MASTER'S THESIS 2022

Improving Tearing in a Modelica Compiler

Oskar Kari

ISSN 1650-2884
LU-CS-EX: 2022-16

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2022-16

Improving Tearing in a Modelica Compiler

Förbättra tearing i en Modelica kompilator

Oskar Kari

Improving Tearing in a Modelica Compiler

Oskar Kari
oskarkari93@gmail.com

Mars, 2022

Master's thesis work carried out at Modelon AB.

Supervisors: Niklas Fors, niklas.fors@cs.lth.se
Filip Stenström, filip.stenstrom@modelon.com
Markus Olsson, markus.olsson@modelon.com

Examiner: Görel Hedin, gorel.hedin@cs.lth.se

Abstract

Modelica is a simulation language for systems containing electrical, mechanical, thermal and other components. The Modelica code is compiled to an equation system containing both algebraic and differential equations which is then simulated using numerical solvers. Tearing is a dimensionality reduction technique applied on this equation system and is important both for optimization of the runtime code and for accuracy of the simulation. Developing a Tearing algorithm is hard. Several things must be considered such as numerical properties of the equation system and how to maximally reduce the dimension, the latter problem is NP-complete. In this thesis I will develop two tools to mitigate bad decisions taken by the algorithm. The first tool allows the user to manually do parts of the Tearing while the second tool will provide input from the user to the algorithm.

Keywords: Tearing, Method of Tearing, Diacoptics, Optimization, Compiler, Modelica

Acknowledgements

I would like to thank Filip Stenström, Markus Olsson and Agnes Ramle at Modelon for all the help with this thesis and thanks for having me at Modelon.

I would also like to thank Niklas Fors, my supervisor at LTH, for all the advice with the project and for the help with the report.

Contents

1 Introduction **7**

2 Background **9**
- 2.1 Modelica 9
- 2.2 The Compiler Pipeline 10
  - 2.2.1 Alias Elimination 11
  - 2.2.2 BLT 11

3 Tearing **15**
- 3.1 The need for Tearing 15
- 3.2 A Simple Tearing Example 16
- 3.3 Node Tearing 17
- 3.4 Automatic Tearing 20

4 Contributions to Tearing **23**
- 4.1 Start Values 23
- 4.2 HGT for Dynamic Systems 24
- 4.3 Unpaired Tearing 25

5 Implementation **27**
- 5.1 Hand Guided Tearing for Dynamic System 27
- 5.2 Unpaired Tearing 28
- 5.3 Alias Elimination 31

6 Evaluation **33**
- 6.1 HGT for Dynamic Systems 33
- 6.2 Unpaired Tearing 34
- 6.3 Linear Blocks 35
- 6.4 Compile time and Runtime 35

7 Discussion **37**
7.1 Uneven Unpaired Tearing 37
7.2 The two Models from Modelon 38
7.3 Linear Blocks 38
7.4 Selection of MSL Models 39
7.5 Tearing in OpenModelica 39

8 Conclusion **41**

References **43**

Chapter 1
Introduction

Modelica is a modeling language in which it is possible to build and simulate complex models for systems containing electrical, mechanical, thermal and other subcomponents [11]. Examples of such systems can be airplanes, automotive powertrains, air conditioners etc. There are two different kinds of models; steady state and dynamic systems. Steady state systems are described using a system of algebraic equations (AE system). Dynamic systems consists of a combination of algebraic equations and differential equations. These combined systems are called Differential Algebraic Equations (DAE). A Modelica compiler must translate the Modelica code to either an AE system or a DAE system. After this several optimizations on the system are performed because models of physical systems often put a lot of demand on computer power during simulation. Furthermore the compiler must make sure that the system can be solved and simulated during runtime and there are several transformations on the system that can be done for this purpose.

The company Modelon has developed a compiler for Modelica that will be modified in this thesis. In this report it will be called the OCT compiler. In this compiler there are several optimization steps being conducted. The last of these optimization steps and the focus of this thesis is an optimization called BLT. In this step the equation system is partitioned into smaller blocks that can be solved semi-independently. On each of these blocks an optimization called Tearing is then done. In this thesis I will do contributions to this Tearing optimization.

Tearing is done mainly for two reasons; to make the simulation faster during runtime and to increase the likelihood that it is possible to solve the equation system with the generated runtime code. What Tearing does is to make the problem that needs to be solved smaller. First the equation system is partitioned into smaller blocks, then Tearing is applied to each of these smaller parts to break them into even smaller blocks.

The first step in the Tearing algorithm is to pick variables and equations in the equation system. We call the variables that are picked for iteration variables and the equations that are picked for residual equations. The iteration variables and the residual equations must form pairs, so an equal number of residual equations and iteration variables must be picked and

each iteration variable must be paired with a residual equation. The pairing affects numerical properties of the tearing problem.

When picking iteration variables and residual equations there are several things that must be taken into consideration. One important goal is to make the resulting blocks after tearing as small as possible. Unfortunately picking the optimal residual equations and iteration variables that fulfills creates the smallest blocks is a NP-complete problem [9]. In addition to this other properties must be taken into consideration like numerical properties of the system. Sometimes domain knowledge of the system can also give insight in how to select iteration variables and residual equations. An other factor that is relevant when picking iteration variables is if the variables have start values and how good these start values are.

In the OCT compiler there is an algorithm called automatic tearing that will attempt to pick the best iteration variables and residual equations. However as this is a very hard problem to solve this algorithm will occasionally pick very bad variables or equations. Therefore it is important to have an option for the modeller to pick iteration variables and residual equations manually. This will be called Hand Guided Tearing (HGT) in this thesis. Currently the OCT compiler only has a rudimentary support for HGT where only steady state is supported. In this HGT algorithm the user must specify both the residual equations and the iteration variables. Since the equations and variables must form pairs the user must specify an equal number of equations and variables.

In this thesis the support for HGT is expanded. First the current HGT will be extended to also support dynamic systems. Then a new kind of HGT is implemented that is a mix between the current automatic tearing algorithm and HGT. In this new HGT algorithm the user only needs to specify variables or equations. It is also possible to specify an unequal number of equations and variables. The algorithm will then match the surplus equations or variables with the best possible match.

The evaluation of the two different HGT algorithms developed in this thesis shows that both works as intended and provides no statistical significant increase to compile time. Futhermore I will also use these algorithms to solve two problems that could not previously be solved.

In Chapter 2 I will give a background to concepts in the OCT compiler that are necessary to understand this thesis excluding tearing. The background to tearing will be given in Chapter 3. In Chapter 4 I will give some additional motivation for tearing and also describe my own contributions in detail. In Chapter 5 I will describe the implementation of my own contributions and in Chapter 6 the implementation will be evaluated in different ways. The discussion takes place in Chapter 7.

Chapter 2
Background

2.1 Modelica

Modelica is a declarative, object-oriented simulation language [11]. Models in Modelica can be described by a mathematical system of algebraic, differential or discrete equations. A model can describe different physical components like electrical components, mechanical components, thermal components etc. By using several models it is possible to describe large and complex physical systems that have many different kinds of subcomponents. One example of a simple system programmed in Modelica can be seen in Listing 2.1. This model is called *BouncingBalls* and describes two balls that bounces. The model has two subcomponents *b1* and *b2* that are both instances of the model *Ball* which describes a ball.

The two balls have different coefficient of restitution witch results in different bounce heights even if the drop hight is the same. The result when simulating this model can be seen in Figure 2.1.

