Maintaining Source Origin in a Modelica Compiler
(Master thesis)

Alexander Åhlander
ahlander.alexander@gmail.com

March 29, 2017

Master’s thesis work carried out at
the Department of Computer Science, Lund University.

Supervisors: Jon Sten jon.sten@modelon.com,
Jonathan Kämpe jonathan.kampe@modelon.com,
Niklas Fors, niklas.fors@cs.lth.se

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

Modelica is a modelling language used to describe a physical system and can be used to simulate the system. A Modelica compiler transforms a Modelica model into an equation system in order to simulate it. During this transformation the information about where the equations originates from is lost. This information could be useful to have when, e.g., trying to locate an error. In this thesis we solve this problem for the JModelica.org Modelica compiler by propagating information about the source origin through the compiler. This makes it possible for us to show the origin of an error. We have also made sure that the connections between the transformed model and the original model can be viewed even when no error has occurred in a HTML document. The increase in memory usage in the current implementation is well within acceptable levels, with the highest memory increase measured being only 0.18%.

Keywords: Modelica, source tracking, compilers, multiple sources
Acknowledgements

I would like to thank my supervisors Niklas Fors, Jonathan Kämpe and Jon Sten for their support and feedback during this project. I would also like to thank my examiner Görel Hedin and everyone at Modelon AB.

Thank you very much!
# Contents

1 Introduction ................................................. 7  
1.1 Related Work ............................................. 8  

2 Background .................................................. 9  
2.1 Modelica ................................................... 9  
2.2 JModelica.org .............................................. 11  
2.2.1 Flattening .............................................. 12  
2.2.2 Transformations ......................................... 12  

3 Source Origin ............................................... 15  
3.1 Motivating Example ....................................... 16  
3.2 Multiple Sources ......................................... 16  
3.3 Solution .................................................... 18  
3.3.1 Error Messages ......................................... 18  
3.3.2 HTML Presentation ..................................... 19  

4 Implementation ............................................. 21  
4.1 Internal Representation of the Source .................... 21  
4.2 Multiple Sources ......................................... 22  
4.3 Source Propagation ....................................... 22  
4.3.1 Design Patterns ........................................ 23  

5 Evaluation ................................................... 25  
5.1 Completeness .............................................. 25  
5.2 Memory Usage ............................................. 25  
5.2.1 Measurements ........................................... 26  
5.2.2 Future Improvements .................................. 28  
5.3 Execution Time ............................................ 29  

6 Conclusion .................................................. 31  
6.1 Future Work ............................................... 31
Chapter 1

Introduction

Modelica is a modelling language used to describe different kinds of physical systems such as mechanical, electrical or hydraulic systems [1]. A model in Modelica is often used to simulate and collect data about the system.

JModelica.org [2] is an open source tool for the Modelica language, which includes a compiler for Modelica. The JModelica.org compiler transforms the model several times to generate the intermediate code, which will then be used for simulation. During these transformations the connection between the original model and the final model is lost. Not having a connection between the original and final model is a problem if an error occurs and the user needs to find and correct the error in the original model. For example, a user might receive an error during simulation telling them that it failed due to a division by zero, with the error being expressed in terms of the transformed model. The user will then need to find the corresponding error in the original model by comparing it to the transformed model.

It can be trivial to find the connection between the final transformed model to the original model if the code that caused the error was not modified during compilation. While this is true when only minor modifications were made during compilation, it can be a tedious task if the code where the error occurred was heavily modified during compilation. The user would have to guess from which file an equation originates from based on the name of the variables in the equation. They would then have to search the file for the definition and or uses of those variables. Things can become a bit more difficult if, e.g., the variables have been replaced with temporary variables. Because of this, having a solution to automatically find the source of the transformed model should be useful.

As large models are common, it is also worth paying attention to performance and memory usage when propagating and storing the source information.

To solve this, we modified the JModelica.org compiler to make sure that the source information gets propagated through the entire compilation. We also made sure that multiple sources are supported, as two or more expressions are sometimes transformed into one expression. The source information is displayed in the form of an HTML representation of...
the transformed model. The implementation results in an increase in memory usage which is within acceptable levels, with the highest memory usage difference measured being a 0.18% increase.

