MASTER'S THESIS | LUND UNIVERSITY 2016

Symbolic Simplification Framework in a Modelica Compiler

Johan Calvén, Zimon Kuhs

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2016-25

Symbolic Simplification Framework in a Modelica Compiler

at Lund University, Faculty of Engineering

Calvén, Johan
tna05jca@student.lu.se

Kuhs, Zimon
eng08zku@student.lu.se

20th June 2016

Abstract

The aim of the study is to develop a framework for symbolic simplification algorithms in the JModelica.org compiler. It should make the procedure of adding new algorithms to the compiler easier. Instead of the fixed order currently used in the compiler, algorithms will perform their simplifications to the model equations iteratively. This opens up the possibility for more simplifications being made, when all algorithms have access to the constantly updated, reduced and simplified model equations. We have implemented such a framework in the JModelica.org compiler for the algorithms *Alias Elimination* and *Variability Propagation*. Also, we have designed a canonical form for equations to alleviate the implementation of new algorithms to the framework. The framework improves simulation time at the cost of increased compile time, whereas the effects of the canonical form remain to be evaluated.

Keywords: Algorithm, compiler, expression, framework, JModelica.org, Modelica, symbolic simplification, systems of equations, canonical form, normal form.

Contributions

This thesis was performed by Johan Calvén and Zimon Kuhs and most of the work was done in full collaboration. However, there are a few deviations to this, which are listed below. While the listed items do *not* denote that *all* of the work was performed by the student in question, they do indicate that the majority of the work was performed by the student.

- Johan Calvén
  - Most of the work concerning the two first canonicalization steps; moving to one side (section 4.1), and removal of division (section 4.2).
  - The rudimentary implementation of both the algorithm loops and the worklist (section 3.1).
  - The simplified version of the Variability Propagation algorithm (section 3.2).
  - Setting up tests and performing them (section 5.1). Data aggregation was managed equally.
- Zimon Kuhs
  - The work on the standardized AST (section 4.3).
  - The re-ordering step in equation canonicalization (section 4.4).
  - Implementation of id-use and variable updating.
  - The simplified version of the Alias Elimination algorithm (section 3.2).
  - Presentation of resulting test data (section 5.1). Data aggregation was managed equally.

Acknowledgements

We wish to extend our gratitude towards our supervisor at LTH, Niklas Fors, as well as our supervisors at Modelon AB, Jon Sten and Jonathan Kämpe, and also our examiner, Görel Hedin. We are thankful for the experience this thesis has been.

Contents

1 Introduction

The purpose of prototyping is to be able to test an early model of a, often complex, physical system in a realistic setting before utilizing it in a larger scale. Even so, the construction of even a handful of prototypes could be a costly endeavour. Creating a digital prototype and performing simulations on it can be a preferable approach, since doing so will likely lower the production cost and time spent [15]. *Modelica* [19] is a modelling language capable of describing complex systems on equation form. *JModelica.org* [18] is an open-source platform for Modelica, maintained by *Modelon AB*, for simulation and optimization of such complex physical systems.

In this thesis we have investigated two extensions to the JModelica.org compiler; an algorithm framework and translation of equations to canonical form. The purpose of these extensions are to increase the amount of symbolic simplifications made to produced code. This is done in order to reduce simulation time, while keeping compilation time costs comparatively low.

1.1 Problem description

In JModelica.org several *symbolic simplification algorithms* are already in use, however they are not utilized fully. Their goal is to reduce and simplify the model equations as much as possible. Since algorithms are performed in a fixed order in the compiler, one algorithm might introduce changes which an earlier algorithm would have benefited from.

The number of algorithms is likely to increase with time as the developers improve the compiler. Consequently JModelica.org would benefit from extension with an algorithm framework, since this would facilitate the work of adding new algorithms to it.

Canonical forms are a concern in the field of mathematics, but are also equally relevant within the realm of computer science [1]. A canonical form for equations minimizes the amount of possible representations of equations in a program, possibly reducing algorithm complexity and implementation difficulty [11]. As such, it is important to investigate to which extent the application of a canonical form is possible in the JModelica.org compiler, so that the potential benefits are used in further work. Introducing a *canonical form* to the equations, would mean that management of them in JModelica.org would be easier. The canonical form would translate systems of equations to a predictable form that could facilitate algorithm implementation, such as e.g. *Common Subexpression Elimination* (CSE).

1.2 Related work

As a basis for this thesis lies previous theses performed at Modelon AB.

1.2.1 Symbolic simplification algorithms

The JModelica.org compiler contains several algorithms for symbolic simplification, but this thesis focuses primarily on two of them. The first one is called *Alias Elimination* (explained in Chapter 2.3.1) and is similar to the compiler optimization algorithm *Copy Propagation* [5]. The second one is called *Variability Propagation* (explained in Chapter 2.3.2) and is a combination of the two compiler optimization algorithms *Constant Propagation* [6] and *Constant Folding* [7].

1.2.2 Symbolic simplification algorithm framework

Previous research at Modelon indicated that using algorithms in a fixed order with a predefined number of iterations is sufficient for some programs, but might not be for all of them [11]. To solve this problem, the algorithms should work together rather than separately and continue working until reaching a fix point. In order for such an implementation to be easily extensible and modular, a framework for algorithms is needed. Even for only two algorithms, re-implementing them so that they alternate on a program should improve performance [14]. This would also, preferably, be possible without requiring detailed knowledge of the other implemented algorithms when implementing a new algorithm, by keeping the necessary logic at the framework level.

1.2.3 Canonicalization of systems of equations

Previous research at Modelon has also suggested that it would be possible to improve the performance of the compiler's algorithms with the application of a canonical form for equations [10], [11]. A canonical form is sometimes referred to as a *normal form*, i.e. the possible representation of equivalent equations would be narrowed down (this is explained more thoroughly in Section 2.4). It would make it easier to implement new algorithms for symbolic simplification, since the number of possible expressions in equations would be decreased.

1.3 Results

A rudimentary framework for algorithms was implemented and was shown to decrease simulation time (Section 5.1.2) at the cost of increased compile time (Section 5.1.1) for Alias Elimination and Variability Propagation. As the complexity of combining the two algorithms was higher than expected, there was not enough time to investigate how difficult it is to make further algorithm additions to the framework, and the difficulty of integrating the two suggests that additions of other algorithms could be difficult. However, as a base for the framework is implemented, it should be possible to extend and modify it so that it is more accessible to developers.

Due to time constraints and prioritization of the framework, the implemented algorithm that transforms equations to canonical form was never evaluated. We can see that it *can* transform equations to the specified form, but the effects of it

on performance are still unknown, as the algorithm does not function properly within the framework.

2 Background

In this chapter we will describe some of the concepts required to understand the extensions to the JModelica.org compiler performed in this theses. A brief introduction to Modelica will be given, as well as an illustration of how JModelica.org and its compiler operates. The algorithms we are trying to improve are explained, followed by the concepts of the canonical form. Finally, the compiler construction framework JastAdd [4] is described briefly.

2.1 Modelica

Modelica is a declarative equation-based language for modelling and simulation of different types of systems, e.g. automated, electronic, mechanical or thermal. It uses features similar to ones used in *object-oriented programming*, e.g. using *classes* to represent systems or sub-systems. Models are represented with *systems of equations* (see Figure 1), rather than with assignments.

$$\begin{cases} x = 2y - z \\ y = 2z \\ z = 1 \end{cases}$$

Figure 1: Variables in Modelica are described in relation to each other.

2.1.1 Models

A model is a defined structure much like a class in e.g. `Java` programs. It can contain variables, equations, references to other models (components, see Section 2.1.2 below), functions, etc. Its purpose is to describe a physical system.

2.1.2 Language

Figure 2 shows an example Modelica model, which describes a car's movement across a plane when a force is applied to it. The results of the model simulation are seen in Figure 3.

To give a brief introduction to the Modelica language, some of its key parts are described next:

Components A *component* is an instance of a model, meaning that in creating a model we can re-use several instances as components or sub-models of

```
1  model CarMovement
2    parameter Real mass  = 1500  "Weight (kg)";
3    parameter Real force = 2000  "Applied force (N)";
4    Real pos  "Car's position (m)";
5    Real velocity  "The car's velocity (m/s)";
6  equation
7    velocity = der(pos);
8    force    = mass * der(velocity);
9  end CarMovement;
```

Figure 2: Example of a Modelica model. The `der(x)` operator denotes the time *derivative* of the argument `x`.