Listing 2.1: Modelica code describing two bouncing balls

```
model BouncingBalls
    Ball b1(e = 0.9);
    Ball b2(e = 0.8);
end BouncingBalls;

model Ball
    parameter Real e "Coefficient of restitution";
    parameter Real h0 = 1.0 "Initial height";
    Real h "Height";
    Real v(start = 0, fixed = true) "Velocity";
initial equation
    h = h0;
equation
```

```
    v = der(h);
    der(v) = -9.81;
    when (h < 0) then
        reinit(v, -e * pre(v));
    end when;
end Ball;
```

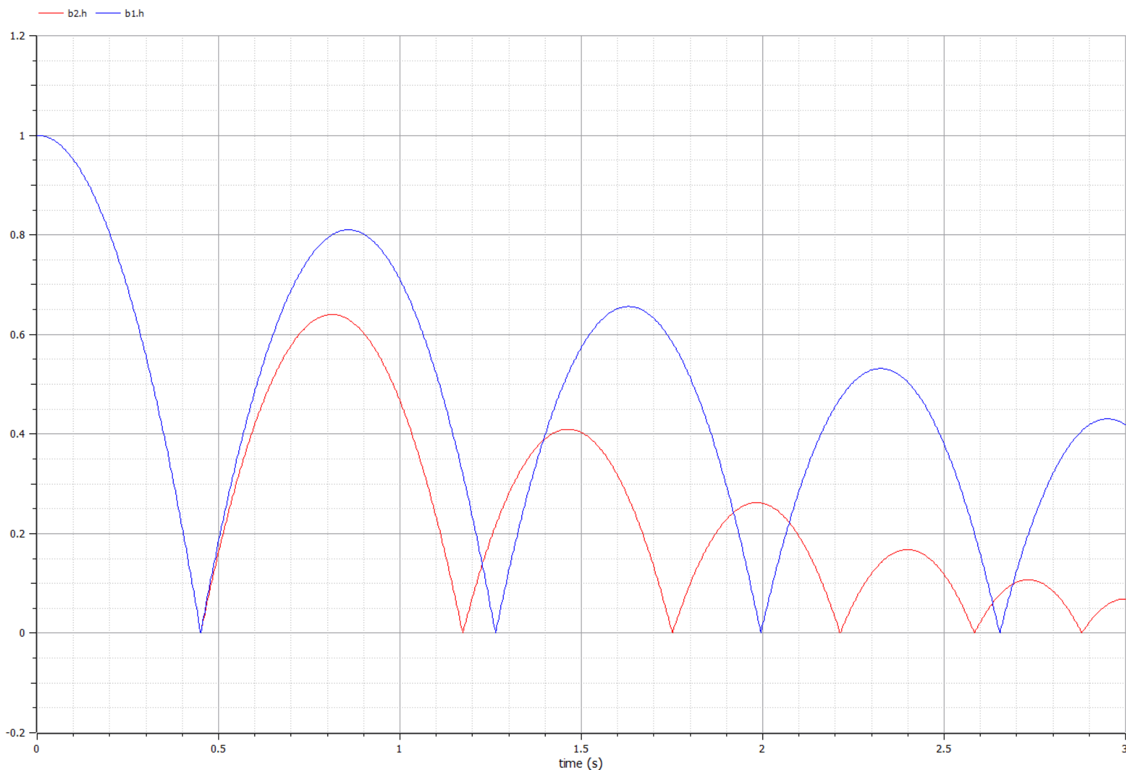


Figure 2.1: The simulation results when simulating the model of the two balls described in Listing 2.1. The height of the balls in meters is plotted as a function of time.

The fact that Modelica is declarative means that the execution order does not matter. For example in Listing 2.1 we have several different equations in the Ball model and also a discrete when statement. The order of these equations and the when statement does not matter.

The compiler that I use in this thesis is the OCT compiler developed by Modelon but there are other compilers as well. For example there is a compiler called OpenModelica that is open source unlike the OCT compiler.

2.2 The Compiler Pipeline

The compiler pipeline is shown in Figure 2.2. First the compiler parses the Modelica code and after this step we have an Abstract Syntax tree (AST) [12]. An AST is a tree like datastructure that describes source code. Later in the compile pipeline there is a step called *flattening*. In this step a new AST called Flat-AST is built and the old AST is thrown away [12]. The Flat-AST has thrown away all object oriented structure and only contains one single model. This single model is one big equation system with variables and equations.

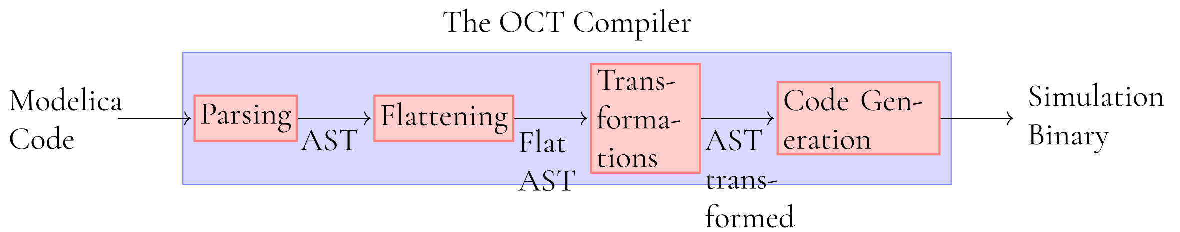


Figure 2.2: The compiler pipeline.

After the flattening step there are several transformations that are done on the Flat-AST for optimization purposes. Two of those will be covered below, first Alias Elimination that is covered in Section 2.2.1 and secondly BLT which is covered in Section 2.2.2.

After these transformations the code is generated and we get a simulation binary.

2.2.1 Alias Elimination

Alias elimination is a simple transformation that checks for expressions of the form $x = y$ or $x = -y$. In this cases alias elimination simply replaces all y with x. This can be done because Modelica is declarative so the order of the equations does not matter. This transformation in it self is very simple but it is used as a step in other, more complex transformations such as variability propagation. These other transformations can create aliases that for optimization purposes should be removed.

2.2.2 BLT

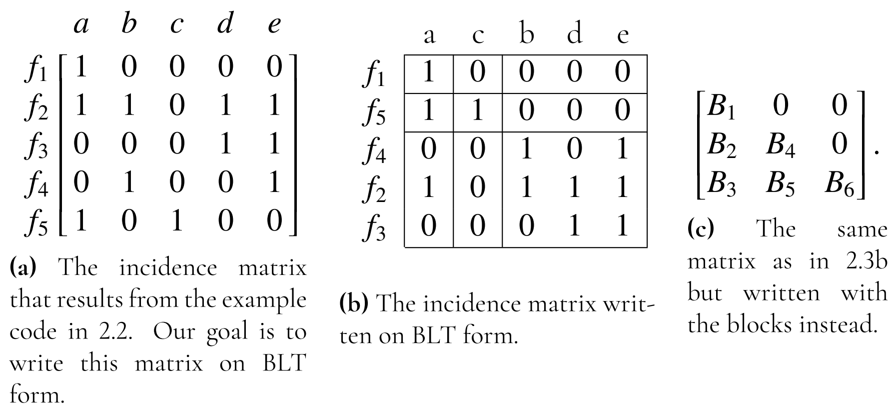
BLT is the last transformation that is done [12] before code generation. It is the last step before we have the *AST transformed* in Figure 2.2. This step is done to partition the model into smaller parts that can be solved semi-independently during runtime. Before this step we have a model that describes one big equation system. BLT splits this equation system into smaller equation systems that must be solved in a specific order. What determines the size of these smaller equation systems is usually circular dependencies between variables. In the OCT compiler this step also contains Tearing which is the topic of this thesis.

BLT is short for Block Lower Triangular. A lower triangular matrix is a matrix elements where all elements above the diagonal are 0. Non-zero elements are only allowed on the diagonal and below.

A block matrix is a matrix that has been partitioned into blocks using vertical and horisontal lines. A BLT is a matrix where every block above the diagonal only contains 0 elements. An example of a BLT matrix with the blocks marked is shown in Figure 2.3b. This BLT matrix can also be written as shown in Figure 2.3c, here we have for example

$$B_6 = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix}.$$

Before the BLT transformation step we have one model. Because of flattening there cannot be more than one single model. This model can be described using a matrix. For example



(a) The incidence matrix that results from the example code in 2.2. Our goal is to write this matrix on BLT form.

(b) The incidence matrix written on BLT form.

(c) The same matrix as in 2.3b but written with the blocks instead.

Figure 2.3: An incidence matrix that describes the model in Listing 2.2. This incidence matrix is then transformed to an incidence matrix written on BLT form. This BLT form is also shown using blocks.

the code in Listing 2.2 is represented by the matrix in Figure 2.3a. This matrix is an incidence matrix that marks the relationship between variables and equations. The goal of the BLT step is to transform this matrix into BLT form.

