Tools for Understanding, Debugging, and Simulation Performance Improvement of Equation-Based Models

by

Martin Sjölund
This is a Swedish Licentiate’s Thesis

Swedish postgraduate education leads to a Doctor’s degree and/or a Licentiate’s degree. A Doctor’s degree comprises 240 ECTS credits (4 year of full-time studies). A Licentiate’s degree comprises 120 ECTS credits.

Copyright © 2013 Martin Sjölund

ISBN 978-91-7519-624-4
ISSN 0280–7971
Printed by LiU Tryck 2013

URL: http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-90096
Tools for Understanding, Debugging, and Simulation Performance Improvement of Equation-Based Models

by

Martin Sjölund

May 2013
ISBN 978-91-7519-624-4
Linköping Studies in Science and Technology
Thesis No. 1592
ISSN 0280–7971
LiU–Tek–Lic–2013:26

ABSTRACT

Equation-based object-oriented (EOO) modelling languages provide a convenient, declarative method for describing models of cyber-physical systems. Because of the ease of use of EOO languages, large and complex models can be built with limited effort. However, current state-of-the-art tools do not provide the user with enough information when errors appear or simulation results are wrong. It is paramount that the tools give the user enough information to correct errors or understand where the problems that lead to wrong simulation results are located. However, understanding the model translation process of an EOO compiler is a daunting task that not only requires knowledge of the numerical algorithms that the tool executes during simulation, but also the complex symbolic transformations being performed.

In this work, we develop and explore methods where the EOO tool records the transformations during the translation process in order to provide better diagnostics, explanations, and analysis. This information can be used to generate better error-messages during translation. It can also be used to provide better debugging for a simulation that produces unexpected results or where numerical methods fail.

Meeting deadlines is particularly important for real-time applications. It is usually important to identify possible bottlenecks and either simplify the model or give hints to the compiler that enables it to generate faster code. When profiling and measuring execution times of parts of the model the recorded information can also be used to find out why a particular system is slow. Combined with debugging information, it is possible to find out why this system of equations is slow to solve, which helps understanding what can be done to simplify the model.

Finally, we provide a method and tool prototype suitable for speeding up simulations by compiling a simulation executable for a parallel platform by partitioning the model at appropriate places.

This work has been supported by Vinnova in the ITEA2 OPENPROD and MODRIO projects, by SSF in the Proviking HIPo and EDOP projects, and by the National Graduate School of Computer Science (CUGS). The Open Source Modelica Consortium supports the OpenModelica project.

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

Thank you everyone who helped make this thesis come true. When I started thinking of everyone who contributed, I end up with a very long list which I had to shorten.

Without my supervisor, Peter Fritzson, this thesis would never have seen the light of day. Peter not only guided me to some interesting topics of research, he also got the funding and managed those projects. Most of my travels resulted in something useful, rather than sitting through meaningless meetings. This meant I had time to work on projects I enjoy.

Thanks to all my former and present colleagues at IDA. Especially to PELAB for being there all those coffee breaks. Relaxing a few minutes a day makes all the difference in the world.

I would like to thank everyone who contributed to the OpenModelica project. I would like to especially thank Adrian Pop, Per Östlund, Adeel Asghar, Jens Frenkel, Willi Braun, and Lennart Ochel for providing fast bug fixes and improving the compiler in the last few years. My work strongly depended on everything working as well as it did.

Francesco Casella, thanks for providing the many examples and problems that eventually lead to me starting work on the Modelica debugger. As we are mostly concerned with compiler construction and the development environment at PELAB it helped to have someone around that used Modelica for modelling.

Thanks to Bodil Mattsson Kihlström, Åsa Kärrman, Eva Pelayo Danils, and Liselotte Lundberg. You all did a great job as PELAB administrators, mostly making sure I actually signed all those travel expense reports and attached everything that should be there. Anne Moe, thank you for putting up with me enrolling in two different graduate schools, changing activity as a PhD student every 6 months, and coordinating the work on this thesis.

Finally, my family deserves my thanks for making me the man I am today. While most of you do not understand precisely what I do, you have still always supported me.