The structure of this thesis is as follows. Section 2 introduces the Modelica modelling language and the JModelica.org compiler. Section 3 contains a more detailed problem description. The implementation is described in Section 4 and is evaluated in Section 5. Finally, Section 6 contains the conclusions which can be made from this thesis project.

1.1 Related Work

Propagating the source information through the transformations of a compiler is common in debuggers. Implementing a debugger for JModelica.org would be too large of a task for a thesis project, which means that most debugger solutions does not provide much useful information. The OpenModelica Equation Model Debugger [6] is an example of a debugger for the Modelica language, used by the compiler in the OpenModelica [5] open source project. It uses tracing [7], which means that it logs what the transformation steps in the compiler does, which allows the debugger to trace a part of the code back through the transformations which were applied to it.

While not easily applicable to an already existing compiler, there is work on the theory around origin tracking [8] on which a new compiler could base its source propagation on.
Chapter 2
Background

We will in this chapter describe the modelling language Modelica and the compiler JModelica.org, both of which are a very large part of this thesis.

2.1 Modelica

Modelica is a language for modelling of physical systems. The language is declarative, meaning that the code does not have a predefined execution order, as the system is described using mathematical equations. It is also object-oriented, which allows users to create larger models with the help of smaller models, which we will refer to as component models. This is to separate them from the larger model which describes the entire system. A component model can also be a complete model itself but is referred to as component model when it is used as a part of another model.

Modelica models consists mostly of equations but can also contain regular conditional statements and functions. Functions in Modelica have no side effects, which means that an input value will always result in the same output, since they are not allowed to have an internal state. Because Modelica is object-oriented, the code can be split into smaller component models which are usually specified in multiple files for larger systems.

The different files and the components in a file are merged together by the compiler with a transformation called flattening, resulting in a large flat model, i.e., a model without the object-oriented structure. This is needed to be able to simulate multiple models as a single system.

Listing 2.1: Example

```model Water
    Real h = sin(time*10);
end Water;
```
Listing 2.1 shows a model where a ball is falling into water. The height of the water is described as a sine wave. The FallingBall model is only a container for the other two models and has no equations of its own. The Ball model contains most of the equations of the model, with a starting height of the ball at 10 m, the starting velocity of 0 m/s and the gravitational constant of 9.82 m/s². The model also contains the derivative equations used to describe the height and velocity of the falling ball. The Water model contains the equation for the height of the waves, which depends on the time variable, which is built into the Modelica language. While the interaction between the ball and the water is unrealistic in this example, the model can be used to find out when the ball hits the water and at what height, as seen in Figure 2.1, where the simulation results are plotted.

![Figure 2.1: Simulation results of the FallingBall model](image-url)
2.2 JModelica.org

JModelica.org is an open source Modelica tool, initially developed at Lund University, together with the company Modelon AB. It is currently being developed by Modelon AB.

The JModelica.org tool contains a Modelica compiler. The compiler is written in JastAdd [3], which is a meta-compiler system for creating compilers. It uses JastAdd code to build a compiler written in Java. An abstract syntax tree (AST) is created in the JModelica.org compiler by parsing the model code. The AST is a way to represent the code as a tree, which is needed by some transformations. An example of the equation $y = x + 1$ being represented as a part of an AST can be seen in Figure 2.3. The nodes of the tree contain different information depending on what the node represents. An $\text{add}$ node only needs to know its two children, but a $\text{var}$ node needs to know its name and which equation defines its value. The intermediate code is generated after the transformations, which is then used for the actual simulation. An overview of this process can be seen in Figure 2.2, where the simulation results are plotted.

![Figure 2.2: Overview of the JModelica.org compiler](image)

![Figure 2.3: The equation $y = x + 1$ represented as an abstract syntax tree](image)
2. Background

2.2.1 Flattening

An important part of a Modelica compiler is called flattening. As described previously, flattening merges a model and its component models into one large model without object-oriented structure, resulting in a large equation system. Even if the model consists of components from several different files, the result is still a single equation system. The components of the model are expanded into new equations to remove the object structure and produce an entirely flat model, i.e., a model with all variables and equations being declared on a row each. Inside the JModelica.org compiler, flattening involves creating a new AST and throwing the old one away, rather than modifying the old AST. This is relevant because the source information of the nodes are present in the old AST, but is not copied to the new tree when it is created. For example, the equation $x = 1$ initially contains its source information. However, during flattening, the equation $x = 1$ is created in the new AST without the source information, regardless of whether or not flattening the model altered the content of the equation.