Listing 2.2: Modelica example

```
model SimpleEx
  Real a, b, c, d, e;
equation
  sqrt(a) = 65 "Equation f1";
  d = a/(b*e) "Equation f2";
  e = d^3 "Equation f3";
  b = sqrt(e) "Equation f4";
  0 = a^2 + c "Equation f5";
end SimpleEx;
```

To find the BLT form of a matrix we must first find the blocks. Once the blocks are known it is very simple to rewrite the matrix to BLT form.

To find blocks that allows the matrix to be rewritten on BLT form the first step is to make a Bipartite graph from the code. A Bipartite graph is a graph that can be divided into two sets, A and B, so that every edge connects one element in set A with one element in set B. So we cannot have any edges that connects elements in the same set.

The code in Listing 2.2 results in the Bipartite graph in Figure 2.4a. One of the sets contains the equations and the other contains the variables. There is an edge connecting the equations with every variable that exsists in that equation.

Once we have this Bipartite graph then the goal is to get a perfect maximum matching in this graph. A matching is a subset of the set of edges so that no two edges contains the same node. An example of a matching can be seen in Figure 2.4b. As can be seen no edge in the matching subset contains the same equation or variable as an other edge in this subset.

A maximum matching is a matching that contains the maximum number of edges possible. So all other matching that are possible contains either fewer or the same number of edges. Figure 2.4c shows an example of a maximum matching. A maximum matching is called per-

fect if every equation has an edge to a variable and every variable has an edge to an equation. As can be seen the maximum matching in 2.4c is perfect. There are usually several different perfect maximum matchings.

If it is not possible to create a perfect maximum matching the problem is unsolvable and an error is returned. If we have a perfect maximum matching then every equation is matched with a variable.

Once we have the perfect maximum matching it is possible to find the blocks from this. This is done using an algorithm called tarjan's strongly connected components algorithm [1]. This algorithm takes as input the perfect maximum matching and also the bipartite graph shown in 2.4a. The output is the diagonal blocks and also the order in which the blocks are to be solved.

Tarjan's algorithm gives the following diagonal blocks

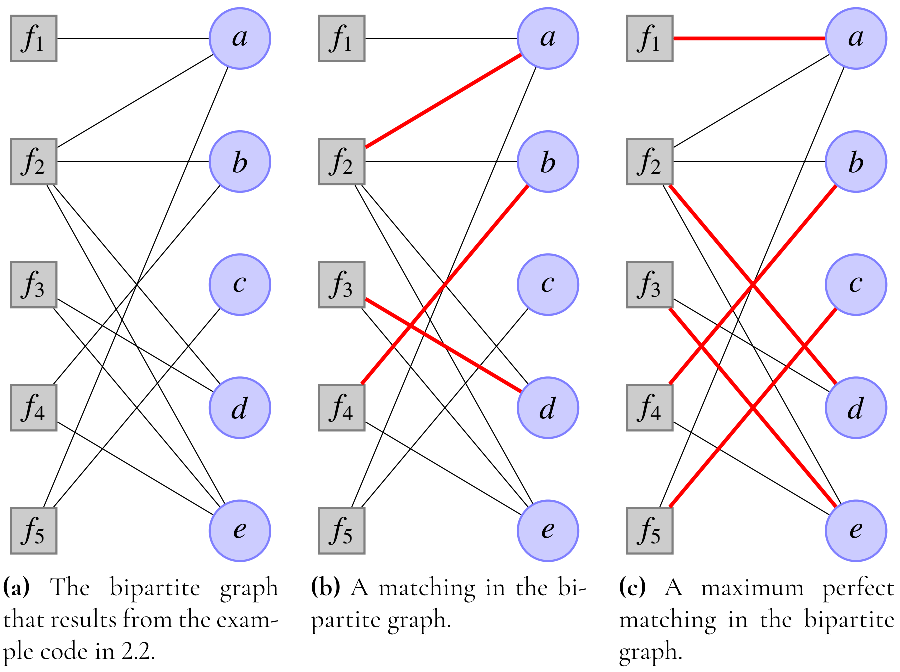
$$B_1 = \begin{bmatrix} 1 \end{bmatrix},$$

$$B_4 = \begin{bmatrix} 1 \end{bmatrix},$$

$$B_6 = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix}.$$

This gives the BLT form shown in Figure 2.3b. The reason why B_6 is of size $3x3$ is because there is a circular dependency between the variables b, d and e. All blocks of size $1x1$ contains no circular dependency, but the blocks of bigger size contains a circular dependency which makes these blocks bigger.

From the BLT form in Figure 2.3b it is now possible to solve the equation system by solving the diagonal blocks one at a time. First solve B_1, then B_4 and finally B_6. The individual blocks can be solved using an optimization algorithm such as Newton optimization. However in almost all cases these blocks are going through a transformation called tearing before the runtime code is generated and the optimization takes place. Tearing is a method to reduce the dimension of the optimization problem.



(a) The bipartite graph that results from the example code in 2.2.

(b) A matching in the bipartite graph.

(c) A maximum perfect matching in the bipartite graph.

Figure 2.4: A bipartite graph with different matchnings.

Chapter 3

Tearing

First invented by Gabriel Kron [8] is a method he called Diacoptics. In the scientific literature this method goes under two different names. Sometimes it is called Diacoptics and sometimes Tearing. These words essentially means the same thing. Coptics is latin and means *to tear* and Dia is a pre-fix that augments the word that follows. So diacoptics means *to tear strongly* in latin. In this thesis I will use the word Tearing because this seems to be the most common name in computer science literature.

Gabriel Kron was an electrical engineer who used Tearing to optimize power lines. Since then Tearing has found uses in a wide variety of fields including chemistry [9], physics [9], computer science [4] and a wide variety of fields in electrical engineering [7].

Tearing in the OCT compiler is applied on diagonal blocks that are bigger that 1x1. So in the example code from Listing 2.2 tearing is only applied on block B_6.

3.1 The need for Tearing

In this section I will use Newton optimization. In Newton optimization we need to calculate a Hessian matrix. This matrix contains all the second derivatives. For an introduction to newton optimization see [13].

Tearing reduces the dimension of the optimization problem. Without tearing the Hessian in Newton optimization gets the same dimension as the block that we are trying to solve. There are several downsides of this. First of the elements in this matrix takes a lot of computer power to calculate as the elements must be numerically approximated. For real problems there can be hundreds of variables in each block and since the size of the Hessian is increasing with $O(n^2)$ even small reductions of the number of variables n can cause big reductions in the number of elements in the matrix.

Secondly the optimizer generally has a harder time to find good values for the variables when number of iteration variables grows. This is especially true if the iteration variables also have very different magnitude. For many real models used at Modelon the magnitude

between variabels can be very big. This is especially true in thermodynamics where differences as large as 10^{15} are common.

Tearing is often very effective. For most real models it reduces the size of blocks of over a hundreds of variables to fewer than a dozen.

3.2 A Simple Tearing Example

Tearing works by breaking algebraic loops. Often blocks are caused by algebraic loops. That is the case in the earlier example in Figure 2.3.

Lets take a very simple example

$$a = f_1(b)$$
$$b = f_2(a)$$

Here we have a algebraic loop shown in Figure 3.1. Our goal is to somehow break this loop as is shown in 3.2.

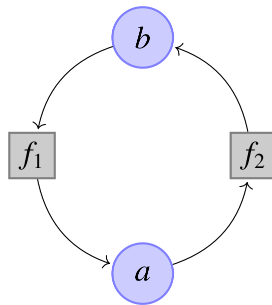


Figure 3.1: A graph that represents the equation system. From the graph it is easy to see the circular dependency of the system. The variable b is used as input to function f_1. This gives variable a that is then used as input to function f_2. This gives variable b and so on.