Martin Sjölund, April 2013
Contents

1 Introduction ................................................. 5
   1.1 Motivation ........................................... 5
   1.2 Contributions ....................................... 6
   1.3 Structure ............................................ 6

2 Background ................................................... 11
   2.1 Modelica ............................................... 11
      2.1.1 Standard Library .................................. 13
      2.1.2 OpenModelica .................................... 14
   2.2 Symbolic Manipulation ............................... 14
      2.2.1 Variable-Equation Dependencies ................. 16
      2.2.2 Sorting and Matching ............................ 17
      2.2.3 BLT Matrix ...................................... 18

3 Debugging .................................................... 21
   3.1 Background ............................................ 21
      3.1.1 Comparison with Traditional Debugging .......... 22
      3.1.2 Sources of Errors and Faults .................... 23
         3.1.2.1 Errors in the evaluation of expressions ... 24
         3.1.2.2 Assertion violations in models .............. 24
         3.1.2.3 Errors in the solution of implicit algebraic
                  equations .................................. 24
         3.1.2.4 Errors in the integration of the ODEs ....... 25
      3.1.3 Common Symbolic Operations ....................... 25
         3.1.3.1 Variable aliasing ............................ 25
         3.1.3.2 Known variables ............................. 26
         3.1.3.3 Equation Solving ............................ 26
         3.1.3.4 Expression Simplification ..................... 26
         3.1.3.5 Equation System Simplification ............... 27
         3.1.3.6 Differentiation ............................... 27
         3.1.3.7 Index reduction ............................... 27
         3.1.3.8 Function inlining ............................. 28
         3.1.3.9 Scalarisation ................................. 28
   3.2 Debugging Equation-Based Models .................... 29
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.1</td>
<td>Bookkeeping of Operations</td>
<td>31</td>
</tr>
<tr>
<td>3.2.1.1</td>
<td>Variable Substitution</td>
<td>31</td>
</tr>
<tr>
<td>3.2.1.2</td>
<td>Equation Solving</td>
<td>31</td>
</tr>
<tr>
<td>3.2.1.3</td>
<td>Expression Simplification</td>
<td>32</td>
</tr>
<tr>
<td>3.2.1.4</td>
<td>Equation System Simplification</td>
<td>32</td>
</tr>
<tr>
<td>3.2.1.5</td>
<td>Differentiation</td>
<td>33</td>
</tr>
<tr>
<td>3.2.1.6</td>
<td>Index reduction</td>
<td>33</td>
</tr>
<tr>
<td>3.2.1.7</td>
<td>Function inlining</td>
<td>34</td>
</tr>
<tr>
<td>3.2.1.8</td>
<td>Scalarisation</td>
<td>34</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Presentation of Operations</td>
<td>34</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Simulation Runtime</td>
<td>35</td>
</tr>
<tr>
<td>3.3</td>
<td>Conclusions and Future Work</td>
<td>38</td>
</tr>
<tr>
<td>4</td>
<td>Profiling</td>
<td>43</td>
</tr>
<tr>
<td>4.1</td>
<td>Background</td>
<td>43</td>
</tr>
<tr>
<td>4.1.1</td>
<td>gprof</td>
<td>43</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Valgrind</td>
<td>43</td>
</tr>
<tr>
<td>4.2</td>
<td>Implementation</td>
<td>44</td>
</tr>
<tr>
<td>4.3</td>
<td>Conclusions and Future Work</td>
<td>48</td>
</tr>
<tr>
<td>5</td>
<td>Parallel Simulation</td>
<td>51</td>
</tr>
<tr>
<td>5.1</td>
<td>Background</td>
<td>51</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Related Work</td>
<td>53</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Transmission Line Element Method</td>
<td>54</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Hopsan</td>
<td>57</td>
</tr>
<tr>
<td>5.1.4</td>
<td>Modelica</td>
<td>57</td>
</tr>
<tr>
<td>5.2</td>
<td>Example Model with Pressure Relief Valve</td>
<td>58</td>
</tr>
<tr>
<td>5.3</td>
<td>TLM in an Equation-Based Language</td>
<td>60</td>
</tr>
<tr>
<td>5.4</td>
<td>Partitioning</td>
<td>64</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Algorithm</td>
<td>65</td>
</tr>
<tr>
<td>5.5</td>
<td>Distributed Solver</td>
<td>65</td>
</tr>
<tr>
<td>5.6</td>
<td>Runtime</td>
<td>65</td>
</tr>
<tr>
<td>5.7</td>
<td>Future Work</td>
<td>69</td>
</tr>
<tr>
<td>5.8</td>
<td>Conclusions</td>
<td>70</td>
</tr>
<tr>
<td>6</td>
<td>Conclusion</td>
<td>71</td>
</tr>
<tr>
<td>6.1</td>
<td>Future Work</td>
<td>71</td>
</tr>
<tr>
<td>A</td>
<td>Flat Modelica</td>
<td>73</td>
</tr>
<tr>
<td>B</td>
<td>TLM Example</td>
<td>79</td>
</tr>
<tr>
<td>References</td>
<td></td>
<td>85</td>
</tr>
</tbody>
</table>
## List of Figures