Listing 2.2 shows the FallingBall example after the flattening. The variables are given unique names based of the object-oriented structure the original model had. For example, the height of the ball is called $b.h$, as it is the height $h$ of the ball $b$. The three equations of the model are now located next to each other. The flattening also added two initial equations for $b.h$ and $b.v$ which contains the same information as the start values in the variable declaration of respective variable.

Listing 2.2: FallingBall after flattening

```model FallingBall
    Real b.h(start = 10);
    parameter Real b.g = 9.82 /* 9.82 */;
    Real b.v(start = 0);
    Real w.h;

    initial equation
    b.h = 10;
    b.v = 0;

    equation
    der(b.h) = b.v;
    der(b.v) = - b.g;
    w.h = sin(time * 10);
end FallingBall;
```

2.2.2 Transformations

After flattening, a number of transformations are made by the compiler to optimize the flattened model or to make changes necessary in order to produce the intermediate code. These transformations changes the AST in different ways, which can result in loss of source information. We will describe some of the more relevant transformations here.

Alias Elimination

Alias elimination is an optimization in the JModelica.org compiler [4]. The optimization is similar to a common compiler optimization called copy propagation which replaces every
use of $x$ with the variable $y$ when it finds a statement where $x = y$. Alias elimination also includes the case $x = -y$ in JModelica.org. For the example in Listing 2.3 the variable $a$ is removed because of the equation $b = a$, and all the uses of $a$ are replaced with $b$. The equation $b = a$ is also removed since it is not needed.

**Listing 2.3:** An example of Alias Elimination.

\[
\begin{align*}
\text{model } A & \quad \text{model } A \\
\text{Real } a, b, c; & \quad \text{Real } b, c; \\
\text{equation} & \quad \text{equation} \\
\quad a = \text{time}; & \quad b = \text{time}; \\
\quad b = a; & \quad c = b \times b; \\
\quad c = a \times b; & \quad \text{end } A; \\
\text{end } A;
\end{align*}
\]

**Variability Propagation**

Variability propagation [4] is an optimization in the JModelica.org compiler which is based on constant propagation, which is a common optimization in compilers. Variability propagation is used to try to evaluate variables during compile time instead of during simulation. It will mark a variable as a constant if it is assigned a value which can be evaluated as a constant. In Listing 2.4 the compiler checks to see if $x$ is constant, which it is since $x = 2$. When it checks $y = x + 5$, it finds that both $x$ and 5 are constants, which means that $y$ is also a constant and the value of $y$ is calculated using constant folding, which is another common optimization. Constant folding calculates values of equations which are constant, resulting in simpler equations and removes the need to calculate the value every time the compiled code is run. In the JModelica.org compiler, constant folding is done as a part of variability propagation.

**Listing 2.4:** An example of Variability Propagation.

\[
\begin{align*}
\text{model } B & \quad \text{model } B \\
\quad \text{Real } x = 2; & \quad \text{constant } \text{Real } x = 2; \\
\quad \text{Real } y = x + 5; & \quad \text{constant } \text{Real } y = 7; \\
\text{end } B; & \quad \text{end } B;
\end{align*}
\]
2. Background
Chapter 3
Source Origin

When trying to simulate a Modelica model using JModelica.org, there is the possibility that an error will occur. When this happens, the user receives an error message from either the compilation or simulation. Errors occurring during simulation can only provide the location of the error in terms of the transformed flat model, and errors during compilation, after the flattening step, can only provide information in terms of the partially transformed flat model. The compiler can however provide accurate information if the error occurs in the early stages of the compilation, e.g., if a syntax error is found during parsing.