A simple tearing algorithm is described in [3]. We simply use the graph in Figure 3.2c to write the problem as

$$b_i = b_{i+1}.$$

The graph then gives the following equation

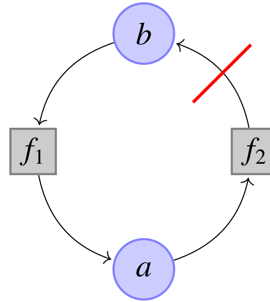
$$f_2(f_1(b)) = b.$$

this allows us to write the problem on the following format

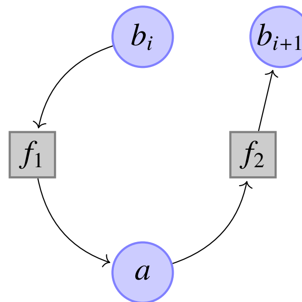
$$b - f_2(f_1(b)) = 0.$$

This can now be solved using a Newton optimizer where b is the variable that is being optimized for. The dimension of the optimization problem has been reduced from 2 to 1.

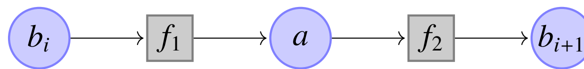
However this rewritting only works when we have a very simple graph with only a single linear dependency. We need a more general way of doing tearing. The one used by the OCT compiler is Node Tearing from [3].



(a) The edge between node f_2 and node b is marked for tearing.



(b) The edge marked for tearing is torn which creates a new graph.



(c) The torn graph is unfolded into a line. This is done to make the graph more readable.

Figure 3.2: An illustration of the simple tearing algorithm.

3.3 Node Tearing

Assume that we have the equation system

$$g_1(a, b) = 0$$
$$g_2(a, b) = 0$$

The goal is now to rewrite the problem so that it is possible to find a solution by only optimizing for one variable. A first step towards this is to calculate the inverse of one of the functions, let's say g_1, with respect to one of the variables, let's say a. We get

$$g_{1_INVa}(b) = a$$
$$g_2(a, b) = 0$$

It is now possible to solve the above system using by a one dimensional optimization algorithm in the following way. First we have a start value for b. This start value is then used

as input to $g_{1_INVa}(b)$ to get a new value for a. We can then use this new value as input to g_2. By using this value in $g_2(a, b)$ we turn g_2 from a two dimensional function to a function in one dimension, $g(b)$. We can then find a new value for b by using an optimization algorithm on $g(b)$ so that the function value gets as close to 0 as possible. This new value for b can then be used as input in $g_{1_INVa}(b)$ to get a new value for a and the process can then be repeated until the new value for b is the same or very similar to the value of b in the previous iteration. The process can be shown using a graph as can be seen in Figure 3.3. Here b_i is called an *iteration variable* and g_2 is called a *residual equation*. The variable a_i is called a *torn variable* and the equation g_i is called a *torn equation*. These have basically been *torn* from the optimization problem.

In Figure 3.3 the equation g_1 and variable a_i forms a *torn pair*. We use g_1 to calculate a_i. Similarly b_i and g_2 forms a pair. All residual equations must be paired with an iteration variable that contributes to the equation. All variables can only be paired with one equation and vice versa.

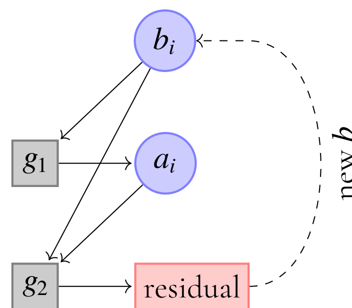


Figure 3.3: An illustration of node tearing.

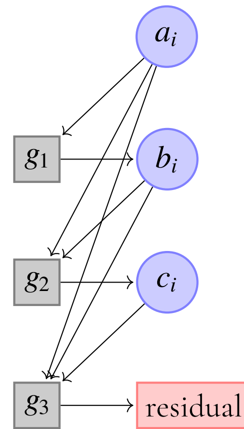
It is also possible to solve larger equation systems than the above using this method. For example assume that we have a system of three equations. Two possible tearings for a system of three equations are shown in Figure 3.4.

For a system of two equations there is only one way the tearing graph can look. If the number of equations are increased to three there are suddenly more options. We might have one or two residual equations. If we have two residual equations like in Figure 3.4b then we also have two iteration variables, in this case called a and b. The opimization problem becomes two dimensional and the Hessian is a 2x2 matrix.

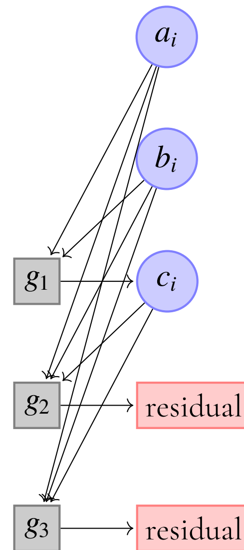
There are several reasons why there might be more than one residual equation. In Figure 3.4b we see that g_i and c_i makes a *torn pair*. For such a torn pair to be possible two things must be true. First the variable must of course exist in the equation. Secondly the variable must be linear in the equation. The reason for this is that the OCT compiler cannot calculate the inverse of an equation with regards to a variables that is not linear. For example a_i and b_i in 3.4b might not occur linearly in any of the equations. In that case we have no choice but to have two residual equations.

In both the examples in Figure 3.4 all the variables exists in all the equations. This is typically not the case. Usually equation systems in Modelica are sparse. For real models a block can contain hundreds of variables and equations but a typical equation often contains just a handful of variables. This puts a big constraint on how to do tearing because only a few if any variables are eligible to be torn variables for a specific equation.

One important goal when deciding how to do tearing is to reduce the number of residual



(a) An example of tearing in an equation system of three equations.



(b) An other example of a such system.

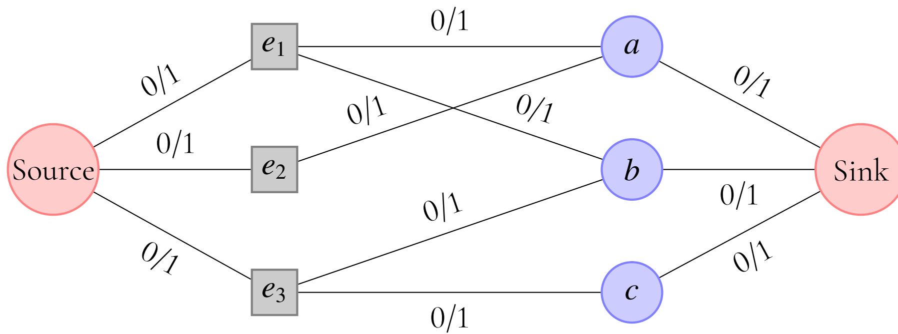
Figure 3.4: Two examples of tearing of systems of three equations.

equations as much as possible. This is the main reason why we do tearing; to reduce the dimension of the optimization problem. Unfortunately this problem is NP-complete [9]. There is also an other aspect of the tearing problem that is very important and that is the numerical properties of the torn system. For example as mentioned earlier the magnitude of the variables in newton optimization should be as close to each other as possible. It might be worth it to pick a solution that has more residual equations if the iteration variables are closer to each other in magnitude. This is just one of a magnitude of numerical considerations that must be taken into account.

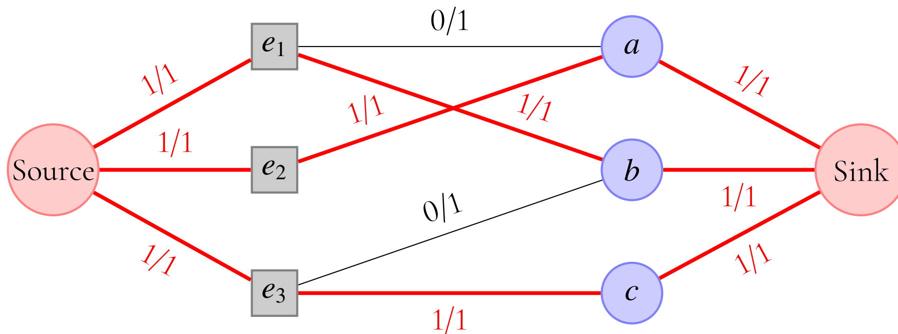
In OCT there is an algorithm called Automatic Tearing that tries to selects iteration variables and residual equations as well as possible.