2.1 Simple RC Circuit .......................................................... 12
2.2 Simple RC circuit simulation with input square wave \( v_p \),
capacitor voltage \( c_v \), and resistor voltage \( r_v \) ................. 13
2.3 OpenModelica Compiler Workflow ....................................... 15
2.4 Dependencies as a bipartite graph ...................................... 16
2.5 Bipartite graph using adjacency matrix representation ............ 16
2.6 Bipartite graphs using biadjacency matrix representation ........ 16
2.7 BLT matrix representation ................................................. 18

3.1 Using Information from the Translation in Subsequent Phases 29
3.2 The cumulative number of symbolic operations performed on
each equation in the EngineV6 model ................................. 36
3.3 OpenModelica Trace (Snipped for Brevity) .......................... 39

4.1 KCachegrind graph when examining eqFunction_564 ............... 45
4.2 Number of calls to the function per time step ....................... 47

5.1 Transmission line components calculate wave propagation through
a line using a physically correct separation in time .................... 56
5.2 The example system consists of a volume and a pressure relief
valve. Boundary conditions is represented by a constant flow
source and a constant pressure source .................................. 58
5.3 A pressure relief valve is designed to protect a hydraulic sys-
tem by opening at a specified maximum pressure ....................... 59
5.4 Pressure increases until the reference pressure of 10 MPa is
reached, where the relief valve opens ................................... 61
5.5 Comparison of spool position using different TLM implemen-
tations. The curves for delay, der, and sample overlaps almost
completely ................................................................. 62
5.6 Comparison of system pressure using different TLM imple-
mentations. The curves for delay, der, and sample overlaps
almost completely ......................................................... 62
5.7  Comparison of spool position in OpenModelica and Dymola. The curves overlap almost completely. The markers are in different positions because OpenModelica does not round or interpolate the output points in the result-file.  63
5.8  Adjacency matrices in lower triangular form.  64
5.9  Comparison of spool position using a volume split into more segments.  68
5.10 Comparison of system pressure using a volume split into more segments.  68
List of Tables

3.1 Mean overhead cost of adding debug information . . . . . . 30
3.2 Overhead cost of comparing expressions . . . . . . . . . . 33

5.1 Performance comparison of methods to model TLM in Modelica 64
5.2 Performance of parallelisation . . . . . . . . . . . . . . . . 67
Listings

2.1 Simple RC Circuit ................................. 12
2.2 Matching ........................................... 18
2.3 Sorting .............................................. 19
3.1 Alias Model with Poor Scaling ....................... 35
3.2 Runtime Error ..................................... 37
3.3 Runtime Error Dependency .......................... 37
3.4 OpenModelica Trace (XML snippet) .................. 40
3.5 OpenModelica ODE code fragment .................... 41
3.6 OpenModelica residualFunc3 .......................... 41
3.7 Modelica non-linear system .......................... 41
4.1 Simple non-linear equation .......................... 46
4.2 Binding equation is an array ......................... 47
5.1 OpenMP code ....................................... 66
5.2 Serial code ......................................... 66
A.1 Flattened RC Circuit ................................ 73
B.1 TLM Example ...................................... 79
Chapter 1

Introduction

1.1 Motivation

Equation-based object-oriented (EOO) modelling languages provide a convenient, declarative method for describing models of cyber-physical systems. The use of equations in models makes it easy to write a model in the same way as equations described in a standard textbook on the subject. The object-oriented nature of these languages makes it easy to compose a new model by connecting two or more existing model components. This fits nicely with how many companies do product development these days – combine standard off the shelf components to build a new product. The only thing that needs to be done is to test the components together, which can be performed by using a simulation tool. This saves time and money since it is possible to test many aspects of the final system without spending resources on building a physical prototype.

