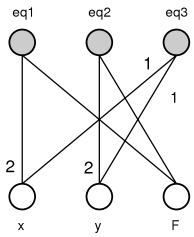

Figure 4.7: Bipartite graph for the pendulum example with equation three differentiated once

Chapter 5

The Debugger

This chapter will present the debugger. This debugger is not a stand-alone debugger like gdb or jdb, but an integrated part of the optimizer. This is because it takes advantage of some of the operations performed in the optimizer, most notably the Dulmage and Mendelsohn canonical decomposition.

5.1 Debugging Over-Constrained systems

As stated in Section 4.2.2 an over-constrained system is generated when the number of equations is greater than the number of variables. Ideally, the debugger should point out, or even correct the extra equation or the missing variable. This is, however, very difficult to do without human interaction. We shall therefore concentrate on facilitating and minimizing human interaction needed when debugging a faulty model.

5.1.1 Assumptions in Debugging of Over-Constrained Systems

As shown in Section 4.2, the Dulmage and Mendelsohn canonical decomposition gives us the over-constrained parts for a given system. This might

still be be quite a few equations though but there are ways of eliminating some of them using a few basic assumptions.

- *Equations generated by a connect statement are not redundant.* These equations are very rigid in the aspect that they are almost always necessary to keep the model connected. Therefore they are unlikely to be redundant or faulty.

- *If an equation is associated with many other equations, then that equation is unlikely to cause fault.* Removal of an equation associated with many other equations will affect the associated equations as well, which is generally not desirable.

If the structural analysis combined with these assumptions can not determine exactly where the error is located, the user must be asked to participate in the debugging process.

5.1.2 Debugging Over-Constrained Systems as Bipartite Graphs

The objective here is to find the set of equations that can safely be considered for removal. This elimination should make the system well-constrained with as little human interaction as possible. In order to accomplish this, we must first define what a safe equation is. This definition is from [Bunus, 2002].

Definition 21 *Any equation node v is safe of the over-constraining subgraph O_G^{k+} is safe for removal if by removing that equation and the corresponding incident edges $inc_E(v)$ the remaining undirected graph is connected.*

Thus, we have to find the equations whose removal will not affect the connectivity of the graph. In order to do this the algorithm described in [Bunus, 2002] will be applied. It makes use of both algorithms for enumerating perfect matchings from [Fukuda and Matsui, 1994] or the improved algorithm from [Uno, 2001] and the strong components algorithm by [Tarjan, 1972].

After computing the safe set of equations, there will be a number of sets of equations that are considered safe for removal. The number of sets will be equal to the number of over-constraining equations.

We can further reduce these sets by making use of language semantics, for example the `connect` statement i Modelica. We can then order the safe equations by their number of incident edges so that the equation with the the fewest number of incident edges is presented as the most likely for removal.

The remaining sets of equations can be permutated to form valid sets to be considered for elimination. If this results in only one set, the debugger can eliminate these equations and proceed with the compilation. If the number of sets however is greater than one, human interaction is required to choose a set of equations that should be eliminated.

As an example if over-constrained debugging we shall consider the example system in Figure 5.1. An extra equation will now be introduced into the resistor model in order to show how the debugger handles the problem. The faulty resistor class will be as seen in Figure 5.2. The flattened

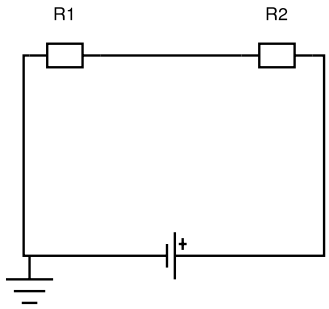

Figure 5.1: A model with two resistors and an AC-source

equations for the system in Figure 5.1 are shown in table 5.1.