**Listing 3.1: Original Model**

```model A
    Real offset = 2;
    Real a = sin(time) + offset;
    Real b = 1 / a + offset;
end A;

model B
    Real x = 0.5;
    extends A(offset = 0.5);
end B;
```

**Listing 3.2: Flattened Model**

```model B
    Real x = 0.5;
    Real offset = 0.5;
    Real a = sin(time) + offset;
    Real b = 1 / a + offset;
end B;
```
Listing 3.3: Flattened and Transformed Model

```plaintext
model B
  constant Real x = 0.5;
  Real a;
  Real b;
  equation
    a = sin(time) + 0.5;
    b = 1 / a + 0.5;
end B;
```

3.1 Motivating Example

In the example in Listing 3.1, there will be an error during simulation if time reaches about 0.524, which will make \( a = 0 \), resulting in \( b \) being equal to 1 divided by zero. A user receiving this error will at most know that there is a problem with the equation which defines \( b \). Any additional information would have to be gathered by tracking the definitions of related variable uses in the model. Listing 3.1 shows the original model, where model \( B \) extends \( A \) and modifies the \texttt{offset} variable, defined in \( A \), to 0.5. Listing 3.2 shows the model after flattening, where the two models have been merged together into one model, just called \( B \), and the old default value of \texttt{offset} has been replaced with 0.5. Listing 3.3 shows the model after the transformations, which in this example have determined that the variables \( x \) and \texttt{offset} are identical and removed \texttt{offset}. Because \( x \) and \texttt{offset} are also constant, all the uses of them have been replaced with the literal 0.5.

While trivial in this small example, being able find out where, for instance, the 0.5 in the equation \( a = 0.5 \) in the flattened model originates from can potentially be very time consuming, depending on the size of the model and how much the compiler has transformed the code. Replacing a variable with another variable with the same value and adding temporary variables and similar changes are very common and can cause problems if one wants to backtrack to find a problem.

Large models usually contain a large amount of component models which are defined in different files. Figuring out the origin of an error is made a lot harder due to the fact that the compiled model is flattened, and does not contain any information about which equation or variable of the compiled model originates from which file. Since large models use a lot of memory during compilation, running out of memory is a potential problem if a change in the compiler increases the memory usage.

3.2 Multiple Sources

Some transformations merge several expressions into one expression. While the information lost is not very relevant during more simple transformations such as constant folding, e.g., \( y = 2 + 3 \) being folded into \( y = 5 \), it can be interesting for other transformations. For example, if you do variability propagation on the equation \( y = 2 + x \), where \( x = 3 \), the result is also \( y = 5 \). However, in the second example, the source of \( x \) is not the same as the source of the literal 2.
We have found three transformations in the JModelica.org compiler where two expressions result in one expression. They are constant propagation, alias elimination and common subexpression elimination.

Constant propagation has the problem with multiple sources in the way described previously. While it is trivial by itself, combined with other transformations such as variability propagation, can make things more complicated. Listing 3.4 illustrates another example of this, where the connection between the value 5 and the variables \( a \) and \( b \) is not obvious. It is worth noting that constant propagation is a part of variability propagation in JModelica.org.

**Listing 3.4:** An example of Constant Propagation.

```plaintext
model B
  Real a = 2;
  Real b = 3
  Real c = a + b;
end B;

constant Real a = 2;
constant Real b = 3
constant Real c = 5;
end B;
```

Alias elimination is another transformation where the result loses information about one use of the variables. Listing 3.5 shows an example of this transformation. Since the compiler removes the variable \( a \) when it finds that \( c = a \), all information connecting \( b = c + c \) with \( a \) is lost.

**Listing 3.5:** An example of Alias Elimination.

```plaintext
model A
  Real a, b, c;
equation
  a = time;
  b = a + c;
  c = a;
end A;

Real b, c;
equation
  c = time;
  b = c + c;
end A;
```

The last transformation which can cause this kind of loss of information is called common subexpression elimination [9]. Two or more identical expressions are stored in a temporary variable instead of being calculated twice. The expression is replaced with a use of the temporary variable. In Listing 3.6 the function call \( \text{func(time)} \) is used in the equations for both \( a \) and \( b \). Instead of calling \( \text{func(time)} \) twice during simulation, the result is stored in the temporary variable \( \text{temp}_1 \). This results in the function only being called once.