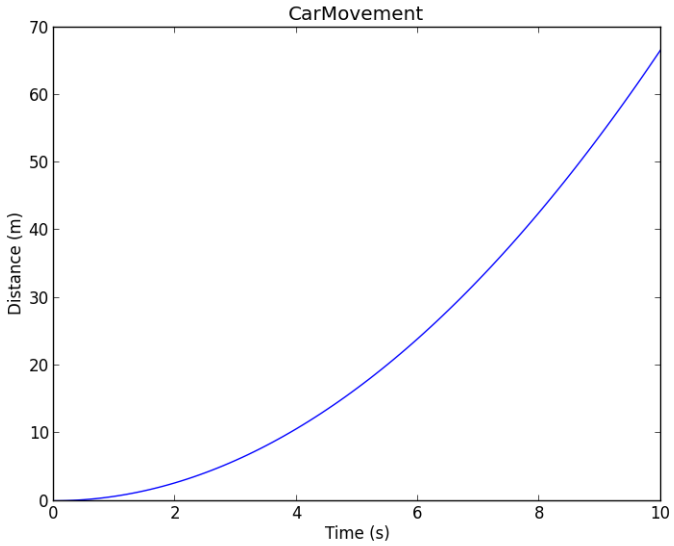


Figure 3: Simulation results for the `CarMovement` model.

other models. This is similar to how classes in Java can have references to instances of other classes.

Equations Unlike most programming languages, the equals sign in Modelica describes equality rather than assignment. This means that the left-hand side and the right-hand side of an equation *are equal*, which must hold true throughout the entire simulation. This is called using *declarative* programming rather than *imperative*, since we only describe the *relation* between variables rather than define how they are calculated.

Functions While equations are the focus for Modelica, true imperative programming is sometimes more practical or even necessary. A *function* allows the programmer to write code imperatively, with assignments (Modelica uses the := operator) and statements. An example is given in Figure 4.

```
 1  function maxValue  "Return maximum of two values"
 2    input  Real value1;
 3    input  Real value2;
 4    output Real result;
 5  algorithm
 6    if value1 > value2 then
 7      result := value1;
 8    else
 9      result := value2;
10  end maxValue;
```

Figure 4: An example of a Modelica function.

Simulation Simulation of a model means collecting data about how the model responds for different sets of input over a period of time. When simulating a Modelica model, a developer can see the behaviour of the model from generated simulation data (see Figure 3). This is done in order to get an idea about how the system would behave in real situations, which is the central purpose of Modelica.

2.1.3 Modelica Standard Library

The Modelica Standard Library (MSL) [17] was created by the *Modelica Association* [19], in order to avoid re-creating components or models that many developers would like to use. It contains model components and standard component interfaces, which are used in models for simulation in a wide variety of domains. E.g. something as simple as a capacitor or a valve might be part of just about any modelled system. Naturally, this also applies to more complex constructs, such as low- or high-pass filters.

2.2 JModelica.org

JModelica.org is a platform for simulation of intricate dynamic systems and is based on Modelica. The platform is written in `C, C++, Java`, and `Python`, and is open-source.

2.2.1 Compiler steps

The compiler will go through a couple of steps in order to transform `Modelica` code into runnable code. During this process, three different abstract syntax trees (AST) will be created.

The files are parsed and checked if syntactically correct. The parser creates an AST called *source* AST which represents the parsed program. The source AST is then unfolded when the chosen model is instantiated. All of its components are also populated with the content that belongs to the component's type in a transitive manner. The resulting AST that is created is called the *instance* AST. This process is mostly run by the semantic error checking. Most of the error checking and analysis is performed on the instance AST.

The last AST created is called the *flat* AST. Since a model can utilize other models via connections, these other models have their own sets of equations. *Flattening* means that the source AST is transformed to a form where the entire program consists of one system of equations rather than several. It is analogous to moving all logic behind an entire `Java` program, all classes, methods and data, into a singular class where all logic is contained. This is done in order to improve execution time through removal of call overhead [12]. On the flat AST different transformations are made to be able to simulate the system. The transformed flat AST is used to generate `C` code.

2.2.2 Algorithms for simplifying systems of equations

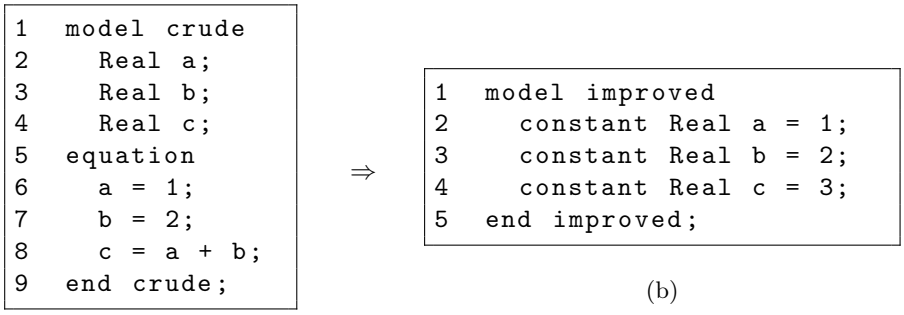
The JModelica.org compiler features a collection of algorithms which analyse and improve the flat AST, so that the simulation and remaining compilation steps will be faster. These include e.g. Alias Elimination (explained in Section 2.3.1) and Variability Propagation (explained in Section 2.3.2).

Simplification of systems of equations
Using the flat AST it is time to perform simplification on the equation system. This means analysing the structure of individual equations with the purpose of finding and eliminating redundancy, as well as finding areas of improvement for the equations. Improvements that will decrease simulation time of the model while still retaining its behaviour.

Consider Figure 5a where we know at compile time, that the variables a and b are literal integers. This means that if we replace the usages of a and b in the equation on line 8 with the literals, we can immediately evaluate the value of c at compile time. This yields the model in Figure 5b, which requires less memory and fewer clock cycles to process.

2.3 Symbolic simplification algorithms

Following sections gives a brief explanation of how Alias Elimination and Variability Propagation works.



(a) (b)

Figure 5: Result of simplification algorithms applied on a Modelica model.

2.3.1 Alias Elimination

For an equation on the form $a = b$, we have two variables that are equivalent. Alias Elimination (AE), similar to Copy Propagation [5], processes all equations in order to find such expressions. It then replaces all usages of such variables with one of them.

Consider the system of equations in Figure 6a, where there is an easily identifiable alias. If AE chooses a over b, the system will instead have the form given in system of equations in Figure 6b. The choice of variable does not matter for the correctness of the modified model. It can be chosen according some heuristic, such as the greatest number of occurrences in the model.

$$\begin{cases} a &= b \\ c &= d \\ x &= b + c - d \\ y &= a + 2 * b - c + e \\ z &= e + x \\ a &= 1 \\ b &= 3 \end{cases} \Rightarrow \begin{cases} x &= a + c - c \\ y &= a + 2 * a - c + e \\ z &= e + x \\ a &= 1 \\ b &= 3 \end{cases}$$

(a) (b)

Figure 6: AE replaces the variables b and d with the variables a and c, respectively.

Note that it is possible in the system of equations in Figure 6b that further simplifications can be made by other algorithms, as we will see in Section 2.3.2.

2.3.2 Variability Propagation

The variability of an equation can sometimes be lowered during compile time. There are four levels of variability [2], which, ordered from top to bottom, are *continues-time*, *discrete-time*, *parameter* and *constant*. The constant variability means that the value of the variable in an equation is always fixed. For

an equation on the form $a = 1$, the variable a can only take the value of 1. Variability Propagation (VP), a combination of Constant Propagation [6] and Constant Folding [7], finds such equations, changes the variable into a constant, removes the constant equation and then propagates all usages of the constant to its numerical value in other equations. Constant values in equations are also computed during compile time to simplify the equations even further. An equation on the form $x = 1 + 2 + 3 + a$ will then be simplified to $x = 6 + a$.

Continuing with the system of equations example in Figure 6b, VP will first find $a = 1$ and $c = 3$. This will lead to the system of equations in Figure 7b, which is further reduced to the system of equations in Figure 7c. Here, AE would have been able to continue simplifying the alias equation $y = e$ if being performed again. It is possible to run algorithms several times, however, for large systems this might be costly in terms of compile time.

$$\boxed{\begin{cases} x &= a + c - c \\ y &= a + 2 * a - c + e \\ z &= e + x \\ a &= 1 \\ c &= 3 \end{cases}} \Rightarrow \boxed{\begin{cases} x &= 1 + 3 - 3 \\ y &= 1 + 2 * 1 - 3 + e \\ z &= e + x \end{cases}} \Rightarrow \boxed{\begin{cases} y &= e \\ z &= e + 1 \end{cases}}$$

(a) (b) (c)

Figure 7: VP finds and lowers the two variables a and c to constants and their values are propagated in (b). In (c), the same has happened to x and the remaining equations have been folded.