Since there are two components of the resistor model, two extra equations have been introduced. In the table above, they are listed as eqn 5 and eqn 10. We will now explain how the debugger would treat this example. The bipartite graph corresponding to the flattened equations is shown in Figure 5.3. A perfect matching cannot exists since the number of equa-

```
class Resistor
  extends TwoPin;
  parameter Real R;
equation
  v = R * i;
  i = 2;
end Resistor
```

Figure 5.2: Modelica code for the **Resistor** model with an extra equation

eqn 1	R1.v = -R1.n.v + R1.p.v	var 1	R1.p.v
eqn 2	0 = R1.n.i + R1.p.i	var 2	R1.p.i
eqn 3	R1.i = R1.p.i	var 3	R1.n.v
eqn 4	R1.i R1.R = R1.v	var 4	R1.n.i
eqn 5	R1.i = 2	var 5	R1.v
eqn 6	R2.v = -R2.n.v + R2.p.v	var 6	R1.i
eqn 7	0 = R2.n.i + R2.p.i	var 7	R2.p.v
eqn 8	R2.i = R2.p.i	var 8	R2.p.i
eqn 9	R2.i R2.R = R2.v	var 9	R2.n.v
eqn 10	R2.i = 2	var 10	R2.n.i
eqn 11	G1.p.v = 0	var 11	R2.v
eqn 12	AC.v = -AC.n.v + AC.p.v	var 12	R2.i
eqn 13	AC.p.i = AC.i	var 13	AC.p.v
eqn 14	0 = AC.n.i + AC.p.i	var 14	AC.p.i
eqn 15	AC.v = AC.VA * sin[2*time*AC.f*AC.Pi]	var 15	AC.n.v
eqn 16	AC.p.v = R2.n.v	var 16	AC.n.i
eqn 17	AC.n.v = R1.p.v	var 17	AC.v
eqn 18	R1.n.v = R2.p.v	var 18	AC.i
eqn 19	AC.p.i + R2.n.i = 0	var 19	G.p.v
eqn 20	R2.p.i + R1.n.i = 0	var 20	G.p.i
eqn 21	R1.p.i + G.p.i + AC.n.i = 0		
eqn 22	AC.n.v = G.p.v		

Table 5.1: Flattened equations for the model in Figure 5.1.

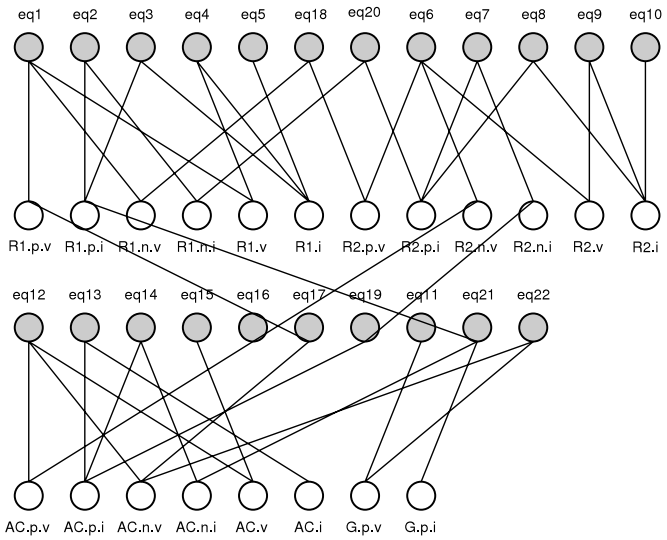

Figure 5.3: Over-constrained bipartite graph

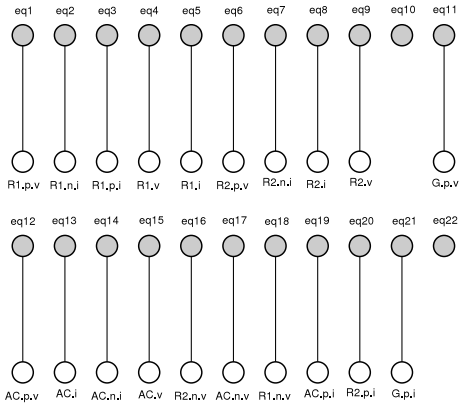

Figure 5.4: A maximum cardinality matching of an over-constrained graph

tions is greater then the number of variables. By choosing a maximum cardinality matching (depicted in Figure 5.4) and performing the D&M canonical decomposition the two over-constrained subgraphs pictured in Figure 5.6 and the well-constrained part in Figure 5.5 are isolated.