3.4 Automatic Tearing

The automatic tearing algorithm in the OCT compiler contains many steps [10]. Most of them are not relevant for this thesis however there are a few things that are. The automatic tearing algorithm takes in one block at a time. One of the first steps in the automatic tearing algorithm is a bipartite matching of the equations and variables in the block. A new bipartite graph g is created and with the variables and equations that exists in the block. As in all bipartite graphs there are two sets of nodes, in this case one set consists of variables and the other of equations. In this graph g a new bipartite matching is done.



(a) First the graph is made to a network flow problem by giving all edges the capacity of 1 and adding a sink and a source.



(b) Secondly the max flow is found by some algorithm for example the ford fulkerson algorithm. The edges with flow in them (red edges) that connects a variable with an equation marks a matching.

Figure 3.5: An illustration of solving the bipartite matching problem using network flow.

All variables and equations have already been matched before automatic tearing. However unlike before in this new graph g equations and variables can only match if they can be torn pairs. For example if a variable occurs non-linearly in an equation they cannot match because the variable cannot be torn from that equation as it cannot be solved analytically by the OCT compiler.

There are two steps in this matching. First there is a greedy algorithm that iterates the equations and tries to match the equation with the first eligible variable in that equation. If there are still unmatched equations then a network flow algorithm is applied as is shown in Figure 3.5. Note that when solving the max flow problem augment flows are often used in the algorithm so that matchings from the greedy algorithm might be changed.

After these two matching algorithms there are usually equations that remains unmatched. These are gathered in a data structure with the variable name *SUME*. If an equations exists in this data-structure then this is a strong indication that it might be a good residual equation. For an example an equation with all variables non-linear will always end up in this data-structure because non of the variables can be torn.

In addition to this there is an algorithm than for a given equation compares two variables in this equation to see which is the best pairing. This algorithm has many heuristics and most of these heuristics checks numerical properties. There is also a similar algorithm that for a variable compares two equations.

Chapter 4

Contributions to Tearing

As seen in the previous chapter it is very hard to select good iteration variables and residual equations. Finding the maximum dimensionality reduction is a NP-complete problem. Furthermore numerical properties of the equation system must be take into account. In addition to this it is often important to pick iterationvariables that has good start values.

There are many complex problems that the automatic tearingalgorithm must solve and therefore it is important for the user to manually be able to select iteration variables and residual equations. Two different methods for this will be developed in this thesis. Both these methods can be used together with automatic tearing. The user selects some iteration variables and residual equations and the automatic tearing algorithm selects more if necessary.

4.1 Start Values

In addition to decreasing the dimensionality as much as possible and select iteration variables and residual equations with good numerical properties it is also important to take into consideration start values. For most but not all equation systems it is important that the variables selected as iteration variables has good start values. Most variables have start values but sometimes it is hard for the modeller to determine good start values for variables. It is also a lot of work to find good start values for all variables so if HGT makes it possible to avoid doing this work for a lot of variables then that is a good thing. If start values are bad then the system might not converge. If the start value for a variable is absent then the algorithm must find a good start value automatically and for some variables and equations they have numerical properties that makes it possible to find good start values automatically, but often this is not the case.

4.2 HGT for Dynamic Systems

HGT stands for Hand Guided Tearing. It means that the user manually can affect the selection of iteration variables and residual equations instead of letting the Automatic Tearing algorithm do all the work.

A part of this thesis is to make it possible to use HGT on dynamic systems. This is important because the vast majority of models are not steady state but contains a combination if algebraic and differential equations. For an example the code in Listing 4.1 is not a steady state model as there is a differential equation in that model. As can be see in this model the last equation must be a residual equation because none of the variables can be solved analytically by the OCT compiler. If c is the iteration variable paired with this equation then this is the only iteration variable necessary because a can be solved in the second equation and b in the third.

Listing 4.1: HGT example

```
model SimpleEx2
  Real x;
  Real a;
  Real b;
  Real c;
initial equation
  x = 5;
equation
  der(x) = 1 - x;
  0 = x/c^2 - a;
  0 = sqrt(c) + a^2 - b;
  0 = a^2 + b^2 + c^2 annotation(__Modelon(
     ResidualEquation(iterationVariable=c)));
end SimpleEx2;
```

Unlike steady state all equation that have circular dependency are not a part of this block. Only algebraic equations with circular dependency forms a block that we can do tearing on. In the model in Listing 4.1 the differential equation forms its own block. This block also looks different in the inital system and in the DAE. In the initial system x is always 5 but in the DAE the differential equation must be solved at every time step to get a value for x in that time step to feed into the block that we do tearing on.

Because the initial system and DAE has different properties the user might want to to tearing differently in these two systems. Therefore a part of this thesis is also to make it possible for the user to do different HGT tearing decisions for the initial and the DAE systems. To allow the user to do this a new *system* annotation should be introduced. For example if the user want a tearing pair to only be for the initial system the the user can use the annotaion in Listing 4.2. If the user wants it only for the DAE then the user can use the annotation in Listing 4.3, and if the user wants it for both then the user can use the annotation in Listing 4.4. The default is both so if the user does not write anything like in Listing 4.1 then it defaults to both.

Listing 4.2: Annotation for inital system only.

```
annotation(__Modelon(ResidualEquation(iterationVariable=
   c, system = "init")));
```

Listing 4.3: Annotation for DAE system only.

```
annotation(__Modelon(ResidualEquation(iterationVariable=
   c, system = "DAE")));
```

Listing 4.4: Annotation for both initial and DAE system.

```
annotation(__Modelon(ResidualEquation(iterationVariable=
   c, system = "both")));
```

4.3 Unpaired Tearing

In unpaired tearing the iteration variables and residual equations are selected individually as can be seen in Listing 4.5. Unlike normal HGT we do not specify the residual equation and iteration variable pair. We only specify which equations should be residual equations and which variables that should be iteration variables.

This can be used in combination with normal HGT. The OCT compiler requires there to be an equal number of unpaired iteration variables and residual equations selected in the OCT compiler. At compile time the unpaired tearing algorithm uses heuristics to match the unmatched HGT variables with the unmatched HGT equations. With HGT variables I simply mean an iteration variable used in HGT and for HGT equation i mean a residual equation used in HGT.

Listing 4.5: Unpaired tearing example

```
model SimpleEx3
  Real x;
  Real a;
  Real b;
  Real c annotation(__Modelon(IterationVariable));
initial equation
  x = 5;
equation
  der(x) = 1 - x;
  0 = x/c^2 - a;
  0 = sqrt(c) + a^2 - b;
  0 = a^2 + b^2 + c^2 annotation(__Modelon(
     ResidualEquation));
end SimpleEx3;
```

As for normal HGT this only works for steady state systems and a part of this thesis is to expand support so that it also works for dynamic systems. An other goal is to make unpaired tearing work even if the number of unpaired HGT variables and HGT equations are not the same. For example the modeller might only pick unpaired iteration variables or only unpaired residual equations. When the variables and equations are different we call this *uneven unpaired tearing*.

Unpaired tearing is often used at Modelon as something in-between automatic tearing and normal HGT. The user has some information about how to do tearing but not enough to know the best pairs. As such it is very useful to rewrite the Unpaired algorithm to make it possible to select an uneven number of HGT variables and HGT equations. For example the user might know several iteration variables that are good but not as many residual equations or not any residual equations at all. Instead of having to pick the same number of residual equations it is good if it is possible to rewrite the algorithm automatically can find suitable residual equations.

The new *system* annotations that was added for normal HGT were also added for unpaired HGT.

Chapter 5

Implementation

In this chapter I will describe the implementation of the two tools for HGT described in Section 4.

All code where written in Java. Sometimes the Java meta-compilation tool JastAdd [6] was used. JastAdd was mostly used for front-end changes in the OCT compiler that were necessary to create and propagate the new annotations. All the code for the algorithms in 1, 2, 3 are written in plain Java.

5.1 Hand Guided Tearing for Dynamic System

To make HGT for dynamic systems possible many small changes were necessary. Instead of mentioning many very minor code changes I will describe the one major change that was necessary. The major change was to make the init/DAE/Both annotation possible. As the code was written originally it made no difference between initial and DAE because there is no DAE part for steady state. It was not just the HGT algorithm that made no such difference but also a lot of methods from other classes that are used in this algorithm.