2.4 Canonical form

An equation can be represented in many different ways, e.g. Equation (1) and Equation (2).

$$\boxed{x = a * b * c + a * b * d} \tag{1}$$

$$\boxed{x = (b * (c + d) * a)} \tag{2}$$

The equations are *semantically* equivalent [3]. This means that for every combination of values for a, b, c and x in one equation the other equation is also satisfied. They are, however, *lexically* different. The purpose of maintaining a canonical form (also referred to as a *normal form*) for equations is to decrease the number of possible representations for them. This means that to *canonize* an equation it would have to be transformed towards a normal form, and that a semantically equivalent equation undergoing the same transformation would come close to the same representation (ideally the same).

In our canonical form we move all operands to one side, remove divisions, expand multiplications[1], reorder variables according to a set of rules, and trans-

[1]The aim is to remove the need for parentheses. E.g. $(a + b)(c + d) \rightarrow ac + ad + bc + bd$

late negations and subtractions to multiplications with the literal -1. Both the example equations would then have the form illustrated in Equation (3).

$$\boxed{0 = a * b * c + a * b * d + x * (-1)} \tag{3}$$

When two equations are lexically equivalent they are easy to compare. It would be difficult for a program to identify the two original equations as equivalent, but with both equations on the same form it is only required to iterate through all the characters in the respective strings[2].

2.5 JastAdd

The JModelica.org compiler has been developed in the compiler construction framework *JastAdd*. JastAdd is Java based and uses object-orientation, static aspects and declarative computations to make the compiler easily extensible [4]. By adding new modules, called aspects, it is possible to extend the behaviour of already existing classes without editing the classes directly. JastAdd also uses *attributes*, data attached to AST nodes described in relation to methods or other attributes.

When JastAdd generates Java code all behaviour related directly to a class is weaved together into the same class. This means that attributes and methods that belong in a class can be specified outside of it, where it may be more contextually relevant.

2.5.1 Abstract grammar

The structure of an abstract syntax tree is defined by an *abstract grammar*. E.g, Equation (4) and Equation (5) using infix and prefix notation respectively, are semantically equivalent but lexically different. Abstract grammar is a viewpoint which concerns itself with what parts of a language expression are actually semantically significant, in this case, the fact that we have a summation of two operands.

$$\boxed{x := 1 + 2} \tag{4}$$

$$\boxed{x := + \ 1 \ 2} \tag{5}$$

2.5.2 Aspects and attributes

As was mentioned in Section 2.5 it is possible to define behaviour for a class outside of its class using aspects, which are later weaved together when generating the Java code. See for example Figure 8, where the aspects can be declared in separate files, but the resulting Java code will assemble the methods `a()` and `b()` in the same class, `A`.

[2] An added benefit is that strings of non-equal length are immediately identified as non-equivalent.

```
1  aspect A {
2    public void A.a() { ... }
3  }
4
5  aspect B {
6    public int A.b() { ... }
7  }
```

Figure 8: Example of two JastAdd attributes.

Weaving assembles all attributes and methods declared for `A` into the class declaration of `A`. This means that extending a node type with functionality requires only the addition of a new aspect, and disabling of specific functionality requires only the removal of pertinent aspects.

3 An algorithm framework

This chapter describes the symbolic transformation framework and its basic functionality. The two algorithms Alias Elimination and Variability Propagation have been implemented in the framework.

3.1 Symbolic transformation framework

The transformation framework processes one equation at a time from the system of equations, using the implemented algorithms. In order to keep track of what equations have yet to be processed, a worklist (instance of a class `WorkList`) of equations is maintained. Should an algorithm find that it can modify the system using the information in the currently processed equation, it will update the worklist with all equations it has modified.

In addition to this, a list of symbolic simplification algorithms is maintained, and the algorithms therein are run one by one on the current equation, referred to as the *working equation* (see Figure 9). The loop structure in Figure 9 is explained below in Section 3.1.2.

```
 1  List worklist <- getAllEquations()
 2  List listOfAlgorithms <- addAlgorithms()
 3
 4  preConfiguration(listOfAlgorithms)
 5
 6  WHILE worklist has equations
 7    workingEquation <- worklist.poll()
 8    FOR each algorithm in listOfAlgorithms
 9      IF algorithm.run(workingEquation) THEN
10        break
11      ENDIF
12    ENDFOR
13  ENDWHILE
14
15  postConfiguration(listOfAlgorithms)
```

Figure 9: Pseudo code of how the framework runs each algorithm in turn. The algorithms update the worklist with the equations they modify. The methods preConfiguration and postConfiguration handles the algorithms that needs to perform work before and after the loops.

The loop continues until the worklist is empty. This means that as long

as modifications are performed, at least one repetition of all algorithms in the worklist will be performed.

This equation-oriented procedure has an advantage over having each algorithm work on the entire system of equations each time. If the latter procedure was used, each algorithm would need to process all equation each time it executed, resulting in unnecessary operations. With the equation-oriented procedure, an equation will only be revisited if it is modified.

3.1.1 Pre-requisite for loop termination

For the algorithm loop to terminate, it is required for each algorithm used in the framework to always perform some form of reduction of the system of equations. As the loop in itself always reduces the worklist unless a change is performed, the requirement can be further narrowed to the following: for the framework to terminate, any changes performed by the algorithms used in it must be reductions of the system in some form.

3.1.2 Adding an algorithm to the framework

To add an algorithm to the framework, the algorithm needs to extend an abstract class `SymbolicTransformationAlgorithm` that requires implementation of two methods. The first method is `isUsed()` that should check if the algorithm is used by the compiler[3] and if so be added to the algorithm list in the framework. The second method is `run()` that takes an equation as argument and runs the algorithm process. The run method returns a boolean value, true if the algorithm succeeds in making any changes and false otherwise. The reason for this is that if the algorithm succeeds, it might remove the current working equation (this is the case for both AE and VP). If the run method returns true, the framework will begin anew and poll the next equation in the worklist. The algorithm list will then start over with the first algorithm in the list. If `run()` returns false, the next algorithm will process the current working equation. This main loop will terminate when all algorithms have processed the working equation and there are no additional equations to poll from the work list.

3.1.3 The `preRun()` and `postRun()` methods

While not necessary for the algorithm to function correctly inside the framework, it *can*, depending on its character, need to use the `preRun()` and `postRun()` methods. The method `PreRun()` perform modifications *before* the framework's loops (the nested loops in Figure 9, line 6), and an algorithm can thus process the set of equations without interference from the other algorithms if needed. Analogously, `postRun()` will process the set of equations *after* the loops. This is illustrated in Figure 9, where the method `preConfiguration()` will call the preRun method of each algorithm and the method `preConfiguration()` will call the postRun method of each algorithm. The order in which algorithms process equations in the loops is maintained for the pre- and postrun methods; e.g. if AE processes an equation γ first, and VP then processes γ, AE's

[3]The compiler uses settings flags for using different features in it. Its algorithm do as well, and the framework refers to these flags to see if it should use the algorithms.

`preRun()` method is run before VP's, and the same order applies for `postRun()`. These methods are not required to be implemented by algorithms extending `SymbolicTransformationAlgorithm`.

A possible addition to the framework would be to enable re-ordering of the algorithms' `preRun()` and `postRun()` methods, but for the sake of simplifying the rest of the framework implementation, this was never attempted.

3.2 Simplified algorithms

As a first step, two simpler versions of the two algorithms AE and VP were implemented. This was done in order to explore the framework solution without involving the full complexity of the two existing algorithms.

The simpler AE algorithm was capable of eliminating *direct* alias equations (as opposed to being able to manage e.g. negative alias equations) and consequently update the remaining equations to reflect this. The simpler VP algorithm was capable of propagating constants from constant-expression equations (e.g. $x = 1$), replacing variable uses in other equations. It could also identify parameters and fold constants.

3.3 Modified algorithms

From the experience of the simpler algorithms, we moved on to implement the versions of AE and VP used in the JModelica.org compiler. The functionality of AE is explained in Section 2.3.1 and VP in Section 2.3.2. Both algorithms were modified to work on a single equation, compared to originally having access to the set of all equations. One of the major changes to the algorithms, since they no longer had exclusive access to the set of all equations, was how the modified equations were updated. Before, the algorithms could perform a rewrite of the flat AST at the end of execution, when being done simplifying the equations. Rewrites are costly in compile time and inside the framework, the algorithms need to access constantly updated equations. Instead, the flat AST is mutated at the nodes where changes have taken place. Rewrites will only take place when the iteration of the worklist has ended and there are no more equations to process. Extending AE and VP with the abstract class `SymbolicTransformationAlgorithm` meant to work out which parts should be in either the method `run()` or the method `postRun()`.