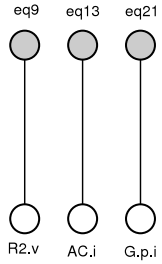

Figure 5.5: Well-constrained subgraph

Performing a variable reachability analysis gives us the equations that can be considered for removal. Component one leaves us with only two alternatives: equation 5 or equation 10 since removal of any other equation would disconnect the graph. Component two on the other hand leaves us no less then 14 possible equations (equation 1, 2, 3, 4, 5, 6, 8, 9, 11, 15, 16, 17, 18 and 20) that could be removed while maintaining connectivity. Fortunately we can now further reduce the set of equations to be considered for removal by using our language specific assumptions. Equations 16, 17, 18 and 20 are generated from a `connect` statement and can thus be eliminated from the set of equations that are possible to remove. We can further reduce the set by removing those equations that have a higher number of associated equations. This operation will remove equation 1, 2, 3, 4, 6, 8, and 9 since they have two or more other equations associated with them, contrary to equation 5, 11 and 15 who only have one.

We are now ready to present the user with elaborated information on what went wrong and what should be done about it. Two equations has to be removed. One equation cannot be removed twice and since equation 5 is present in both the first and the second set of equations we can with the help of the graph representing possible removals in Figure 5.7, where every path of length 1 represents a possible set for removal, inform the user that