**Listing 3.6:** An example of Common Subexpression Elimination.

```plaintext
model A
  Real a, b;
equation
  a = \text{func(time)} + 1;
  b = \text{func(time)} + 2;
end A;

function func
  ...

model A
  Real a,b,temp_1;
equation
  a = temp_1 + 1;
  b = temp_1 + 2;
  temp_1 = func(time);
end A;

function func
  ...
```

While it might be possible to find one source which is more important than the others...
in cases like these, we decided that we should save both sources instead. There are two possible downsides of this. It requires more memory, since more information is stored. This will be described more in chapter 4. The other downside is that it can cause problems when presenting the sources, especially if the same expressions have a lot of sources. This could easily happen if constant folding is done to a large expression, e.g., \( x = 1 + 2 + 3 + \ldots + 49 + 50 \). This problem can be reduced by removing sources which are contained within other sources.

3.3 Solution

In this master thesis, we have managed to propagate most of the source information through the transformations in the JModelica.org compiler. The error messages and the debug output have been updated to include this information.

3.3.1 Error Messages

Since the source information was originally lost during flattening, error messages which occurred after the flattening did not contain the source. However, we noticed when propagating the source information through the transformations in the compiler, that the error messages automatically included the source location in the error messages if a source location was found. This means that no changes had to be made in order to support this. An example of an error message can be seen in Figure 3.1.

![Example of an error message with and without source information](image)

**Figure 3.1:** Example of an error message with and without source information
3.3 Solution

3.3.2 HTML Presentation

The debug output of the JModelica.org compiler includes a Modelica representation of the AST, which can be used to see what changes are made between each transformation. This is what is shown in Listing 3.2 and 3.3.

We modified this output to be printed in HTML instead of plain text, in order to more easily include the source information. Figure 3.2 shows the model from Listing 3.6 being printed after the transformations. It includes underlines which indicate that you can see where the underlined code originates from. The source of a part of the code can be seen by hovering over it with your mouse cursor as can be seen in Figure 3.2. The values which are shown after the filename is the line and column where a node begins and ends. There can be several underlines with different lengths under the same node. This indicates that there is a node with a different source than its parent in the AST, with the shorter underline marking the child and the longer underline marking the parent and all of its children. In Figure 3.2, $A.f(t)$ has two underlines because its source is different from the rest of the equation.

```model A
Real a;
Real b;
Real temp_1;
equation
  a = temp_1 + 1;
  b = temp_1 + 2;
  temp_1 = A.f(time);
end A;
```

Figure 3.2: Source is displayed when hovering over the code.
Chapter 4
Implementation

The transformations in the JModelica.org compiler lose source information in several different ways, which makes it hard to propagate the information using a separate module. Thus, a lot of time of this project went into making minor modifications to existing code, which we will not describe in detail. We will instead describe how the source is represented inside the compiler and how some of the problems we encountered were solved.

4.1 Internal Representation of the Source

The JModelica.org compiler stores the source as a string for the filename and a pair of integers for the position in the file. This is used for error output in the earlier stages of the compilation, before it is lost when the AST is flattened. Worth noting is that the position integers are automatically added by JastAdd, making it more difficult to modify, while the filename is manually added to the JModelica.org compiler.

Listing 4.1: The SourceLocation class

```java
public class SourceLocation{
    private String fName;
    private int row;
    private int column;
    public SourceLocation(String fName, int row, int column){
        setLocation(fName, row, column);
    }
    ...
}
```
4. Implementation

4.2 Multiple Sources

In order to support multiple sources, we created a simple class called \texttt{SourceLocation} to contain both the filename and the position, as can be seen in Listing 4.1. An array of these can then be used to represent multiple sources if necessary, replacing the previous string variable. However, storing every source, which was previously only represented by a string reference and two integers, now uses more memory even if it only has a single source. To reduce this problem we made the filename reference into an \texttt{Object} reference, which allows us to store the source in the same way as before on nodes until they need multiple sources. Referring to a string with the \texttt{Object} reference is preferred over referring to a \texttt{SourceLocation} object when multiple sources are not needed. This is because the strings of the filenames need one allocated object per file, while \texttt{SourceLocation} objects would need one allocated object per node, as these also include the integers.