4 A canonical form

A canonical form for equations denotes a normalized way of expressing an equation. As described in Section 2.4, it is possible for two semantically equivalent equations to have differing lexical representations. A transformation to canonical form *reduces* the amount of possible lexical representations. According to *Richard's theorem* [8] it is impossible to construct a canonical form with which two semantically equivalent expressions are also lexically equivalent.

Deciding on a canonical form for equations was a continuous process. Beginning with one rule, moving all operands and operations to one side, new rules were established from discussion with the supervisors after assessing the efficiency of the previous step. This procedure was chosen since the canonical form was meant to be tailored to the needs of JModelica.org rather than considering using a present one, such as the one used in `ModSimPack` [16].

4.1 Removal of left-hand side

Each equation, being on the form $f(x) = g(x)$, is transformed by subtraction with $f(x)$ to $0 = g(x) - f(x)$. This means that we do not need to gather information about operations and operands from both the left-hand side and right-hand side, resulting in fewer collections to manage in the compiler.

4.2 Division removal

In order to simplify equations, all division expressions are translated into multiplication by multiplying all operands with their greatest common divisor. This means that e.g. $\frac{a*b}{c} + \frac{c*a}{d*c} - \frac{d}{a+b}$ would be multiplied by $c*d*(a+b)$, translating it into $a*b*d*(a+b) + c*a*(a+b) - d*c*d$. This removes one operation, division, entirely from equations. As such, there are fewer types of nodes to consider in the AST.

4.3 A standardized AST

In order for a model to be in predictable form, the AST representing the program should use an as simple structure as possible. The chosen structure for the AST was to use only *additions* and *multiplications*. A negated expression and a subtraction is represented with a multiplication with the literal -1

In addition, multiplications are "expanded" as thoroughly as possible, e.g., $(a+b)(c+d) \rightarrow ac+ad+bc+bd$. This means that the AST will be transformed to have additions above the multiplications in the tree, creating what we refer to as an *add-mul tree* (see Figure 10).

Equation (6) shows a non-canonicalized equation and in Equation (7) the multiplications have been expanded and the resulting subtractions have been translated to multiplications with -1.

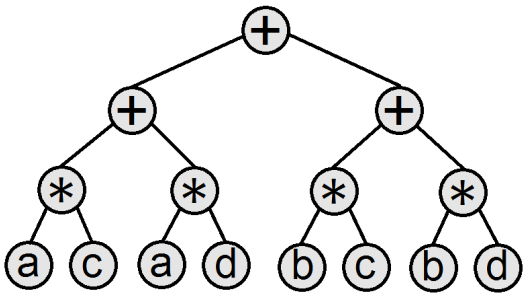


Figure 10: The canonical form transforms the AST to keep additions at the upper nodes and multiplications at the lower. All subtractions and divisions are removed.

$$\boxed{x = a - b \cdot (1 - c)} \tag{6}$$

$$\boxed{x = a + bc + b \cdot (-1)} \tag{7}$$

4.4 Ordering

In order for semantically equivalent equations to be able to become lexically equivalent, expressions must be ordered according to some rules. For example $a + b$ can also be written as $b + a$, so both variables must be arranged in a predictable manner. At this algorithm step, only additions, multiplications, variables, and literals can occur in an equation, why the respective ordering rules only govern those types of expressions.

Below, individual ordering priority for additions, multiplications, variables and literals is listed.

1. Positive expressions are ordered before negative expressions.
   $(0 = -a + b) \Rightarrow (0 = b + a \cdot (-1))$

2. Multiplication expressions are ordered before variables.
   $(0 = a + bc) \Rightarrow (0 = bc + a)$

3. Variables are ordered before literals.
   $(0 = 1 + a) \Rightarrow (0 = a + 1)$

4. Variables contained within a multiplication are ordered lexically.
   $(0 = cadb) \Rightarrow (0 = abcd)$

5. Multiplication expressions with equal sign are ordered lexically according to their first non-equal variable[4].
   $(0 = abde + abce) \Rightarrow (0 = abce + abde)$

6. Variables outside a multiplication expression are ordered as they would be in a dictionary.
   $(0 = \texttt{varNameMore} + \texttt{varName}) \Rightarrow (0 = \texttt{varName} + \texttt{varNameMore})$

7. Literals are ordered numerically in descending order.
   $(0 = 45 + 123) \Rightarrow (0 = 123 + 45)$

As an example, consider Equation (8) and Equation (9), where the first equation is unordered and the second one is ordered.

```
0 = - ab * aa + c + d - e - f;
```
(8)

```
0 = c + d + aa * ab * (-1) + e * (-1) + f * (-1);
```
(9)

[4]Due to compiler simplifications performed before the framework is executed, equations on the form $C_1 f(\alpha) + C_2 f(\alpha) ... + C_n f(\alpha)$ for constants C_i and a multiplication expression $f(\alpha)$ can not occur at this point. They are added together, producing $(C_1 + C_2 + ... + C_n) f(\alpha)$

5 Evaluation

In this chapter we present a comparison between the compiler's performance when using the framework and not using the framework. We also evaluate the effect of the framework on the resulting models simulation time. Furthermore, we briefly discuss the canonical form.

In order to test the framework for numerous models in a realistic setting a benchmark is required. The MSL (*Modelica Standard Library*, see section 2.1.2) was used for this, due to its variety in both kinds of systems and the scenarios covered by the test models. MSL was used in order to measure both compile time and simulation time, but also to verify correctness.

5.1 Framework results

In order to gauge the effectiveness of the framework, we use several different metrics for it when evaluating the AE and VP algorithms. We gather information about the compilation time for the models, as well as the simulation time when running the resulting code. In addition, we look at how many times the algorithm processes equations; the framework's purpose is to use algorithms until a fix point is reached, which can be costly in terms of compilation time. The corresponding results for the original implementation, i.e. the compiler version when the framework was first created, are consequently compared to these metrics.

The eleven models that were used in the end are listed in Table 1. They are ordered according to average compile time, with the lowest time being listed first. For brevity we refer to the different models by their last name in the remainder of the report, `Math.Nonlinear.Examples.FirstExample` e.g. will be referred to simply as `FirstExample`.

We did not have enough time run all models in MSL (366 models), so we decided to limit the number of models to a sample of 15. The smaller sample consisted of five small, five medium and five large models. Unfortunately two of these did not compile for the original JModelica.org compiler and another two did not compile when adding the framework extension to the compiler. This meant that we ended up with eleven models for comparing the results. In Section 5.2, an explanation to why we had more compile errors with the framework is given.

5.1.1 Compile time

There is not much room for increasing compile time in JModelica.org since it already suffers from relatively slow compile time, as it compiles very large models and performs a lot of optimizations on them. This puts pressure on the

Model Names
Math.Nonlinear.Examples.quadratureLobatto3
StateGraph.Examples.FirstExample
Media.Examples.SolveOneNonlinearEquation.Inverse_sine
Thermal.HeatTransfer.Examples.TwoMasses
Electrical.PowerConverters.Examples.ACDC.RectifierBridge2Pulse.ThyristorBridge2Pulse_RL
Electrical.QuasiStationary.SinglePhase.Examples.ParallelResonance
Electrical.Machines.Examples.DCMachines.DCPM_Start
Magnetic.FluxTubes.Examples.Hysteresis.SinglePhaseTransformerWithHysteresis2
Mechanics.MultiBody.Examples.Loops.EngineV6
Media.Examples.R134a.R134a1
Media.Examples.R134a.R134a2

Table 1: The eleven models used to test the framework.

framework extension to keep the compile time comparably close to the original compiler. Having more simplifications being made to a model should, in theory, increase the time at this step of the compiler. At the same time, later steps in the compiler could be given less work to do, leading to the overall compile time being nearly unaffected or even reduced. We ran 50 compilations of each of the eleven models and calculated the average time. As can be seen in Table 2 the changes in compile time are noticeable. The average time was *increased* by 9.09%. *Difference* denotes the compilation time difference between using the framework and not. Especially for the already time-consuming models (i.e. `EngineV6`, `R134a1`, and `R134a2`) the increase in compile time was significant.