Understanding the complex transformations an EOO compiler performs on a typical input model is a daunting task that not only requires knowledge of the numerical algorithms used by the tool during translation and simulation. In order to understand why a transformed model is different from the original model, a user may have to spend a lot of time trying to figure out the sequence of symbolic transformations even for small models containing only tens of equations. The main reason for this difficulty is the large gap in abstraction between the model and the executable simulation code. EOO modelling languages are declarative and at a high level of abstraction where models express what to compute but not how to perform the computation.

An EOO compiler has to bridge that gap when generating efficient executable code. Another issue is the complexity of large models. EOO compilers can easily handle models containing tens of thousands of equations. Finding out what goes wrong in such big models is very hard without additional tool support. Searching through all these equations manually is possible but error-prone and certainly not practical.
This situation makes it hard for the modeller to find out and analyse how a simulation is performed computationally, for example in what order equations are solved, how many times a system of equations is evaluated, and the actual performance of different parts of the simulation. This is due to the complete lack of tools supporting even basic debugging and profiling of models in EOO languages.

Simulation of equation-based models using numerical solutions is a well-researched topic where reasonably efficient methods are known. A problem for further improved simulation performance is that most of these algorithms were developed with uni-processor systems in mind. However, it is no longer the case that single-processor performance of typical computers is increasing annually. To partially compensate this lack, new chips are designed to contain an increasing number of processor cores. Unfortunately, the recent annual increase of simulation speed has become stagnant due to a lack of research and improved methods for parallelisation of equation-based model simulation. Developing new improved methods for utilising the power of multi-core architectures is thus highly motivated.

1.2 Contributions

The main contributions of this work are:

- A method and prototype for tracing the symbolic operations performed in the back-end of a Modelica compiler to be used in debugging tool support and in generating explanations for the user.

- A design and implementation of a real-time profiler for simulations. The tool can find out which equation systems are slow to simulate. Combined with debug information, the tool can also point out why they are slow to simulate.

- A method to parallelise simulation models partly based on transmission line modelling, and partly on automatic dependency analysis and partitioning of systems of equations.

1.3 Structure

The structure of the thesis is as follows. First a general background (Chapter 2) is presented for all the chapters. Chapters 3, 4, and 5 each corresponds to one paper as shown below. These chapters contain enhanced and expanded versions of the original papers and have also been reworked to get continuity throughout the text. The thesis is concluded in Chapter 6. Moreover, some supplementary information including larger sections of source code have been placed in the Appendix.
• Chapter 3 is partly based on Martin Sjölund and Peter Fritzson. “De-
bugging Symbolic Transformations in Equation Systems”. In: Pro-
cedings of the 4th International Workshop on Equation-Based Ob-
ject-Oriented Modeling Languages and Tools. Ed. by François Cellier,
David Broman, Peter Fritzson, and Edward Lee. Zürich, Switzerland: 

• Section 3.1.2 provides some additional background. It is based on 
Adrian Pop, Martin Sjölund, Adeel Asghar, Peter Fritzson, and Fran-
cesco Casella. “Static and Dynamic Debugging of Modelica Models”. 
In: Proceedings of the 9th International Modelica Conference. Ed. by 
Martin Otter and Dirk Zimmer. Munich, Germany: Linköping Univer-

• Chapter 4 is partly based on Michaela Huhn, Martin Sjölund, Wuzhu 
Chen, Christian Schulze, and Peter Fritzson. “Tool Support for Model-
ica Real-time Models”. In: Proceedings of the 8th International Model-
cica Conference. Ed. by Christoph Clauß. Dresden, Germany: Linköping 
University Electronic Press, Mar. 2011. DOI: 10.3384/ecp11063537