Instead of rewriting a lot of code I made a resetting algorithm and the same algorithm could then be called two times. This resetting algorithm marks a HGT pair for resetting if certain conditions are true. For example if the algorithm is called on the initial system and the HGT variable in the pair has the annotation init then the entire pair must be reset no matter if the equation has the annotation init or both. The variable should not be a HGT variable in the DAE system and the equation must be paired with an other variable.

5.2 Unpaired Tearing

The first step was to rewrite the current algorithm that pairs unpaired HGT variables and equations. This was rewritten so that it works even if the number of unpaired HGT variables and unpaired HGT equations are not the same. Several minor changes had to be made for this to work.

Algorithm 1 Graph matching

```
1: procedure GET_SUME_SUMEV(g)        ▷ g is a graph that describes the equation system
2:     <Greedy matching in g>
3:     <Network flow matching in g>
4:     SUME ← all unmatched equations in g
5:     SUMV ← all unmatched variables in g
6: end procedure
```

The next step was to write an algorithm that can identify good potential equations and variables that can be paired with the remaining unpaired HGT variables or equations. In this step I took a lot of inspiration from the automatic tearing algorithm, section 3.4. The same kind of greedy matching was first done followed by the same flow algorithm. The equations that where still unmatched after this were gathered in a datastructure with the variable name *SUME*. I also find the unmatched variables which where gathered in *SUMV*. The pseudocode for the second step can be seen in Algorithm 1.

The third step is to try to match the unpaired HGT variables or equations with those in *SUME* or *SUMV* using a greedy algorithm. If there are many options, let's say that an *SUME* equation contains many unpaired HGT variables then we use the algorithm from automatic tearing that compares variables in an equation. Similarly if we have several unpaired HGT equations for a variable the automatic tearing algorithm that compares equations for a variable is used. The pseudocode for this is the first procedure in Algorithm 2. If we have unpaired HGT equations left then these are if possible paired with the *SUMV* variables using a greedy algorithm. The pseudocode for this can be seen in the first procedure of 3.

At this point if there are still unpaired HGT variables or equations then we move on two procedure two in either Algorithm 2 or in Algorithm 3. In this step we first check if the unpaired HGT variables or equations have a matching in the graph g from Algorithm 1 after the flow algorithm. If this is the case then the unpaired HGT variable or equation is paired with its matching in g. If the HGT variable or equation is still not paired then a greedy algorithm is used to do the pairing as can be seen in the second part of the second procedure in Algorithm 2 or 3. If there are still unpaired HGT variables then a compiler error is returned.

The Init/DAE/Both annotation was implemented in the same way as for normal HGT.

The first procedure in Algorithm 2 and 3 is a weighted bipartite matching. Unlike Figure 2.4 the goal is not to make a perfect maximal matching but to match the surplus equations or variables. Furthermore the problem is weighted, i.e. the elements in the other set has different weights. If we have more variables then the eligible equations have different weights as they are not all equal good.

Algorithm 2 More variables

```
 1: procedure GREEDY_PAIRING_OF_SUME_EQUATIONS(g, SUME)
 2:     for <e ← next eqn in SUME> do                    ▹ Iterate all equations in SUME
 3:         bestIter ← null
 4:         for <v ← next variable in e> do                ▹ Iterate all variables in e
 5:             if <v is an unpaired HGT var> then
 6:                 if <v is better than bestIter> then
 7:                     BestIter ← v
 8:                 end if
 9:             end if
10:         end for
11:         if <bestIter ≠ null> then
12:             <Pair bestIter with e>
13:         end if
14:     end for
15: end procedure
16: procedure GREEDY_PAIRING_OF_REMAINING_UNPAIRED_HGT_VARIABLES
17:     for <v ← next unpaired HGT var> do   ▹ Iterate all remaining unpaired HGT
    vars
18:         e ← v:s matching in g
19:         if <e ≠ null> and <e ≠ paired HGT eqn> then   ▹ If v:s matching in the
    graph g exists and is not paired to a HGT var
20:             <Pair v with e>
21:             <BREAK>
22:         end if
23:         for <e ← next eqn containing v> do ▹ Iterate all equations that contains
    the variable v
24:             if <e ≠ paired HGT eqn> then
25:                 <Pair v with e>
26:                 <BREAK>
27:             end if
28:         end for
29:     end for
30: end procedure
```

Algorithm 3 More equations

```
 1: procedure GREEDY_PAIRING_OF_SUMV_VARIABLES(g, SUMV)
 2:     for <v ← next var in SUMV> do                          ▷ Iterate all variables in SUMV
 3:         bestEqn ← null
 4:         for <e ← next equation containing v> do            ▷ Iterate all equations containing v
 5:             if <e is an unpaired HGT eqn> then
 6:                 if <e is better than bestEqn> then
 7:                     BestEqn ← e
 8:                 end if
 9:             end if
10:         end for
11:         if <bestEqn ≠ null> then
12:             <Pair bestEqn with v>
13:         end if
14:     end for
15: end procedure
16: procedure GREEDY_PAIRING_OF_REMAINING_UNPAIRED_HGT_EQUATIONS
17:     for <e ← next unpaired HGT eqn> do        ▷ Iterate all remaining unpaired HGT eqns
18:         v ← e:s matching in g
19:         if <v ≠ null> and <v ≠ paired HGT eqn> then   ▷ If e:s matching in the graph g exists and is not paired to a HGT var
20:             <Pair e with v>
21:             <BREAK>
22:         end if
23:         for <v ← next var in e> do                         ▷ Iterate all variables in e
24:             if <v ≠ paired HGT var> then
25:                 <Pair e with v>
26:                 <BREAK>
27:             end if
28:         end for
29:     end for
30: end procedure
```

5.3 Alias Elimination

In addition to the BLT compilation step a small change was also necessary in this compilation step. This compilation step was modified so that a variable that is marked for HGT cannot be alias eliminated. The code for this was done in JastAdd.

Chapter 6

Evaluation

In this chapter the implementation of HGT is evaluated. I will evaluate both the normal HGT and Unpaired HGT. This evaluation will be carried out on three different sets of models; my own set of test models, models selected from the Modelica Standard Library (MSL) and two models provided from Modelon.

6.1 HGT for Dynamic Systems

To evaluate the implementation of normal HGT several things are done. First I have developed a test suit that tests HGT for many different dynamic systems. I also developed many different tests for the system annotation. In total this test test suit contains about 50 test cases that tests normal HGT. With the exception of one issue discussed in 6.3 and 7.3 every test that I wrote works. The problem with these tests however is that they are all very small models.

To test the implementation on real models I used 6 different models from the Modelica Standard Library (MSL) these are shown in Table 6.1. Even the models that are labeled small are far bigger than my own tests. I check if my implementation works by turning off Automatic Tearing and manually selecting the same iteration variables and residual equations selected by Automatic Tearing. Then I check that everything is the same as for automatic tearing including the simulation results. My implementation works for all these models.

In addition to the models from MSL I also got two models from Modelon that has problems with automatic tearing. The first model is a model of an evaporator. The problem with this model is that automatic tearing selects an iteration variable without start value which makes the model not converge. For some reason automatic tearing does this despite there being a similar variable that has a start value attached. I used HGT to select this variable as an iteration variable and pair it with the same residual equation. After this the model did converge.

The second model that I was provided from Modelon was a model of a heat exchanger.

MSL Models		
Model Name	Type of Model	Model Size
FullRobot	Mechanical	Very Large
ThreeTanks	Fluid	Small
BatchPlant	Fluid	Large
DoublePendulum	Mechanical	Small
CoupledClutches	Mechanical	Small
InvertingAmp	Electrical	Small

Table 6.1: The MSL models used. Small models have around 50-300 equations, the Large model around 2000 equations and the very large model around 5000 equations.