Since we want to be able to store the source as either just a string or an array of \texttt{SourceLocation} objects, some modifications had to be made to the existing code in order to properly cast the source into a string.

Most of the changes used in order to propagate the source involves a method called \texttt{setLocation}, which copies the source information from one node to another. To be able to add additional sources to the same node, a new method would have to be written. Listing 4.2 shows a simplified version of this method, called \texttt{addLocation}. It uses a method called \texttt{mergeLocations} which returns an array with all the combined sources. It also removes duplicate sources and any source which is contained within another source. The destination node is then set to have the remaining nodes as source. If there is only one source left, the array of \texttt{sourceLocations} is discarded and only the string and integers are saved.

\begin{listing}
\begin{verbatim}
void addLocation(ASTNode sourceNode, ASTNode destNode){
    SourceLocation[] temp = mergeLocations(sourceNode, destNode);
    if(temp.length == 1){
        destNode.source = temp[0].getFileName()
        destNode.line = temp[0].getLine()
        destNode.column = temp[0].getColumn()
    } else if (temp.length > 1){
        destNode.source = temp;
    }
}
\end{verbatim}
\end{listing}

4.3 Source Propagation

The creation of new nodes without the source to replace old nodes during a transformation is the cause for a majority of the loss of source information. The structure of the code in the JModelica.org compiler varies quite a bit depending on which step in the compiler you are looking at. This includes how and where new nodes are created.
As the transformations affect the different parts of the AST differently, which means that the creation of nodes is dependant on the type of node and transformation. This results in having to search through the transformations to find every occurrence of a specific node being created to propagate its source information. The actual code needed to propagate the information is very simple, however.

### 4.3.1 Design Patterns

Some of the transformations in the JModelica.org have patterns which can be used in order to simplify the propagation through that transformation. One of the transformations, called *scalarization*, has a pattern which reduced the amount of small changes needed to propagate the source information. When handling an expression node, it would first call a generic method for expressions. The method would then continue by calling a method to handle the scalarization of the specific expression. This made it easy to copy the source information of expressions through the scalarization by modifying the generic method. The flattening step has a structure which is similar to the scalarization, but lacks this generic method when calling expressions. It is worth noting that expressions are a large part of the nodes which are useful to know the source of.

There are `createNode` methods which creates a copy of the node which calls the method. If an `Add` node calls its `createNode` method, it will create an empty `Add` node and then copies the left and right child from the old node to the new node. We could easily add a statement to copy the source to the new node. However, these methods seems to only be used in some of the transformations.
4. IMPLEMENTATION
Chapter 5
Evaluation

It is important to make sure that performance is not significantly impacted. We have used two models from the Modelica standard library for the evaluation, called R134a1 and Spice3BenchmarkFourBitBinaryAdder. We will for convenience refer to the second model as Spice. These models have been chosen as they are two of the larger models available in the Modelica standard library.

5.1 Completeness

We realized early on in the project that it would not be feasible to propagate all information through the compiler within the scope of a thesis project. It was too time consuming to propagate the information through every transformation. Limited knowledge about the transformations and lack of usable patterns resulted in us having to find every instance where a type of node might lose the source information.

Because of this, the result of this thesis is a proof of concept which can be extended with more source information being propagated and more user friendly presentation.

5.2 Memory Usage

The project started out with a large emphasis on avoiding impacting the memory usage when propagating the source information. This is due to the risk of running out of memory when compiling very large models. We had planned to avoid adding a reference field to the source in every node in the AST, as there are a lot of nodes which does not require it. However, this field was already present in every node in the AST, since it was used for error messages which occurred before the flattening step. This meant that we only had to make sure that nothing else we added used too much memory.
5. Evaluation

Adding support to multiple sources increased the memory usage, as more information needs to be stored. For every extra source a node has, the memory usage increases by the amount of two integers and a string reference. The actual string object is never copied, as this would require a lot more memory.