• Chapter 5 is partly based on Martin Sjölund, Robert Braun, Peter 
in Modelica using Transmission Line Modeling”. In: Proceedings of the 3rd International Workshop on Equation-Based Object-Oriented 
Modeling Languages and Tools. Ed. by Peter Fritzson, Edward Lee, 
François Cellier, and David Broman. Oslo, Norway: Linköping Univer-

The following related paper on debugging gives a wider perspective and 
overlaps slightly with the concepts presented in this thesis:

• Adeel Asghar, Adrian Pop, Martin Sjölund, and Peter Fritzson. “Ef-
ficient Debugging of Large Algorithmic Modelica Applications”. In: Pro-
cedings of MATHMOD 2012 - 7th Vienna International Confer-
ence on Mathematical Modelling. Vienna, Austria, Feb. 2012. URL: ht
tp://seth.asc.tuwien.ac.at/proc12/web/html_sessions/MS23 .html (visited on 2013-04-01)

The following publications are not covered in this thesis. They are all related 
to interoperating with or enhancing the OpenModelica Compiler in some 
way:

• Anand Ganeson, Peter Fritzson, Olena Rogovchenko, Adeel Asghar, 
Martin Sjölund, and Andreas Pfeiffer. “An OpenModelica Python In-
terface and its use in PySimulator”. In: Proceedings of the 9th Inter-
national Modelica Conference. Ed. by Martin Otter and Dirk Zimmer.
Munich, Germany: Linköping University Electronic Press, Sept. 2012. DOI: 10.3384/ecp12076537


• Mohsen Torabzadeh-Tari, Peter Fritzson, Adrian Pop, and Martin Sjölund. “Generalization of an Active Electronic Notebook for Teaching Multiple Programming Languages”. In: Proceedings the 1st Annual Engineering Education Conference. Madrid, Spain, Apr. 2010. DOI: 10.1109/EDUCON.2010.5492457


Chapter 2

Background

2.1 Modelica

Modelica [27, 51] is an EOO language which has been designed to allow tools to automatically generate efficient simulation code with the important objective of facilitating exchange and re-use of models, model libraries, and simulation specifications. Modelica supports acausal modelling, which means that it is possible to model with equations and state a system of equations in the model without knowing how it should be solved. The Modelica tool will determine the computational solution order automatically and generate efficient executable code. Thus, it is possible to change how a model is solved by specifying different unknown variables to solve for. For example, to solve for and compute an unknown voltage instead of an unknown current will make the Modelica tool automatically generate a different computational order for the equation solution process. If traditional causal models and tools would be used instead, such a change in unknowns may necessitate rewriting the entire model from scratch.

The object-oriented aspect of Modelica allows creation of models in a re-usable and structured fashion. Composing a model from existing library model components is as easy as dragging components onto a diagram and connecting them together. If the goal is to model a serial RC circuit using a rectangular pulse-voltage source, the graphical view may look like Figure 2.1. Behind the scenes, the textual source code of the Modelica model exists and is automatically generated from the graphical model. This model source code is quite easy to read as shown in Listing 2.1. Note that during the translation process, the Modelica compiler will expand the connections, that is, the connect clauses in the model, into equations using Kirchhoff’s law. The connections between the capacitor, source and ground in this model are expanded into equations 2.1, 2.2, and 2.3.
Figure 2.1: Simple RC Circuit

Listing 2.1: Simple RC Circuit

```model RC
    import Modelica.Electrical.Analog.Basic;
    import Modelica.Electrical.Analog.Sources;
    Basic.Capacitor c;
    Basic.Resistor r;
    Basic.Ground g;
    Sources.PulseVoltage pv;
    equation
        connect(pv.n,g.p);
        connect(r.p,pv.p);
        connect(g.p,c.n);
        connect(r.n,c.p);
    end RC;
```
2.1. MODELICA

\[ c.n.i + g.p.i + pv.n.i = 0.0 \]  \hspace{1cm} (2.1)

\[ c.n.v = g.p.v \]  \hspace{1cm} (2.2)

\[ c.n.v = pv.n.v \]  \hspace{1cm} (2.3)

The text-based model does not only model the same system as the graphical model – both models are in fact different views of the same model. The only thing done to the model in Listing 2.1 was to remove the graphical annotations that give the coordinates and transformations of the components in order to reduce the volume of text, just as any good Modelica text editor will do.