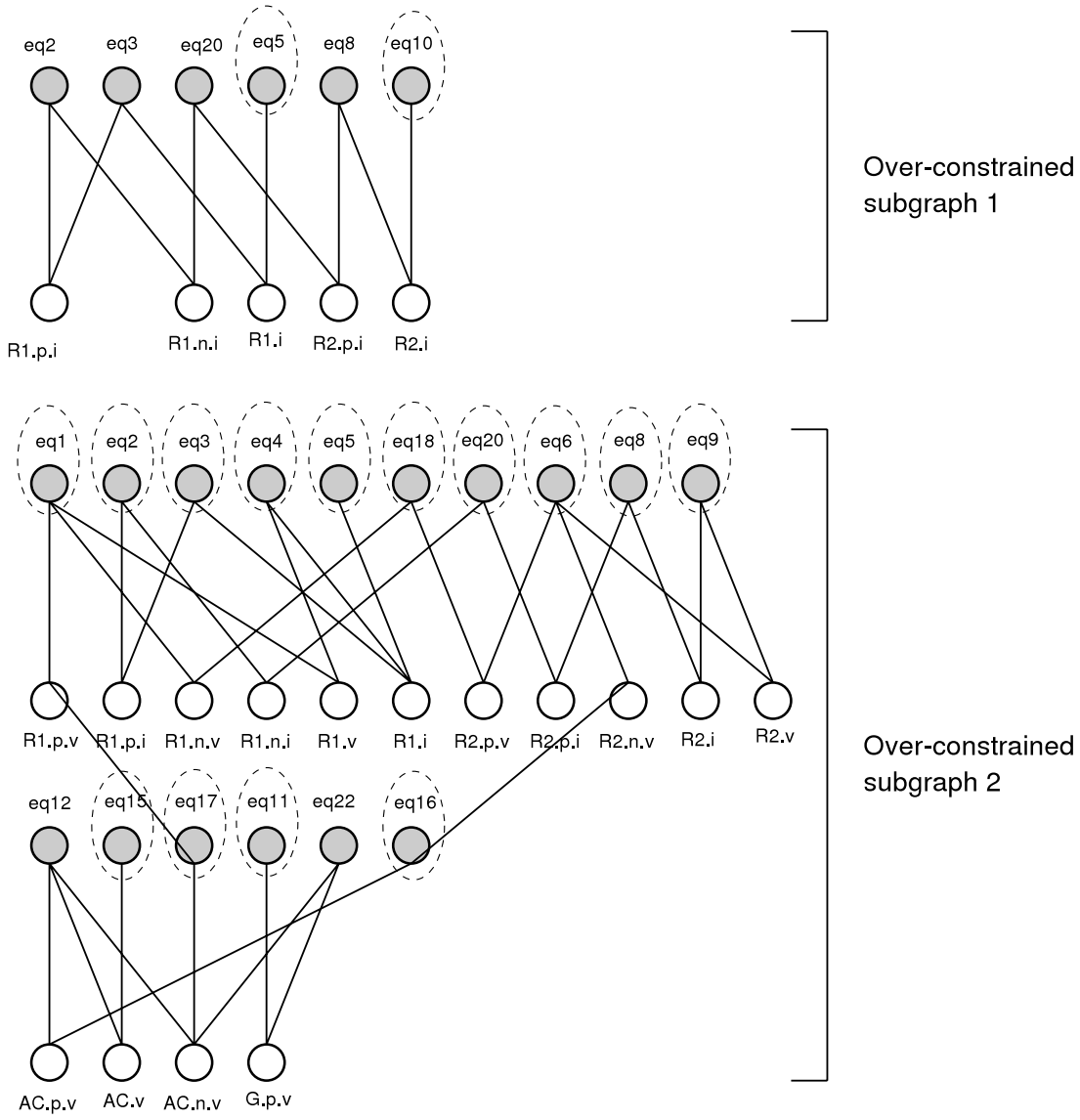

Figure 5.6: Over-constrained subgraphs

his options for debugging the model are:

1. Removing equation 5 (R1.i=2) and
   equation 11 (G1.p.v=0)

2. Removing equation 5 (R1.i=2) and
   equation 15 AC.v = AC.VA * sin[2*time*AC.f*AC.Pi]

3. Removing equation 10 (R2.i=2) and
   equation 5 (R1.i=2)

4. Removing equation 10 (R2.i=2) and
   equation 11 (G1.p.v=0)

5. Removing equation 10 (R2.i=2) and
   equation 15 AC.v = AC.VA * sin[2*time*AC.f*AC.Pi]

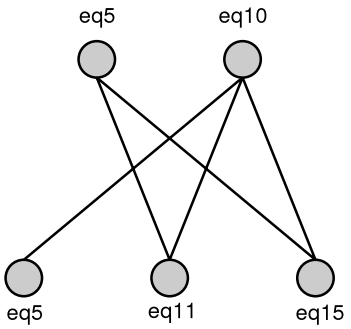

Figure 5.7: Equationsets to be considered for removal

It is possible to give even more aid to the user by backtracking to the origin of the equation. Doing so we would find that equation 5 and 10 both originate from the same equation in the resistor model. This would be a strong indication that this equation is the one causing the over-constrainment problems. This has however not been implemented in this thesis.

5.2 Debugging Under-Constrained systems

Just as in the case with over-constrained components, we will receive the under-constrained components from the D&M decomposition. Debugging under-constrained system is however much more difficult due to the multitude of possible solutions that has to be considered. Two distinct error fixing solutions exists:

- *Removal of free variable nodes*
- *Addition of new equation nodes*

In the case of removal of variables, the reasoning is similar to the over-constrained case. Removals must not disconnect the graph, cause further over-or under-constrained graphs or remove unflexible variables, such as variables generated by a `connect` statement.

In the other case when we seek to insert extra equations instead, we need to find out what variables they should, and should not contain. This is done by performing a variable reachability analysis, in conjunction with user interaction as the user in the end must point out in which part of the model the debugger should insert the extra equations.

Variable reachability analysis, described in [Bunus, 2002] starts by creating an inheritance-instantiation graph for the objects in the model. This graph should also contain from which object the different variables originate. This graph can then help us to see what effects introduction of an extra equation in the different objects will give us in the end and which variables could be included in such an equation.

For an example of under-constrained debugging we refer to [Bunus, 2002].

5.3 Debugging Simultaneous Over and Under-Constrained systems

When we have both under-constrained and over-constrained components in a model, a special situation arises. The most convenient way to solve this is obviously to connect the free variable in the under-constrained part to the

free equation in the over-constrained part. This could solve both constrainment problems simultaneously. If this is possible or not can be determined by the same sort of variable reachability analysis that is performed in the case of under-constrained systems.