5.2.1 Measurements

The measurements of the memory usage during compilation after implementing the source propagation can be seen in Figure 5.1 and 5.2. Three measurements were done with the compiler before implementing source propagation and ten measurements were done after implementing source propagation. The measurements were done on a laptop running Windows 7 with a 2.6 GHz processor and 8 GB RAM. The memory is measured using a tool for the JModelica.org compiler which prints the time and memory usage at every transformation step in the compiler.

The total memory usage is shown as an area in the figures, varying on which transformation step is being executed. The memory usage of the models before adding source propagation is omitted as it looks identical to the memory usage of the model after source propagation. In the area showing the memory usage, we can see that the memory usage has a spike around transformation step 2 and 6. This spike consists of both the creation of the AST and the flattening. The smaller increase at transformation step 4 is where the flattening starts. The dip at step 7 is where the old AST is thrown away, leaving only the flattened AST. An increase in memory can be seen at step 10, due to a transformation called scalarization, which is a transformation which requires more memory than most of the transformations in the compiler. The memory usage is then relatively unchanged until step 41 where unused functions are removed, followed by code generation at step 43.

The second line with circular points in the figures shows the difference between the memory usage before source propagation was implemented compared to after. While the
5.2 Memory Usage

Figure 5.2: Total memory use and memory increase of the Spice model

Table 5.1: Measured max memory usage before and after implementing source propagation, measured in MB

<table>
<thead>
<tr>
<th>Model</th>
<th>Before</th>
<th>After</th>
<th>Diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>R134a1</td>
<td>77.903178 ± 0.000058</td>
<td>77.904308 ± 0.000051</td>
<td>0.001130</td>
</tr>
<tr>
<td>Spice</td>
<td>1185.166784 ± 0.000784</td>
<td>1185.168852 ± 0.000430</td>
<td>0.002069</td>
</tr>
</tbody>
</table>

changes seem to vary a lot more in the R134a1 model, the scale on the Spice model is about 25 times larger. The large increase in Figure 5.1 between step 7 and 9 is just an increase of 0.005% (600 bytes), from of a total of 11.2 MB. The largest increase, in both size and percent, is found is in the Spice model at step 24, where the memory increases by 32.2 KB from a total of 179 MB, which is an increase of 0.181%. The reason for this increase is most likely due to a relatively large amount of multiple sources being added at step 22 through 24, which contains an alias elimination, the variability propagation and a second alias elimination. This increase is still low, when considering that almost no change can be seen at the peak of the memory use at step 2 through 6, where an increase in memory usage could result in running out of memory.

Table 5.1 shows the memory usage of the compiler during its peak with a confidence interval of 95%. As the confidence intervals do not overlap, it shows that there has been an increase in memory usage. We are not certain of what could cause an increase in memory

Table 5.2: Measured max memory increase before and after implementing source propagation, measured in MB

<table>
<thead>
<tr>
<th>Model</th>
<th>Before</th>
<th>After</th>
<th>Diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>R134a1</td>
<td>49.593210 ± 0.001030</td>
<td>49.595816 ± 0.000117</td>
<td>0.002605</td>
</tr>
<tr>
<td>Spice</td>
<td>178.698973 ± 0.000577</td>
<td>178.997200 ± 0.000482</td>
<td>0.198227</td>
</tr>
</tbody>
</table>
usage during the flattening, as no new objects are created. The small difference of 1-2 KB could possibly be caused by minor changes in the code causing the Java Virtual Machine to allocate memory differently.

Table 5.2 shows the memory usage during the peak difference, i.e., where the largest increase in memory was found. For both of the models, this was during variability propagation. As the confidence intervals does not overlap, an increase in memory usage has occurred here as well. The increase in difference between Table 5.1 and 5.2 is much larger in the Spice model. This is because the Spice model uses a lot more multiple sources. This is still true when taking the size difference between the models into consideration. The number of calls to the `addLocation` method, which is only used for multiple sources, is 78 in R134a1 and 5163 in Spice. Although the solution is not complete, propagating more information through the compiler should not increase the memory usage, with the exception of a small increase due to needing additional multiple source. We measured that 3592 of the 5163 uses of multiple sources in Spice are currently being propagated. It is therefore unlikely that the memory usage due to multiple sources would increase to problematic levels when all the source information is propagated.