There are many different ways in which a Modelica model can be used. It is possible to do parameter optimisation, steady state initialisation, hardware-in-the-loop simulation, real-time simulation, and more. The typical way a Modelica tool is used is to compile a model into an executable, run the executable, that is perform a simulation, and plot the results.

The result of simulating the simple RC circuit can be seen in Figure 2.2. The \( \times \) signal shows the square input wave \( pv.v \), the \( + \) signal shows capacitor voltage \( c.v \), and the \( \square \) signal shows the voltage over the resistor \( r.v \). The input wave is a perfect square and is modelled as a discrete component. Used together with continuous components, the components together model a hybrid system. Such hybrid continuous-discrete-time modelling is supported by Modelica, making the language more expressive than a purely continuous-time language.

2.1.1 Standard Library

The Modelica Standard Library (MSL) version 3.2 [53] contains 1280 models from many different application domains. It is the most well-known Mod-
elica library, it is open-source, supported and standardised by the Modelica Association. Being a standard, we sometimes use models from the standard library as reference models for benchmarks since they are readily available and well-known in the community.

The version of the MSL used in this work is the current development version\(^1\), slated to become version 3.2.1. The models are the same as in 3.2 but are much better supported by OpenModelica.

### 2.1.2 OpenModelica

OpenModelica [30] is an open-source Modelica-based modelling and simulation environment. We use OpenModelica as a base platform in our research but strongly believe that our work is of more general applicability and not tied to our implementation. It should be possible to use similar approaches also for other EOO languages.

The workflow of the OpenModelica Compiler is shown in Figure 2.3. In this work we are mostly concerned with flattened models, that is Modelica models without the object-oriented aspects. This is basically the pure equation system and is similar to equations in other equation-based languages, even those without object-oriented aspects (see Appendix A for an example of how flat Modelica models looks like). It should also be possible to use most parts of our work for a purely continuous-time modelling language; that is merely a subset of Modelica without event handling, discrete-time variables or hybrid systems.

In the OpenModelica Compiler, the compiler phase that compiles the flattened model to executable simulation code is called the backend. It handles all of the symbolic operations that are needed for the code generator to produce code that is both efficient and provides numerical stability.

### 2.2 Symbolic Manipulation

Numerical solvers are used to iteratively solve for the values of a selected set of unknown variables, the state variables, given the current state (the current values of the state and non-state variables) of the system. In order to simulate models correctly, equation-based simulation tools usually first perform a series of symbolic operations on the system of equations. An introduction may be found in [27]. It suffices to say that these operations are performed in order to transform the equation system to index-1 ordinary differential equation (ODE) form since most numerical solvers available require this form. More importantly, the performance of the numerical simulation can be improved one or several orders of magnitude by this symbolic pre-processing, as well as improving numerical stability of the solution process.

\(^1\)https://svn.modelica.org/projects/Modelica/trunk r6065
Figure 2.3: OpenModelica Compiler Workflow
2.2.1 Variable-Equation Dependencies

A Modelica backend is mainly concerned with variables and equations. In order to speed up many algorithms, the backend needs to keep track of which variables belong to which equations. The relation between equations and variables is represented as a bipartite graph (Figure 2.4). Bipartite graphs contain two sets of nodes that are disjoint. In our case, the variables belong to one set and the nodes to each other. A vertex between an equation and a variable denotes that the variable is used in this equation. A bipartite graph, like any graph, can be represented in many different ways. One
representation is the adjacency matrix shown Figure 2.5. It corresponds to the same graph as Figure 2.4, where the dotted vertex is represented with an X and the other vertices with *. For any bipartite graph, the adjacency matrix has the structure of
\[
\begin{bmatrix}
0 & B \\
B^\tau & 0
\end{bmatrix}.
\]
It is always possible to use the more compact biadjacency matrix representation for a bipartite graph (Figure 2.6).

While the biadjacency matrices in Figure 2.6 look dense, most Modelica models give rise to very sparse matrices. In OpenModelica, the internal data structure representation uses arrays of lists to get the answer quickly. Both the $B$ and $B^\tau$ are stored since it is then possible to query which variables are used in a given equation or which equations use a given variable. We will refer to the biadjacency matrix as the adjacency matrix even though the full adjacency matrix is never constructed.