Chapter 6

Implementation

This chapter describes details of the implementation of the optimizer and debugger described in this thesis.

6.1 Reused Libraries

The combined optimizer and debugger are designed to be included in ModSimPack which in turn is included in the Open Source Modelica Compiler. Since the compiler is intended to be open source, probably LGPL [Free Software Foundation, 1999], commercial libraries are generally not usable. The primary need for an external library has concerned graph representation and the algorithms related to it. The final choice was the boost graph library described in [Siek et al., 2001] and [Siek et al., 2000], which provides both a customizable graph representation and some of the algorithms needed in the implementation, for example the algorithm that computes the algorithm for strongly connected components.

Furthermore there was a need for symbolic manipulation of equations and variables. For this particular problem, the Symbolic C++ library described in [Kiat et al., 1998] was chosen.

A third library is also used for computing the BLT form of the incidence matrix. This library is written in Fortran and is due to [Duff and Reid, 1978b]

and [Duff and Reid, 1978a].

6.2 Architecture and language

The implementation is written in ANSI C++ although Fortran is also used to some extent (see Section 6.1).

The implementation has been tested on Solaris, Gnu/Linux with gcc and Windows XP Professional using Microsoft Visual .NET compiler. Most platforms with a C++ compiler should be able to compile and use the optimizer. Installation of the included libraries is however necessary, but this should present only minor problems in terms of compatibility as they are all distributed with source code.

6.3 Limitations in this implementation

The most notable limitation in the current implementation is the lack of user interaction in the debugger. This limits the possibility to let the user participate in the debugging process. Particularly the debugging of under-constrained systems suffer from this lack. Part of the problem is that no query language has yet been specified to facilitate interaction between the front-end of the Open Source Modelica Compiler and ModSimPack.

The optimizer is fairly complete, except for the index reduction parts which are not entirely ready for integration into ModSimPack yet. The computation of the structural index is however, and only the actual rewriting of the indicated equations to index one remains.

Chapter 7

Conclusions and Future Work

7.1 The Debugger

The debugger has demonstrated that it is useful and it does indeed present understandable error fixing alternatives to the user. Error correction is not yet possible in the debugger without human interaction but error detection and identification is very accurate. It is very rare that the debugger is not able to provide any help whatsoever.

7.2 The Optimizer

Tests have shown that the decomposition into BLT form does give a significant performance. In Figure 7.1 is a comparison between solving of optimized and unoptimized typical problems.

As can be seen, there is always at least a small gain in performance. The magnitude of the gain is, however, dependent of the type of problem.

The index reduction part does not contribute significantly to the compilation speed but it does make it possible for us to use a general solver