Model	Difference (s)	Percentage Change
quadratureLobatto3	0.09172	+2,79%
FirstExample	0.17354	+5,11%
Inverse_sine	0.06966	+2,04%
TwoMasses	0.0907	+2,6%
ThyristorBridge2Pulse_RL	0.45538	+7,50%
ParallelResonance	0.34404	+5,5%
DCPM_Start	0.25242	+4,02%
SinglePhaseTransformerWithHysteresis2	0.10404	+1,62%
EngineV6	12.01552	+24,9%
R134a1	38.528	+26,24%
R134a2	37.80302	+25,64%

Table 2: Comparison of compile time between using the framework and not.

If we use the data in Table 2 to view percentage as a function of the original compile time we can illustrate this as a graph (refer to Figure 11). Using the original compile time we get a sense of how complex it is for the compiler to compile the model. We see that as the time of compiling a model increases, so does the percentage increase rapidly. This is especially evident for the bigger models, where the percentage is high relative to the smaller ones.

One must, however, consider how statistically certain the increase in compilation time is. Because of this we calculated the confidence interval of the tests

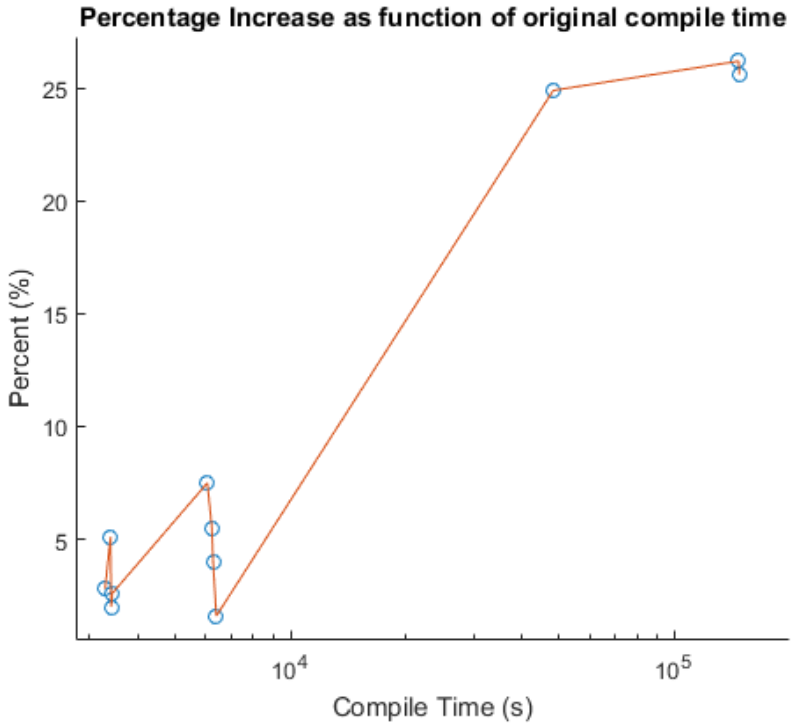


Figure 11: Comparison of compile time between using the framework and not. The x-axis show the compile time using the framework. The y-axis show the percent change in compile time after using the framework compared to no using it.

for each model at a 95% confidence level, as can be seen in Table 3. There are no overlaps between respective confidence intervals, which means that we can be certain with 95% confidence of the increase in compile time.

Model	Without Framework	With Framework	Overlap
quadratureLobatto3	(3273.2, 3300.6)	(3364.7, 3392.6)	No
FirstExample	(3377.1, 3410.8)	(3550.5, 3584.5)	No
Inverse_sine	(3399.3, 3422.9)	(3465.4, 3496.1)	No
TwoMasses	(3416.3, 3446.4)	(3505.1, 3539.0)	No
ThyristorBridge2Pulse_RL	(6035.1, 6102.9)	(6469.9, 6578.8)	No
ParallelResonance	(6195.6, 6273.4)	(6528.1, 6629.0)	No
DCPM_Start	(6251.5, 6313.5)	(6495.4, 6574.4)	No
SinglePhaseTransformerWithHysteresis2	(6368.5, 6440.5)	(6477.8, 6539.3)	No
EngineV6	(48157.0, 48427.8)	(60089.1, 60526.7)	No
R134a1	(146503.9, 147192.1)	(184828.4, 185923.6)	No
R134a2	(147135.3, 147727.7)	(184808.0, 185661.0)	No

Table 3: Confidence intervals for compilation time for not using the framework and using the framework.

5.1.2 Simulation time

Improving the simulation time is of real interest. It would show that the framework increase the efficiency of the algorithms. To investigate this we ran 50 simulations for each of the eleven models used when investigating differences in compile time. The average time was calculated. As can be seen in the comparison of simulation times in Table 4, the difference between using the framework and not is minuscule if anything. However, the framework *decreases* overall simulation time by roughly 1.09%.

Model	Difference (s)	Percentage Change
quadratureLobatto3	-0.0004	-1,5%
FirstExample	-0.0018	-6,3%
Inverse_sine	-0.0005	-2,7%
TwoMasses	0.0001	+0,5%
ThyristorBridge2Pulse_RL	0.0006	+0,2%
ParallelResonance	-0.0004	-1,2%
DCPM_Start	-0.0006	-0,6%
SinglePhaseTransformerWithHysteresis2	-0.1614	-3,6%
EngineV6	-0.6580	-1,0%
R134a1	0.0003	+0,1%
R134a2	0.0473	+4,3%

Table 4: The difference in simulation time using the framework.

If we investigate the percentage change as a function of the original simulation time we get the graph in Figure 12. Judging from this graph, there seems to be little to no correlation between simulation time and any performance effects from the framework.

As with compile time, we must consider the confidence of the generated data. Therefore the confidence interval was calculated for simulation time as well (also at a 95% confidence level), as is illustrated in Table 5. As can be seen there is

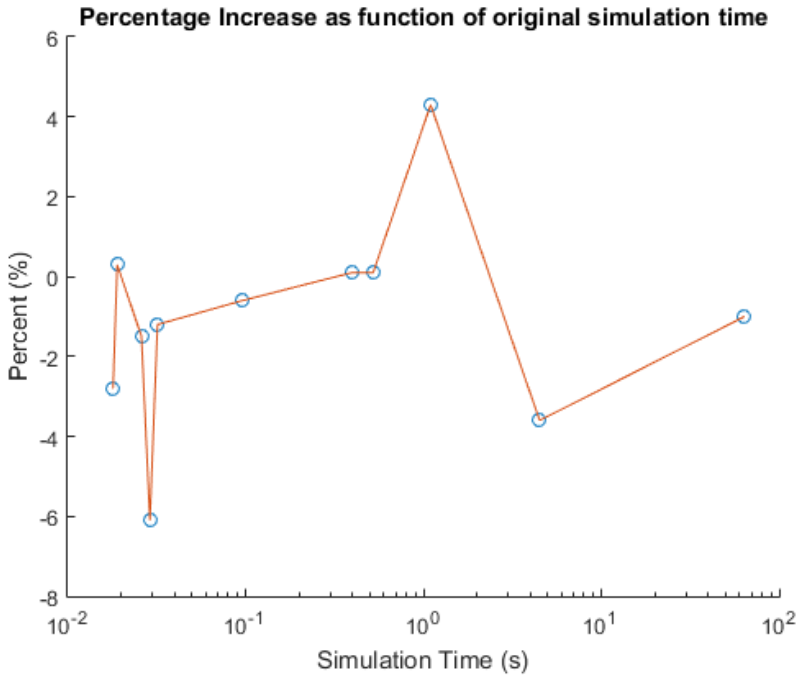


Figure 12: Comparison of simulation time between using the framework and not. The x-axis show the simulation time using the framework. The y-axis show the percent change in simulation time using the framework compared to not using it.

quite a lot of overlap between the respective confidence intervals. This means that even though the simulation time has improved, it is with some uncertainty.

Model	Without Framework	With Framework	Overlap
quadratureLobatto3	(0,018, 0,034)	(0,018, 0,033)	Yes
FirstExample	(0,014, 0,043)	(0,013, 0,041)	Yes
Inverse_sine	(0,006, 0,031)	(0,006, 0,030)	Yes
TwoMasses	(0,011, 0,027)	(0,011, 0,027)	Yes
ThyristorBridge2Pulse_RL	(0,385, 0,408)	(0,385, 0,409)	Yes
ParallelResonance	(0,022, 0,043)	(0,021, 0,043)	Yes
DCPM_Start	(0,087, 0,105)	(0,086, 0,105)	Yes
SinglePhaseTransformerWithHysteresis2	(4,490, 4,518)	(4,321, 4,365)	No
EngineV6	(63,764, 63,955)	(63,100, 63,303)	No
R134a1	(0,513, 0,532)	(0,515, 0,531)	Yes
R134a2	(1,091, 1,108)	(1,139, 1,155)	No

Table 5: Confidence intervals for simulation time for not using the framework and using the framework.