### 2.2.2 Sorting and Matching

One goal of the model compiler backend in an equation-based simulation tool is to perform sorting and matching of the system of equations. For each equation, a matching algorithm selects one variable to solve for which becomes the left hand-side of the generated assignment. A sorting algorithm also topologically sorts the equations subject to the dependencies between variables and equations. The result is a directed adjacency matrix with strongly connected components, that is, it contains cycles.

\[
\begin{align*}
2x + y + 3z &= 10 \\
x + y + a \ast z &= 6 \\
a \ast x + 3y + 2z &= 13
\end{align*}
\]

If there is a strong connection between components, they must be solved simultaneously either as a linear system of equations (Equations 2.4, 2.5, and 2.6) or as a non-linear system of equations (Equations 2.7 and 2.8). The strong connected components are condensed into blocks of equations in order to make the graph free of cycles and thus possible to sort topologically.

\[
\begin{align*}
x^y + y &= 10x + z \\
y^x + x &= z
\end{align*}
\]

As an example of what the output of the sorting and matching is, see Listings 2.2 and 2.3. This is the actual output of the OpenModelica backend with the RC circuit as input. The sorting listing is actually the output of the strongly connected components algorithm, but in this case there are no equations that need to be solved simultaneously. After sorting, the equations can be translated into statements and executed sequentially from top to bottom.
Listing 2.2: Matching

var 1 is solved in eqn 6
var 2 is solved in eqn 7
var 3 is solved in eqn 8
var 4 is solved in eqn 2
var 5 is solved in eqn 4
var 6 is solved in eqn 3
var 7 is solved in eqn 5
var 8 is solved in eqn 1
var 9 is solved in eqn 9

\[
\begin{bmatrix}
E2 & E0 & E3 & E1 \\
V3 & * & & \\
V1 & * & * & \\
V0 & * & * & \\
V2 & * & * & *
\end{bmatrix}
\]

Figure 2.7: BLT matrix representation

Keep in mind that some of the variable names in the system are calculated only during initialisation and then considered as inputs to the system. The process of the sorting and matching is one of the main goals of a Modelica tool – being able to transform a system of equations, with no causality given and ending up with a sequence of operations for the computer to execute.

2.2.3 BLT Matrix

A block lower triangular (BLT) matrix is a matrix where the upper triangle is empty if the strong connected components are condensed into single nodes. If they are not, sub-blocks on the diagonal may extend slightly into the upper triangle. It is related to the biadjacency representation, sorting, and matching. Since the graph has been topologically sorted according to dependencies, there exists an absolute ordering of the equations. When the equations are put into an adjacency matrix in the order they were sorted, the matrix is in BLT form. The reason this is interesting is that it corresponds to a causal form of the system of equations where any sub-block of equations that should be solved has all its dependencies already calculated and thus appears above and to the left in the BLT form. Figure 2.7 shows that the execution order of equations \((2, 0, 3, 1)\) is safe. Another valid sorting would have been \((2, 0, 1, 3)\).

While there exists algorithms that use this BLT form, it is mostly useful for describing algorithms. In fact, the BLT matrix is never created explicitly.
Listing 2.3: Sorting

eqn 9: whenCondition1 = integer((time - pv.signalSource.startTime) / pv.signalSource.period, 0) > pre(pv.signalSource.count)

eqn 7: when whenCondition1 then
    pv.signalSource.T_start := time

eqn 6: when whenCondition1 then
    pv.signalSource.count := 1 + pre(pv.signalSource.count)

eqn 8: pv.v = pv.signalSource.offset + (if time < pv.signalSource.startTime or pv.signalSource.nperiod == 0 or pv.signalSource.nperiod > 0 and pv.signalSource.count >= pv.signalSource.nperiod then 0.0 else if time < pv.signalSource.T_start + pv.signalSource.T_width then pv.signalSource.amplitude else 0.0)

eqn 5: r.v = pv.v - c.v

eqn 2: r.R_actual = r.R * (1.0 + r.alpha * (r.T - r.T_ref))

eqn 3: r.v = r.R_actual * r.i

eqn 4: r.LossPower = r.v * r.i

eqn 1: r.i = c.C * der(c.v)
in OpenModelica. It is instead created indirectly by changing the ordering of the access indices to the equations and variables, referring to the original adjacency matrix since this gives fewer data structures to update when the system of equations changes.
Chapter 3

Debugging

3.1 Background

Equation-based object-oriented (EOO) modelling languages have significant advantages in describing large models since it is easy to construct a large hierarchical models by just connecting model components from a library.