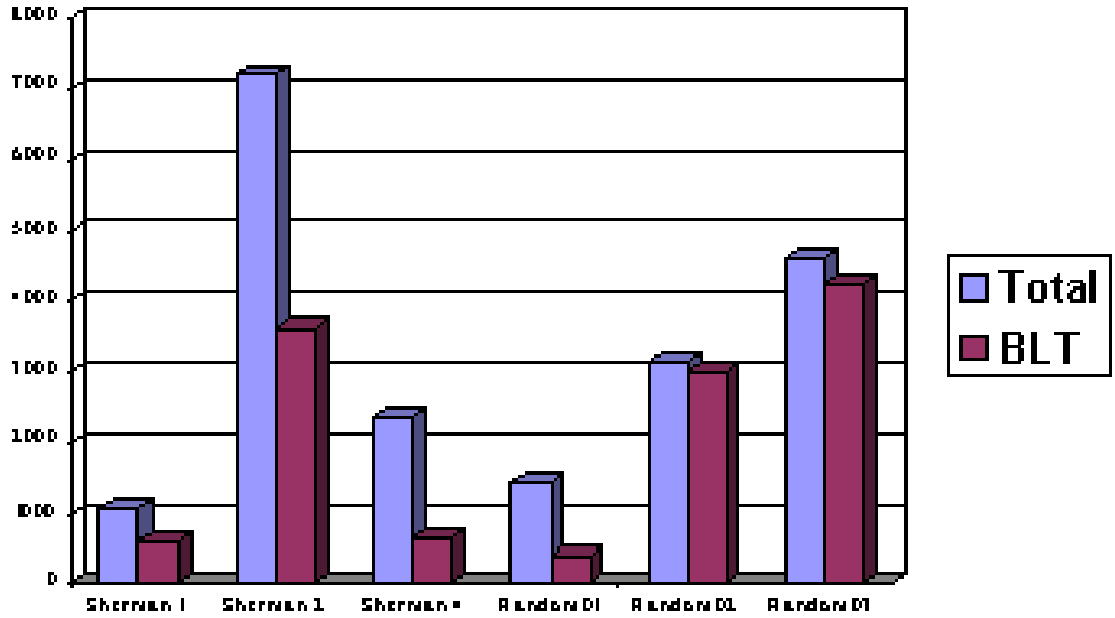

Figure 7.1: Performance when solving optimized problem compared to solving same problem without optimization

that can take care of any problem we can produce, since its index will be at most one. We do however have to make a small sacrifice in precision to obtain this goal. This is a tradeoff one has to make to be able to solve every consistent problem with reasonable speed.

7.3 Future Work

Source code interaction would be of great help since that would provide the possibility to fix a wider range of problems and even to greater extent eliminate the need to recompile the whole model if an error is detected. This should probably be the first priority for improving the debugger.

Some parts of primarily the optimizer could be rewritten to for example fortran in order to make them run faster. Already, the Dulmage and Mendelsohn Decomposition is done with the help of an external fortran library. The maximum weighted matching algorithm, for example could also be rewritten in fortran for greater compilation speed, should the need

arise.

In order to further increase the compilation speed, a better algorithm for maximum weighted matching than the one implemented in this thesis exists, proposed by [Uno, 2001]. Even other algorithms may be improved if we take advantage of various properties of our specific problems. For example, a depth first search can be aborted once an alternating path has been found when computing the maximum matching. An extensive search does not have to be made.

Parallelization of the solving of equations is an interesting aspect which deserves to be explored. A parallel processing capable solver could be a first step. It is also possible that a certain performance increase could be obtained by symbolically solving blocks from the BLT form in parallel and then joining them together in the end. A third approach is obviously to generate C-code that can be executed in parallel. This approach is taken in [Aronsson, 2002].

In order to make the compiler more flexible, different code generation modules could be of great advantage. Instead of generating c-code on could generate for instance matlab, C++ or fortran code. This could give the possibility to take advantage of for example fortrans superior speed or the multitude of interactive analysis tools from matlab.

Dynamic debugging is also an interesting way to extend the optimizer-debugger framework. Numerical singularities that will cause problems in the solver cannot be detected by static debugging. Dynamic debugging, especially with source code interaction could however, and would be of great help in the debugging process.

Chapter 8

Related Work

Structural analysis is nowadays widely used both for optimization and debugging purposes. It has proven quite helpful both in finding errors early in the debugging process and improving performance when solving the system. [Murota, 2000] is one such example which uses a mathematical approach to structural analysis and uses it to formulate solvability criteria for systems. Another example is the Abacuss II environment [John E. Tolsma and Barton, 2002] which is able to via Dulmage and Mendelsohn decomposition determine and visualize over- and under- constrained components and thereby locate the section of equations where the detected error resides.

Constraint satisfaction systems such as ours are also used by others. The implementation most similar to ours is by [Ait-Aouida et al., 1993] who in their system for geometric modeling by constrains uses a similar decomposition technique resulting in well-, over- and under-constrained components. Furthermode, in [C. Bliek and Trombettoni, 1998] a method is proposed to solve well-constrained problems by combining constraint solvers with backtracking techniques.

There are also examples of other solutions to the problems with componens with a high differential index. [Ramos et al., 1998], for example presents a method to analyze the system and to identify common problems such as high index DAEs. Some numerical problems can then be avoided

by exchanging these identified problems with equivalent models which are easier to solve.