5.1.3 Equation modifications

The number of times an algorithm modifies an equation provides us with a sense for how much added work the framework performs. Without the framework, AE will run twice and on each run it will visit the equations only once. VP will only run once, but it continues executing, in a similar way as the framework, until reaching a fix point where no more simplifications are available. This means that it will at least visit all equations once.

The framework will execute until both algorithms are entirely finished with a model. Even though this still provides a finite amount of modifications, equations with many terms may be possible to simplify numerous times by different algorithms. This is because the algorithms re-insert any equations that are changed by them into the worklist, and the algorithms could find new changes to perform after one another .

For the framework, Figure 13 and Figure 14 displays the number of times an equation was modified for a model. Since counting equation modifications is fast relative to measuring compile time or simulation time, it was performed for all MSL models the framework could run. With a mean of 123.16 modifications for AE and 183.41 for VP, the framework performs on average $124.06 + 221.35 = 345.41$ modifications for a model. This is in contrast to how the original implementation performs; $67.17 + 248.98 = 316.15$. This means that the framework enables on average 9.26% more modifications.

As expected *with* the framework, *more* changes are performed than without it. While the end result may be relatively modest (9.26%), it remains to be seen if compilation time is significantly increased as a result of the additional modifications.

The ratio of success is also of interest, seeing as while the framework will find more modifications to perform, it will also process a great deal more equations. Figure 15 and Figure 16 below list the ratio $\frac{modifications}{visits}$ where *visits* is the number of times an algorithm processed an equation. It is notable but not unexpected that the success rate has been *lowered* by the framework. An integral part of the framework is to re-asses equations after they have been modified.

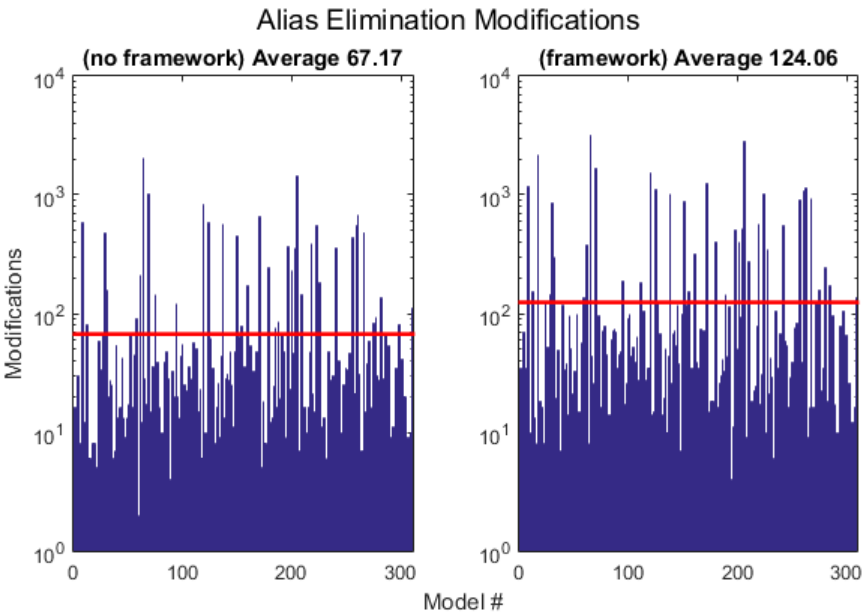


Figure 13: AE modifications for the two implementations.

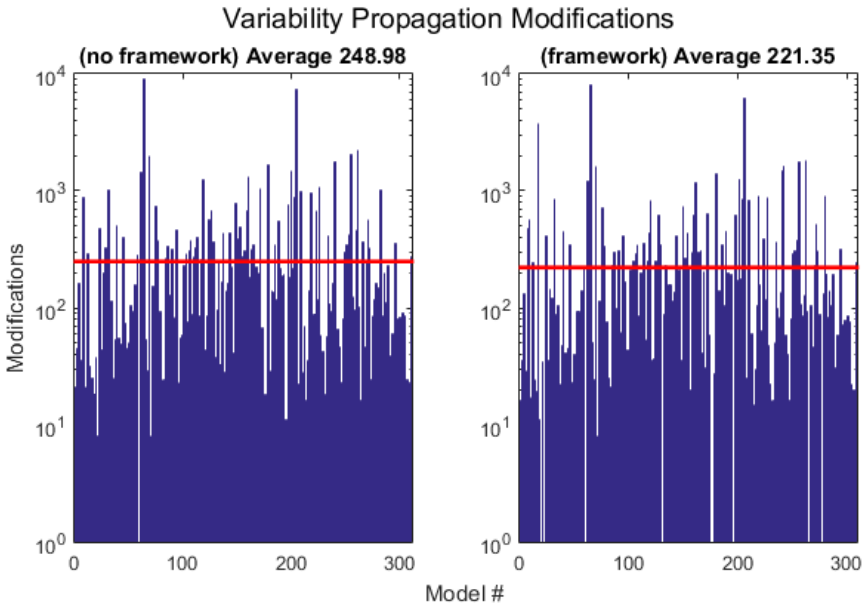


Figure 14: VP modifications for the two implementations.

This is never done in the original implementation.

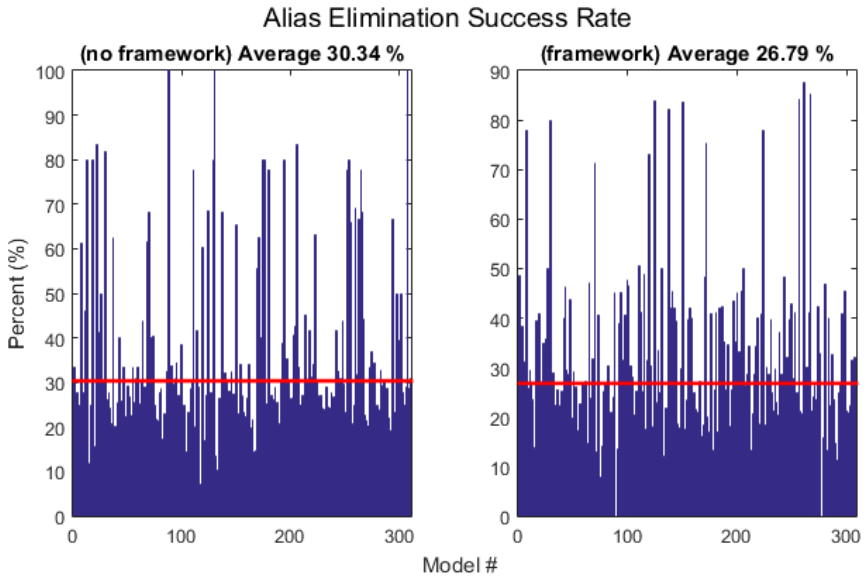


Figure 15: Success ratio of the two implementations for AE.

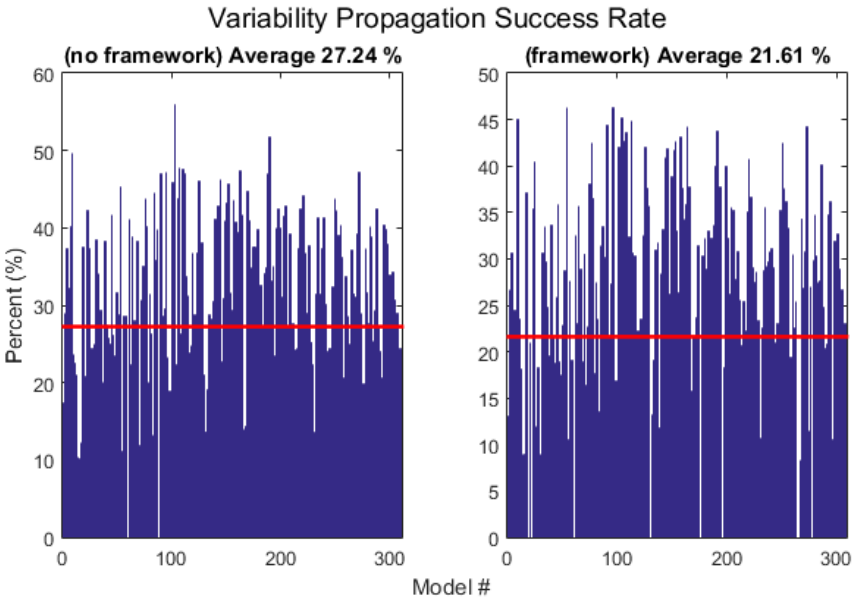


Figure 16: Success ratio of the two implementations for VP.