	2	3	4	5	6	7	8	9	10	11	12
Automatic	O	O	O	O	X	O	X	X	O	X	X
HGT	O	O	O	O	O	O	X	O	O	X	X

Table 6.2: Heat exchanger with automatic tearing and HGT results

The problem with this model is that the initial system does not converge with some start values with the iteration variables and residual equations selected by automatic tearing. The goal is to make the initial system converge for more start values by the use of HGT. In this model there is one parameter value that is problematic. This parameter models the convection heat flux to some of the pipes in the heat exchanger. For real world applications this parameter is in the range of 2-12. As can be seen in Table 6.2 there are 5 values for which this does not converge.

In this heat exchanger there are over a hundred parameters for different pressures. These pressures can have very different magnitudes. After talking with a modeller at Modelon the most likely reason why this model has problems at converging is because automatic tearing selects pressure parameters with to big differences in magnitude. I used HGT to reduce the differences in magnitude as much as possible and the result can be seen in Table 6.2. As can be seen the model converges for more start values in this interval but still has problems.

6.2 Unpaired Tearing

The Unpaired Tearing was evaluated in a similar manner as normal HGT. I developed a test suit that checks if unpaired tearing works. The same issue as for normal HGT exists here which will be discussed in 6.3 and 7.3.

In addition to this I tested Unpaired Tearing on the same MSL models. Unlike normal HGT three different kinds of test were done for each model. I tried to select both variables and equations. The same variables and equations as for automatic tearing. I also tries to select only variables and only equations. I then tested that everything worked and that the simulation results where the same as for automatic tearing. This was the case. I also tested Unpaired tearing on the same two models from Modelon with the exact same result.

6.3 Linear Blocks

In the OCT compiler there is an optimization that gets rid of linear blocks. A linear block is a block where every equation only contains linear variables. On these blocks normal node tearing should not be done. Instead there are more efficient algorithms that can be applied [3]. The optimization algorithm that removes linear blocks are done before the BLT step and involves among other things Alias Elimination. Unfortunately the change made to Alias Elimination in Section 5.3 means that if HGT is used in a linear block then two things can happen; either normal node tearing is done on this block and everything works but suboptimally or we might get a compiler error.

6.4 Compile time and Runtime

To check how my implementation affects compile time I compiled the *FullRobot* model 50 times and threw away the first 20 to avoid measuring the first few runs before the cache memory starts to have significant effects. On the 30 measurements that where not thrown away I measured the average time and a 95% confidence interval for the average time. I did this for Automatic Tearing, normal HGT and Unpaired Tearing, the results can be seen in Table 6.4. Neither normal HGT or Unpaired Tearing had a statistically significant difference from Automatic tearing. I also measured the runtime in the same manner, the result can be seen in Table 6.3. As can be seen there is no statistical significant difference from Automatic Tearing. It was expected that the compile time would not be affected because the tearing algorithm is a very small part of the compiler.

Runtime			
Model	Mean	Lower Bound	Higher Bound
Automatic	0.187	0.178	0.196
Normal HGT	0.193	0.182	0.205
Unpaired vars	0.185	0.176	0.194
Unpaired eqns	0.191	0.181	0.200
Unpaired both	0.195	0.185	0.205

Table 6.3: The runtime for the different ways to do tearing on the FullRobot model. Time is calculated in seconds. Confidence intervals are normal two sided 95% confidence intervals. As can be seen none of the tearings are statistically significantly different from automatic tearing because all the mean values are within the confidence interval of automatic tearing.

Compile Time			
Model	Mean	Lower Bound	Higher Bound
Automatic	31.02	30.92	31.11
Normal HGT	30.93	30.85	31.02
Unpaired vars	31.04	30.95	31.13
Unpaired eqns	31.09	31.01	31.17
Unpaired both	31.08	31.00	31.17

Table 6.4: The compile time for the different ways to do tearing on the FullRobot model. Time is calculated in seconds. Confidence intervals are normal two sided 95% confidence intervals. As can be seen none of the tearings are statistically significantly different from automatic tearing because all the mean values are within the confidence interval of automatic tearing.

Chapter 7

Discussion

7.1 Uneven Unpaired Tearing

As mentioned in Section 5.2 the unpaired tearing algorithm is a weighted bipartite matching problem. Weighted matching of Bipartite graph is a NP-complete problem [5] but greedy matching is the best known polynomial-time deterministic algorithm and often provides good approximations [2]. Flow algorithms cannot solve this problem [2] unlike the non-weighted case. However as can be seen in [2] there are stochastic algorithms that on average performs better than greedy matching.

Ultimately I decided to use greedy matching because the stochastic methods are much harder to implement while providing only marginal improvements. Furthermore it must be easy for the modeller to use the unpaired tearing algorithm. If the algorithm is stochastic there is a risk that for the very same model the algorithm will occasionally work and occasionally not work. It is also possible that the very same model would have different numerical properties for different compilations and that the simulation of the very same model would be different.

When deciding on what algorithm to use to solve this problem I brainstormed several options. All of them were of course NP-complete because tearing is NP-complete. The reason why I ultimately decided to use weighted bipartite matching is because the polynomial-time deterministic algorithm that give best approximations for the weighted bipartite matching is simple. For the other algorithms the algorithms required are more complex that a greedy algorithm.

If the greedy matching fails I try to match the variable or equation with its matching in g if possible. From the tests that I did it is very rare that the variable or equation fails to be matched in the greedy algorithm unless I intentionally picked really poor variables as iteration variables or really poor equations as residual equations. In the cases where a variable or equation failed to be matched in the greedy algorithm it could almost always in my tests be matched in to its pairing in g unless I specifically constructed a problem to prevent it.

The reason why an equation or variable that fails to be matched in the greedy algorithm is matched with its pairing in g is that it will not mess up an other equation or variable. If we instead would have used a greedy algorithm to do the matching then there is a risk that it will mess up an other pair. For example if we have variable x that is a torn variable and can only form a torn pair with equation e then there is no risk that we will match y with e. With a greedy algorithm this risk exists. If against all odds the variable or equation is still unmatched after this then I match it using a greedy algorithm.

7.2 The two Models from Modelon

There where only two real models that had problems working with automatic tearing. One of the reason for this might be selection bias. For steady state OCT has had support for HGT for a few years. For steady state there are many models that utilises HGT and in many of these models automatic tearing is also turned off. Since model developers have had no option apart from full automatic tearing for dynamic system it is not unlikely that they have been tweaking and changing their models so that they actually work when using the automatic tearing algorithm. HGT for dynamic systems have been requested by model developers at Modelon so it is unlikely that it will not be used. Once there is support for this then the number of dynamic models that have problems working solely using Automatic Tearing might start to increase.

It is hard to determine why there are still start values for which the heat exchanger does not converge. It could be that the magnitude difference is still too big and could be reduced by using other variables as iteration variables. But there are a lot of different numerical problems that can occur. Since this thesis is about implementing two different HGT tools and not about numerical analysis it felt out of scope of this thesis to investigate the numerical properties of the Heat Exchanger further.

7.3 Linear Blocks

There is a problem with linear blocks. Ideally HGT should not be possible to do on linear blocks. There is no reason to do normal node tearing on linear blocks because much better algorithms are available. However changing the optimization for linear blocks so that it ignores HGT annotations or gives warnings would be time consuming as it would require me to get a good understanding of an other compile step. Furthermore it is likely that a significant rewrite would be needed because at this compile step the compiler has not yet partitioned the model into smaller blocks. So the compiler does not know what block an equation belongs to or if there are linear blocks. Because of that it was considered out of scope for this thesis.

As it is now the algorithm can give compile error. However this is not considered as a very big problem for several reasons. First it is very unlikely that the modeller would use HGT on linear blocks. HGT is used when automatic tearing for some reason fails and since automatic tearing is not applied to linear blocks these blocks are irrelevant. Secondly if there is an error the error is at compile time and the modeller gets an error message that explains what is wrong. It would be more problematic if the error was during the simulation, especially if

there is no error message. Then the modeller could get a simulation result that is wrong and not know that it is wrong.

7.4 Selection of MSL Models

In Table 6.1 there are six models used in the evaluation. Almost all models at Modelon has fluid or mechanical components. Therefore several fluid and mechanical models were picked. An electrical model was also picked because electrical components are also common. Furthermore it is good to have both smaller and bigger models. The benefit of smaller models is that it is possible with a days work or so to get an overview of the entire model. So if there are bugs in simulation or compilation it is much easier to find than if only using very large models.