Bibliography

[Abadi and Cardelli, 1996] Abadi, M. and Cardelli, L. (1996). *A Theory of Objects*. Springer-Verlag.

[Ait-Aouida et al., 1993] Ait-Aouida, S., Jegou, R., and D.Michelucci (1993). Reduction of constraint systems. *Compugraphic*, pages pp 83–92.

[Anderson et al., 1990] Anderson, E., Sorensen, D., Bai, Z., Dongarra, J., A.Greenbaum, McKenney, A., Croz, J. D., S.hammerling, J.Demmel, and Bischof, C. (1990). Lapack: A portable linear algebra library for high-performance computers. In *In Proceedings of Conference on Supercomputing (New York, United States)*.

[Aronsson, 2002] Aronsson, P. (2002). Automatic parallelization of simulation code from equation based simulation languages. Licenciate thesis, University of Linköping, Sweden.

[Bunus, 2002] Bunus, P. (2002). Debugging and structural analysis of declarative equation-based languages. Licenciate thesis, University of Linköping, Sweden.

[C. Bliek and Trombettoni, 1998] C. Bliek, B. N. and Trombettoni, G. (1998). Using graph decomposition for sloving continous csps. In *Principles and Practice of Constraint Programming*, pages pp 102–116. CP'98, Pisa, Italy.

[Duff and Reid, 1978a] Duff, I. S. and Reid, J. K. (1978a). Algorithm 529: permutations to block triangular form. *ACM Transactions on Mathematical Software (TOMS)*, 4:2:pp 189–192.

[Duff and Reid, 1978b] Duff, I. S. and Reid, J. K. (1978b). An implementation of tarjan's algorithm for the block triangularization of a matrix. *ACM Transactions on Mathematical Software (TOMS)*, 4:2:pp 189–192.

[Dulmage and Mendelsohn, 1963] Dulmage, A. and Mendelsohn, N. (1963). Coverings of bipartite graphs. *Canadian J Math*, 10:pp. 517–534.

[Fegery and Barton, 1998] Fegery, W. and Barton, P. (1998). Dynamic optimization with state variable path constraints. *Computers in Chemical Engineering*, 22:pp. 1241–1256.

[Free Software Foundation, 1999] Free Software Foundation (1999). Gnu lesser general public license. Available at http://www.gnu.org/copyleft/lesser.html; accessed 26th March 2003.

[Fritzon et al., 1998] Fritzon, P., Aronsson, P., Bunus, P., Engelson, V., Johansson, H., Karström, A., and Saldamli, L. (1998). The open source modelica project. *In Proceedings of the 2nd International Modelica Conference (18-19 March, Munich Germany).*

[Fukuda and Matsui, 1994] Fukuda, K. and Matsui, T. (1994). Finding all the perfect matchings in bipartite graphs. *Appl. Math. Lett.*, 7(1):pp. 15–18.

[Health, 2002] Health, M. T. (2002). *Scientific Computing. An Introductory Survey - Second Edition.* MH Higher Education.

[Himsolt, 1996a] Himsolt, M. (1996a). Gml: A portable graph file format. Technical report, Universität Passau 1997.

[Himsolt, 1996b] Himsolt, M. (1996b). Gml: Graph modeling language. Available at http://www.infosun.fmi.uni-passau.de/Graphlet/GML/; accessed 26th March 2003.

[Hindmarsh, 1983] Hindmarsh, A. C. (1983). Odepack, a systematized collection of ode solvers. *Scientific Computing, North-Holland Amsterdam*, pages pp. 55–64.

[Hopcroft and Karp, 1973] Hopcroft, J. and Karp, R. (1973). An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal of Computing*, 2(4):pp. 225–231.

[John E. Tolsma and Barton, 2002] John E. Tolsma, J. C. and Barton, P. I. (2002). Abacuss ii, advanced modeling environment and embedded simulator. Available at http://yoric.mit.edu/abacuss2/abacuss2.html; accessed 4th April 2003.