5.1.4 Difficulty of adding a new algorithm

While it is difficult to predict how complicated the addition of a new algorithm can be (it depends in part on the algorithm in question), the initial framework provides a Java abstract class `SymbolicTransformationAlgorithm` which only requires implementation of four methods, `preRun()`, `postRun`, `isUsed()` and `run()`, see Figure 17. The first two methods specify what needs to be performed in advance and after the algorithm loop respectively. `isUsed()` specifies which compiler option sets the algorithm to be used or not. The `FClass` attribute references the flattened Modelica model on which the algorithms perform their work. The `WorkData` attribute references the collection of equations for the algorithms to process, and data required by the algorithms.

```
public abstract class SymbolicTransformationAlgorithm {
  protected FClass fclass;
  protected WorkData workData;

  public SymbolicTransformationAlgorithm(FClass fclass,
      WorkData workData) {
    this.fclass = fclass;
    this.workData = workData;
  }

  public abstract boolean isUsed();
  public abstract void preRun();
  public abstract void postRun();
  public abstract boolean run(FAbstractEquation eqn);
}
```

Figure 17: The abstract class `SymbolicTransformationAlgorithm` in the framework, which algorithms are to extend.

This means that in order to add another algorithm to the framework, a compiler option entry is needed, as well as pre-loop (`preRun()`) and post-loop (`postRun()`) modifications, and a compiler option entry. The method `run()` describes the algorithm behaviour, which is where most of a developer's work will be focused, even though both the pre-work and post-work could be significant, depending on the algorithm's character. However, this is not the fault of the framework and it can be safely declared that the framework provides an accessible platform for algorithm addition.

However, during the transferral of the AE and VP algorithms there were many complications. Many of them were dependent on the fact that the algorithms presupposed that they were working on each individual equation in the systems of equations undisturbed, rather than only at one equation at a time. The fact that another algorithm may, and most likely will, intervene in between equations means that a bit of logic had to be remodelled to make them fit into the framework properly. This indicates that *modifying an existing algorithm* might be quite difficult. It is of our opinion that if an existing algorithm is to be restructured to fit the framework that the developer at least considers remaking the algorithm entirely.

This again depends on the algorithm in question; the equation canonicalization algorithm (see Section 4) works solely on one equation at a time and it was consequently trivial to remodel it to fit the framework. Algorithm sharing this characteristic would most likely be equally trivial to remodel.

5.2 Especially problematic models

Some models in MSL particularly proved to be difficult to work with, stemming from difficulties in managing updates of derivative (and differentiated) variables in AE. Derivatives are problematic to manage since they are represented as separate implicit variables in the compiler. This also means that derivative references are separate implicit variable references and is treated separately from ordinary variable references. The compiler handles this by letting the variable `Real x` know about its derivative variable `Real der(x)` (and vice versa), but do not share the variable reference (`x` and `der(x)`).

Consider Figure 18, to handle the alias equation `a = b` one of the variables `Real a` and `Real b` will be kept. Since we also have a derivative variable `Real der(a)`, the easiest would be to replace every reference of `b` with `a`. If we had kept the variable `Real b`, we would have had to create a new variable `Real der(b)`, link it to the variable `Real b` and then replace the variable `Real der(a)` with it. Keeping the variable having a derivative was also the approach we opted for.

When both the alias variables have derivatives, as in Figure 19, there is no issue with creating new derivative variables. If keeping the variable `Real a`, the reference of `b` is replaced with the reference of `a` and the reference of `der(b)` is replaced with the reference of `der(a)`.

```
model derivativeVariable1
  Real a;
  Real b;
  Real x;
  Real y;
equation
  a = b;
  der(a) = x;
  b = x + y + 3;
  x = 1;
end derivativeVariable1;
```

Figure 18: An example model with only the alias variable `a` having a derivative variable (`der(a)`). Keeping `a` instead of `b` is much easier to handle. Note that derivative variables are not listed among the normal variables in a model.

5.3 The canonical form

As has been mentioned previously in Section 1.3, the canonical form algorithm was never fully tested due to time constraints. As such, no significant metric data from a collection of models (such as the MSL, Section 2.1.2) was produced,

```
model derivativeVariable2
  Real a;
  Real b;
  Real x;
  Real y;
equation
  a = b;
  der(a) = x;
  der(b) = y;
  b = x + y + 3;
end derivativeVariable2;
```

Figure 19: An example model with both alias variables `a` and `b` having derivative variables (`der(a)` and `der(b)`). Because of this, it does not matter which one to keep.

and it was deemed redundant to produce a few specific test models for the purpose of evaluation. It would not neither be representative of canonicalization's effect on typical Modelica models.

6 Discussion

This chapter starts with describing the benefits and drawbacks of the framework. It then follows with a discussion of future work that could be done for both the framework and the canonical form.

6.1 Benefits and drawbacks of the framework

The addition of the framework has both its benefits and drawbacks. In this section we describe both.

6.1.1 Benefits

As was mentioned in Section 5.1, it is difficult to evaluate the accessibility of the chosen design of the framework. However, using only one abstract class with a few methods only should mean that the overhead complexity of adding a new algorithm is minimal. While the core purpose of the framework is to allow for algorithms to not be run in a fixed order, it is still possible to do so via the `preRun()` and `postRun()` method. If all algorithms do nothing in their `run()` methods and instead execute entirely within `preRun()` or `postRun()`, a traditional fixed order execution is maintained. It is important to consider the fact that some algorithms may not function properly working on a singular equation at a time, why this possibility is important to uphold.

6.1.2 Drawbacks

The most crucial benefit of the framework is that the compiler performs faster simulations with the current version than it does without it. While this conclusion is not statistically significant for most models, it was for the largest ones, which is promising. Thanks to the greater number of equation simplifications simulation is undeniably quicker, meaning that in its current state it could be recommended that it is used in the JModelica.org compiler. However, one should consider weighing this against the increase in compile time that the framework causes. A decrease in simulation time of 1.09% (Section 5.1.2) might not warrant the increase in compile time of 9.09% (Section 5.1.1), even though the decreased simulation time was in comparison to the current compiler which is also performing optimizations.

The working order of the algorithms in the framework is relatively strict, which could cause problems for the developer of a new algorithm. There is currently no easy way of specifying the order in which algorithms execute, and there is no way of re-ordering `preRun()`, `run()`, and `postRun()` methods on their own (e.g. if AE is executed first in `preRun()`, it is also executed first in

the main loop and in `postRun()`). Since the logic in these three steps might be vastly different for different algorithms some features of an algorithm may be difficult or impossible to implement into the framework.

6.2 Future work

As the framework is relatively simple and only manages two algorithms, it can still be developed and improved. Below follows a few suggestions for what future research could entail.

6.2.1 Lifting some algorithm logic to framework level

There are still some logic left in AE and VP that could be handled by the framework instead. In respective `postRun()` method, both algorithms iterates over the lists of equations and variables and updates the content accordingly to what has previously been removed. By letting the framework handle the iterations, each list would only have to be visited once and the algorithms only keep logic to handle the separate objects in the lists.

Even though `WorkData` keeps a list of removed equations, both AE and VP still uses their own separate boolean tags in the `FAbstracEquation` class on top of that, to trace the equations they remove. This logic should only be kept at the framework level, but rather replacing the list in WorkData with a similar boolean tag. There are some logic in VP that has to be taken in concern when doing this, since some tagged equations should still exists in the model, but be moved to another list containing other types of equations.

There are still some rewrites taking place during the process of the equations in the worklist and also multiple rewrites performed during postRun. Reducing or even removing most of these rewrites could have a positive effect on the compile time.

6.2.2 Canonical form

As the canonicalization algorithm was never finalized and thus never part of the framework, benefits and drawbacks of our canonical form are currently unknown. A proper evaluation of the implementation is required in order to gauge the effects of a simple canonical for Modelica models. Since there are different possible canonical forms, there is incentive for investigating the strengths and weaknesses of different representations. It might even be possible that different canonical forms are better suited for different models. Keeping a selection of canonical forms in the framework might be relevant to developers, and it could even be possible for the compiler to deduce an optimal canonical form for a given model or scenario.

It is also important to consider that since the algorithm has not been tested, its impact on memory utilization is unknown. Currently the transformation to an *add-mul* tree likely uses more memory than the model it has transformed, since expansion of multiplications results in more additions and multiplications. E.g. $(a + b)(c + d)$ has two additions and one multiplication, whereas $(ac + ad + bc + bd)$ has three additions and four multiplications. This results in a larger AST than before. It is possible that canonicalization is something that should be done prior to simplification algorithms, but that the end result after

all simplifications should be translated to another canonical form involving other structures that use less memory.