### 5.2.2 Future Improvements

Since there already was a reference field to the source in every node, we tried to look into if we should instead remove the reference field from nodes which does not need it. We measured the difference in memory usage if we removed the field entirely, as this was an easy way to get a grasp on how much memory could be saved. Some of the measurements can be seen in Table 5.3. The results show that up to 3.2% memory could be saved on the Spice model, by removing the field. While it is a decent amount, it is too small of a change when considering that removing all reference fields to the source is not possible. We cannot have a connection from the compiled model to the original model without somewhere to store the information.

The current implementation of multiple sources stores the line and column of one of the sources twice for every node using multiple sources. They are stored once in the node itself and once in the `SourceLocation` array. This is due to the fact that the integers which stores the line and column of the source originates from JastAdd, making it harder to modify. This would be a much larger problem if the `SourceLocation` array was used on every node, rather than on only the nodes which require multiple sources. A solution could be to store the `SourceLocations` in an `Object` array instead, with the first object being a string instead of a `SourceLocation`.

#### Table 5.3: Measured max memory use and memory saved without the source reference field.

<table>
<thead>
<tr>
<th>Model</th>
<th>Memory Difference (MB)</th>
<th>Memory Saved (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>R134a1</td>
<td>2.073664</td>
<td>2.662</td>
</tr>
<tr>
<td>Spice</td>
<td>37.895792</td>
<td>3.198</td>
</tr>
</tbody>
</table>
5.3 Execution Time

Measuring execution time had a lower priority compared to the memory, as propagating the source information is done by just copying the information when needed. Handling multiple sources requires some looping through arrays, which could in theory take a lot of time. However, multiple sources are not used often enough for this to be a problem, with the largest model used for testing only calling the `addLocation` method 5000 times and the arrays being looped through rarely being larger than 5 objects.

While more extensive measurements are needed to draw any conclusions, the `Spice` model was measured to execute slightly slower on average, from 84,491 ms to 86,210 ms. The execution time before implementing the source propagation varies between 80,834 ms and 88,937 ms, which makes the data unreliable.
Chapter 6

Conclusion

We have propagated source information through the JModelica.org Modelica compiler. It was possible without major restructuring of the compiler, though there was not enough time to make a complete solution. However, the solution is still usable in its current state. The solution has support for multiple sources without heavily impacting the memory usage. The memory usage for the entire solution is also kept low, with the highest increase measured being only 0.18%.

We implemented a presentation of the source information in the form of an HTML file containing the compiled Modelica model, with the source information easily viewable.

6.1 Future Work

Since there are some nodes which are not propagated currently, the solution is not complete. While propagating the information the same way as we did is possible, it might be possible to do a restructure of the compiler, writing the code using programming patterns which allow additional information to be more easily added to the nodes in the compiler, without the information being lost during the transformations.

Other ways to present the information could be made, as the current HTML file was made to be very simple. More advanced ways to present the information could include for example a way to automatically show the code of a source instead of listing the name of the file.
Bibliography


Appendices
Felsökningsverktyg till modelleringsspråk för fysikaliska system

Att simulera en modell av ett system är ett billigare sätt att testa systemet än att göra en prototyp. För att bygga en modell behövs verktyg, både för skapa modellen och för att felsöka om något går fel.

Modelleringsspråket Modelica används för att modellera och simulera fysikaliska system, så som elektriska eller hydraliska system. Att skapa en modell i Modelica innebär, precis som i programering, att allt inte nödvändigtvis fungerar som det ska vid första försöket. Detta kan vara allt från felaktiga resultat till att simuleringen krashar. För att simulera en Modelica-modell behöver den kompileras, vilket betyder att man kan behöva något sätt att hitta vilken del av orginalmodellen som orsakar problemet.


För att lösa detta, propagerar vi informationen om orginalmodellen genom kompileringen och visar denna information sedan i en HTML-fil som genereras. Filen som genereras innehåller den genererade modellen tillsammans med informationen om varifrån de olika delarna av modellen kom. Informationen om orginalmodellen visas genom att hålla över en del av modellen med muspekaren. Ett exempel på detta kan ses i figur 1.