[Kågedal and Fritzon, 1998] Kågedal, D. and Fritzon, P. (1998). Generating a modelica compiler from natural semantics specifications. In *In Proceedings of the 1998 Summer Computer Simulation Conference (SCSC1998), Reno Nevada, USA*.

[Kiat et al., 1998] Kiat, S. T., Willi-Hans, S., and Yorick, H. (1998). *Symbolic C++: An Introduction to Computer Algebra Using Object-Oriented programming*. SV.

[Murota, 2000] Murota, K. (2000). *Matrices and Matroids for System Analysis*. SV.

[Pantelides, 1988] Pantelides, C. (1988). The consistent initialization of differential algebraic systems. *SIAM Journal on Scientific and Statistical Computing*, 9:pp. 213–231.

[Pettersson, 1995] Pettersson, M. (1995). *Compiling Natural Semantics*. University of Linköping.

[Petzold, 1983] Petzold, L. R. (1983). A description of dassl: A differantial/algebraic system solver. *Scientific Computing, North-Holland Amsterdam*, pages pp. 65–68.

[Ramos et al., 1998] Ramos, J. J., Àngel, P. M., and Ignasi, S. (1998). The use of physical knowledge to guide formula manipulation in system modeling. *Simulation Practice and Theory*, 5:pp 243–254.

[Siek et al., 2000] Siek, J., Lee, L.-Q., and Lumsdaine, A. (2000). The boost graph library. Available at http://www.boost.org/libs/graph/doc/; accessed 26th March 2003.

[Siek et al., 2001] Siek, J., Lee, L.-Q., and Lumsdaine, A. (2001). *The Boost Graph Library: Users Guide and Reference Manual*. AWPearson Education Inc.

[Tarjan, 1972] Tarjan, R. (1972). Depth-first search and linear graph algorithms. *SIAM J. Computing*, 1.

[Uno, 2001] Uno, T. (2001). A fast algorithm for enumerating bipartite perfect matchings. In *In proceedings of twelfth annual international symposium on algorithms and computation (ISAAC2001), Christchurch, New Zealand*, pages pp. 367–379.

Avdelning, Institution
Division, Department

IDA,
Dept. of Computer and Information Science
581 83 Linköping

LINKÖPINGS UNIVERSITET

Datum
Date

30th May 2003

Språk
Language
☐ Svenska/Swedish
☒ Engelska/English
☐ _____

Rapporttyp
Report category
☐ Licentiatavhandling
☒ Examensarbete
☐ C-uppsats
☐ D-uppsats
☐ Övrig rapport
☐ _____

ISBN
—

ISRN
LITH-IDA-EX–03/040–SE

Serietitel och serienummer **ISSN**
Title of series, numbering —

URL för elektronisk version

Titel Debuggningsstöd för en optimerad Modelicakompilator

Title Debugging Support for an Optimized Modelica Compiler

Författare Kaj Nyström
Author

Sammanfattning
Abstract

The need for modeling and simulation in engineering is continuously rising as systems become increasingly complex and physical prototyping becomes too expensive. The thesis aims at optimizing the performance of simulation in terms of lowering the time spent solving the equations constituting the model. This is done by decomposing the equation system into sequentially solvable components, making it possible for the solver to work on many smaller components instead of one large component. The system is also transformed in order to be solvable with a less complex, and faster solver through rewriting of differential equations of higher degree to differential equations of index one.

The second purpose of the thesis is to facilitate debugging of faulty models through an integrated debugger. The debugger detects inconsistent models and presents to the user elaborated alternatives to correcting the model. This is done at the original source code level and not in some intermediate language which tend to be difficult to understand for the user.

Results from testing of the implementation shows improvements in performance for several different problems and also enables solving of problems that would normally require a special high index solver. It has also demonstrated its ability to facilitate debugging, primarily as a fault detector but also as a solution provider in a user friendly maner.

The implementation is included into ModSimPack which constitutes the symbolic and numerical engine of the Open Source Modelica Compiler, a compiler for the Modelica Language.

Nyckelord
Keywords Modelica, Optimizer, Debugger, Compiler

Copyright

Svenska

English