6.2.3 Common Subexpression Elimination

As was mentioned in Section 1.1, implementation of the *Common Subexpression Elimination* algorithm would be a lot easier using a canonical form. Even if CSE is implemented in the compiler it might be prudent to consider revising the implementation to fit a canonical form, since upkeep of the algorithm or further modification to it would likely be facilitated.

To give an example of how a canonical form could improve CSE, consider the system of equations in Figure 20. In all equations, we see the expressions $(a + b)$ and $(c + d)$ occur. Due to this, it should be possible to replace both expressions with a temporary variable for which, during simulation, the value is calculated once instead of twice, as is done in system of equations in Figure 21. This symbolic simplification is at the moment not implemented in CSE.

$$\begin{cases} (a+b)-(c+d) & = x; \\ (a+b)*(c+d) & = y; \\ (a+b)+(c+d) & = z; \end{cases}$$

Figure 20: A system of equations where expressions could be replaced by variables.

$$\begin{cases} temp1 & = (a+b); \\ temp2 & = (c+d); \\ temp1 - temp2 & = x; \\ temp1 * temp2 & = y; \\ temp1 + temp2 & = z; \end{cases}$$

Figure 21: A system of equations where expressions have been replaced by variables.

Replacing CSE this way might be beneficial in terms of simulation time. In the first system of equations, three arithmetic operations per equation are required to evaluate the values to assign to x, y, and z, in total nine operations. In the second system of equations, only two operations are required to evaluate $temp1$ and $temp2$, and only one operation per equation is required to evaluate x, y, and z, in total five operations.

6.2.4 Modification of the current framework

The abstract class `SymbolicTransformationAlgorithm` carries strict restrictions on the implementation of an algorithm, and it is possible that several abstract classes may be beneficial to the framework's flexibility. It might be that some algorithms would be more clearly modelled by another abstract class using other methods than the current one.

7 Conclusion

Improving the performance of algorithms in a compiler for an equation-based programming language is a complex task. The algorithms can be simple in terms of their effect on a given equation, but this does not infer that they are simple in implementation. Modifying them so that they perform symbolic simplifications in a dynamic manner, could carry tangible benefits for the extent to which they can modify a system. We have shown that even a rudimentary implementation of an algorithm framework with re-assessment of modified data can be beneficial. We have also shown that the framework might not significantly improve simulation time of generated code. The current implementation was shown to be costly in terms of compile time, which infers that the trade-off between the two must always be considered. Further optimization of the modified AE and VP could possibly bring the compilation time down to a more acceptable level.

The framework constructed in this thesis provides the *JModelica.org* compiler with a module to integrate algorithms in. While it is relatively non-furbished in terms of features, it establishes a baseline using the two algorithms AE and VP. It is also constructed for usage with any algorithm, even though the current implementation might not be capable of using any type of algorithm. It provides future algorithm developers with an interface that directs algorithm design towards a structure, that will enable the algorithms to work on a model dynamically.

References

[1] A. Reilles, *Canonical Abstract Syntax Trees*. International Workshop on Rewriting Logic and Applications, 2006.

[2] P. Fritzson, *Principles of Object-oriented Modeling and Simulation with Modelica 3.3: A Cyber-physical Approach*, 2nd ed, Section 2.1.4 Variability, 2015

[3] M. Huth, M. Ryan, *Logic in Computer Science*, Cambridge [U.K.]: Cambridge University Press, 2004.

[4] G. Hedin, *An Introductory Tutorial on JastAdd Attribute Grammars*, Generative and Transformational Techniques in Software Engineering III / Lecture notes in computer science, pp. 166-200, 2011.

[5] J. Skeppstedt, *An Introduction to the Theory of Optimizing Compilers*, pp. 149-151, 2012.

[6] J. Skeppstedt, *An Introduction to the Theory of Optimizing Compilers*, pp. 151-157, 2012.

[7] S. Muchnick, *Advanced Compiler Design Implementation*, pp. 329-331, 1997.

[8] D. Richardson, *Some Undecidable Problems Involving Elementary Functions of a Real Variable*, The Journal of Symbolic Logic, vol. 33, no. 4, pp. 514-520, 1968.

[9] T. Ekman, G. Hedin, *Rewritable Reference Attributed Grammars*, ECOOP 2004 – Object-Oriented Programming, pp. 147-171, 2004.

[10] J. Kämpe, *Applying Constant Propagation in a Modelica compiler*. Lund, department of Computer Science, Faculty of Engineering LTH, 2013.

[11] P. Rizescu, *Applying Optimization Algorithms in a Modelica Compiler*. Lund, department of Computer Science, Faculty of Engineering LTH, 2014.

[12] J. Al Dallal, *How and When to Flatten Java Classes?*, IJCSEIT, vol. 4, no. 2, pp. 73-79, 2014.

[13] F. Casella, *Simulation of Large-Scale Models in Modelica: State of the Art and Future Perspectives*. Proceedings of the 11$^{\text{th}}$ International Modelica Conference, September 21-23, 2015, Versailles, France.

[14] X. Zhang, M. Burger, S. Osher, *A Unified Primal-Dual Algorithm Framework Based onBregman Iteration*, Journal of Scientific Computing, vol. 46, no. 1, pp. 20-46, 2010.

[15] R. Sinha, C. Paredis, V. Liang, P. Khosla, *Modeling and Simulation Methods for Design of Engineering Systems*, Journal of Computing and Information Science in Engineering, vol. 1, no. 1, p. 84, 2001.

[16] P. Bunus, "A Simulation and Decision Framework for Selection of Numerical Solvers in Scientific Computing", 39th Annual Simulation Symposium (ANSS'06), pp. 178-187, 2006.

[17] Modelica Standard Library, 2016. [Online]. Available: http://modelica.github.io/Modelica/help/Modelica.html. [Accessed: 18- May- 2016].

[18] JModelica.org, 2016. [Online]. Available: http://jmodelica.org/. [Accessed: 09- Jun- 2016].

[19] Modelica, *Modelica and the Modelica Association*, 2016. [Online]. Available: https://www.modelica.org/. [Accessed: 18- May- 2016].

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTATIONSDAG 2016-06-03

EXAMENSARBETE Iterative symbol simplification - extending the Jmodelica.org compiler with a framework for symbolic simplification algorithms
STUDENT Johan Calvén, Zimon Kuhs
HANDLEDARE Jon Sten (Modelon AB), Jonathan Kämpe (Modelon AB), Niklas Fors (LTH)
EXAMINATOR Görel Hedin (LTH)

Extending the *JModelica.org* compiler with a framework for optimization algorithms

POPULÄRVETENSKAPLIG SAMMANFATTNING **Johan Calvén, Zimon Kuhs**

Compiler optimization algorithms serve to increase the efficacy of the resulting programs, but usually operate sequentially without revisiting the code post-change. This work has implemented a framework in the JModelica.org compiler that allows its optimization algorithms to re-investigate modified programs and search for additional improvements.

In order to increase the efficiency of programs many algorithms for improving programs have been developed to be utilized by compilers, e.g. by removing unnecessary methods or unused variables. Usually these algorithms are run sequentially in a set order even though the changes made by one algorithm in a later stage could put the program in a state where there is additional opportunity for improvements by an earlier algorithm. By using a framework for these algorithms, designed with re-visitation in mind, it is possible for the algorithms to take turns modifying the program. This allows algorithms to find even more improvements without running them against the entire program ad infinitum.

We have constructed the basis for such a framework for the JModelica.org Modelica compiler. It provides a structure where the developer of a new algorithm is not required to consider in which way it works in the context of other algorithms, but where it will still benefit from the changes made by them.

Modelica is a programming language used to model physical systems using sets of equations. The sets of equations describe how different parts of the system relate to each other, e.g. the set of equations in figure 1.

$$\begin{cases} \texttt{a = b + c} \\ \texttt{b = c + 1} \\ \texttt{c = 1} \end{cases} \tag{1}$$

The algorithm *variability propagation* (*VP*) looks for variables that are constants (e.g. $c = 1$) and replaces the uses of that variable with the constant. In the figure, it would find the last equation and replace the cs in the other two. This would mean that the second equation would become $b = 1 + 1 = 2$, meaning that *VP* could improve the system further. In the framework, the altered equation will be marked so that the other algorithms, *VP* included, will know that it has changed and it might be possible to perform further changes. *VP* will then find and replace the new constant $b = 2$.