The reason why FullRobot was used to evaluate the compile time and runtime is because this model is by far the biggest of the six MSL models and has some very large blocks. This is especially important for the evaluation of uneven unpaired tearing because the weighted bipartite matching problem is NP-complete. So there are methods to find approximations to this problem would give a very large increase in compile time for big blocks.

7.5 Tearing in OpenModelica

The OpenModelica compiler also supports hand guided tearing for dynamic systems [4]. However OpenModelica only supports the selection of variables. Not equations or variable and equation pairs.

To evaluate the usefulness of the additional capabilities of the OCT compiler it would be necessary to find examples where we need to specify the equations. Either where it is not enough to only specify variables but we would need to specify equations as well or an example where only equations are to be specified and we have very little information about variables.

It can be hard to find such an example. In the two Modelon models the problems that were fixed were numerical issues with the variables. However it was still possible but very time consuming to fix these numerical problems with only selecting equations. I had to select equations so that my algorithm would likely pick the correct variables.

The main benefit of being able to select both variables and/or equations are probably to save time.

Chapter 8
Conclusion

Tearing is a tool that is important in the OCT compiler for both optimization purposes and to give equation systems good numerical properties to increase the simulation accuracy. The problem is that it is very hard to develop a good tearing algorithm and therefore it is helpful for the to have a way for the user to manually select tearing variables and equation. This is called Hand Guided Tearing (HGT). In this thesis I developed two different HGT tools. The first tool is an extension of the current HGT so that it also works for dynamic systems. The second is a tool that is a mix of normal HGT and Automatic Tearing. The main purpose of these tools is to complement automatic tearing when automatic tearing does bad decisions but it is also possible to turn automatic tearing off and only use these tools. The evaluation shows that the tearing tools works as intended and provides no statistically significant increase to compile time.

Further work could include fixing linear blocks. It could also include replacing the deterministic greedy matching in unpaired tearing with a stochastic algorithm. It could also include making the weights in the weighted bipartite matching in unpaired tearing better. Since the maximum dimensionality reduction problem is NP-complete most of the effort of making better weights should probably be spent on making the heuristics for numerical properties as good as possible. This would also improve automatic tearing.

References

[1] Carnegie Mellon University. `https://www.cs.cmu.edu/~15451-f18/lectures/lec19-DFS-strong-components.pdf`. *Lecture 19: Depth First Search and Strong Components*, 2018.

[2] Ran Duan and Seth Pettie. Linear-time approximation for maximum weight matching. *Journal of the ACM, Volume 61, Issue 1*, 2014.

[3] Hilding Elmqvist, Dynasim Ab, and Martin Otter. Methods for tearing systems of equations in object oriented modeling. In *In ESM'94 European Simulation Multiconference*, pages 1–3.

[4] Peter Fritzson, Adrian Pop, Karim Abdelhak, Adeel Ashgar, Bernhard Bachmann, Willi Braun, Daniel Bouskela, Robert Braun, Lena Buffoni, Francesco Casella, Rodrigo Castro, Rüdiger Franke, Dag Fritzson, Mahder Gebremedhin, Andreas Heuermann, Bernt Lie, Alachew Mengist, Lars Mikelsons, Kannan Moudgalya, Lennart Ochel, Arunkumar Palanisamy, Vitalij Ruge, Wladimir Schamai, Martin Sjölund, Bernhard Thiele, John Tinnerholm, and Per Östlund. The OpenModelica Integrated Environment for Modeling, Simulation, and Model-Based Development. *Modeling, Identification and Control*, 41(4):241–295, 2020.

[5] Xiangyu Luo Guohun Zhu and Yuqing Miao. Exact weight perfect matching of bipartite graph is np-complete. *Proceedings of the World Congress on Engineering 2008 Vol II*, 2008.

[6] Niklas Fors Jesper Öqvist Görel Hedin, Emma Söderberg. `https://jastadd.cs.lth.se/`.

[7] H.H. Happ. *Diakoptics and Networks.* Academic Press, 1971.

[8] Gabriel Kron. *Diakoptics - The Piecewise Solution of Large-scale Systems.* MacDonald Co., 1963.

[9] Richard S.H Mah. *Chemical Process Structures and Information Flows.* Butterworth-Heinemann, 1990.

[10] Patrik Meijer. Tearing differential algebraic equations. *Lund University*, 2011.

[11] Modelica Association. Modelica specification. `https://specification.modelica.org/master/MLS.html`.

[12] Paul Rizescu. Applying optimization algorithms in a modelica compiler. *Lund University*, 2014.

[13] Stanford University. Newton-type methods. `https://web.stanford.edu/class/cme304/docs/newton-type-methods.pdf`.

EXAMENSARBETE Improving Tearing in a Modelica Compiler
STUDENT Oskar Kari
HANDLEDARE Niklas Fors (LTH), Filip Stenström (Modelon), Markus Olsson (Modelon)
EXAMINATOR Görel Hedin (LTH)

Förbättra Tearing i Modelica

POPULÄRVETENSKAPLIG SAMMANFATTNING **Oskar Kari**

I detta exjobb implementeras två olika algoritmer som gör att simuleringar i programmeringsspråket modelica får potential att bli snabbare och mer precisa. Dessa algoritmer testas på riktiga modeller och innebär en förbättring gentemot tidigare.

En kompilator översätter programmeringskod som människor skriver till instruktioner som hårdvaran i datorn förstår. Det är en slags “brygga” mellan mjukvaran och hårdvaran. Det finns flera kompilatorer för språket Modelica men den kompilator som jag arbetat med i detta exjobb är en kompilator som företaget Modelon utvecklat.

Modelica är ett programmeringsspråk som används för att simulera fysiska system. Det kan vara elektriska komponenter som exempelvis en diod, eller mekaniska komponenter som bromsar. All programmeringskod som skrivs i modelica beskriver ekvationssystem. I modelica så finns det finns det två olika typer av ekvationssystem. Det finns statiska system. Dessa måste lösas en gång och förändras inte med tiden. Sen finns det dynamiska system. Dessa måste lösas massor med gånger för olika tidsvärden eftersom ekvationerna påverkas av tiden.

Ett problem med dessa ekvationssystem är att de ibland kan vara väldigt stora. De kan ibland innehålla tusentals ekvationer. När antalet ekvationer ökar orsakar detta två olika problem. För det första ökar tiden det tar för datorn att utföra simuleringen markant. Det andra problemet är att man får oftast beräkningsfel som riskerar att växa med ekvationssystemets storlek.

Tearing är en metod som bryter ner ekvationssystemet så att det blir mindre. När man genomför tearing på ett ekvationssystem försvinner flera av ekvationerna och variablerna från ekvationssystemet och systemet blir mindre. Efter att man löst det mindre ekvationssystemet är det möjligt att genom tearingalgoritmen få fram värdet på de variabler som försvann från ekvationssystemet.

Att skriva en algoritm som genomför tearing på ett bra sätt automatiskt är mycket svårt, bland annat är ofta domänkunskaper viktiga. Dvs om man exempelvis bygger en model över en diod så måste man ofta ha expertkunskaper inom elektroteknik för att kunna göra bra tearing.

I Modelons kompilator så finns det en algoritm som försöker göra tearing på alla ekvationssystem som innehåller fler än en ekvation. Problemet är att denna algoritm ibland gör dåliga val. Av den anledningen behövs ett sätt för den som skriver koden att manuellt göra tearing. För några år sedan infördes stöd för detta i kompilatorn, dock bara för statiska system. En del av detta exjobb är att utvidga stödet så att det fungerar även för dynamiska system. En annan del är att införa en ny algoritm som ligger mellan automatisk tearing och manuell tearing. Ibland har programmeraren viss information om hur man gör bra tearing men inte tillräckligt för att göra allting manuellt.

Mina tearingalgoritmer testades på modeller från företaget Modelon där automatisk tearing inte fungerade eftersom beräkningsfelen blev för stora. Med hjälp av mina algoritmer minskades felen rejält.