However, in order to simulate such models efficiently, EOO simulation tools perform a large number of symbolic manipulation in order to reduce the complexity of models and prepare them for efficient simulation. By removing redundancy, the generation of simulation code and the simulation itself can be sped up significantly. For example, redundancy that could be removed are equations that could be executed fewer times than stated in the original model or functions that are called with the same input.

One of the drawbacks of this symbolic manipulation is that error-messages become not very user-friendly and understandable due to extensive symbolic manipulation, renaming, and reordering of variables and equations. For example, the following error message says nothing about the variables involved or their origin; the only useful thing about this message is that the simulation time of the error is displayed:

```
Error solving non-linear system 2
 time = 0.002
 residual[0] = 0.288956
 x[0] = 1.105149
 residual[1] = 17.000400
 x[1] = 1.248448
 ...
```

It is usually hard for a typical user of EOO tools to determine what symbolic manipulations have been performed and why. If the tool only generates a binary executable this is impossible. Even if the tool generates source code in some programming language (typically C), it is still quite hard to
understand what kind of equation system has been produced. This makes it difficult to understand where the model can be changed in order to correct modelling errors or to improve the speed and stability of the simulation.

While some tools allow exporting a description of the translated system of equations [58], this is not enough. After symbolic manipulation, the resulting equations no longer need to contain the same variables or have the same structure as the original equations. As a simple example, the resulting equation \( r = t \) in (3.1) holds given that \( r \) does not change value during events and \( t_{\text{start}} = r_{\text{start}} = 0 \).

\[
der(r) = 1.0 \iff r = 1.0 * (t - t_{\text{start}}) + r_{\text{start}} \iff r = t
\]

There are two main aspects of debugging application models. One is debugging the simulation executable itself [60]. In general, simulation tools can compute and store the value of each variable at each time step so that it is possible to observe if the results are correct. However, it is usually not possible to understand the cause(s) of slow simulations just by studying such data.

You need to look earlier in the compilation process where the equation system is symbolically optimised. Moreover, to understand the causes of possible erroneous variable values, you also need to be able to look into the chain of symbolic transformations of the original model. This can be viewed as debugging of the model compiler regarding the efficiency of the generated code with your application model as input, or alternatively as tracing and understanding how a modelling error affects the generated actual system of equations to be solved during simulation.

3.1.1 Comparison with Traditional Debugging

When most people hear the word debugging, they think of a statement or instruction-level debugger such as GDB [70] that provides breakpoints, interactively stepping through executed code, inspecting variable values, etc. A debugger is a computer program designed to help a programmer in the task of finding faults in programs. However, debugging is not limited to just instruction-level debugging.

Many developers prefer a more primitive form of debugging that uses print statements or logging. The reason is that it is often much easier to locate the error in this way, make a change, recompile, and run again. With a debugger, you need to set it up in a way that it stops at the correct location. Another issue is that debuggers often change the behaviour of the executable. For example, it might initialise variables to zero. Stack overflow problems and out of memory issues are also a problem for debuggers since if the computer runs out of memory, chances are the debugger will not work correctly. The goal of this work is to develop high quality low overhead debugging methods for EOO languages like Modelica.
3.1. BACKGROUND

There is a fundamental difference between the needs of a debugger for an EOO language and that of a debugger for a general-purpose programming language. There is also a need to be able to debug and understand the symbolic transformations and optimisations performed on the equation system defined in this application model. Without that, it is not possible to understand the generated code.

While it is possible to use a statement-level debugger to understand the symbolic transformations performed within the model compiler, this is hard to use for a regular user due to the complexity of the compiler itself, and requiring compiler sources and a debug version of the compiler. The user would also need some knowledge about the data structures used by the compiler since representing data in certain ways improves the performance of many algorithms used in these tools. This generally limits the use of such debuggers to the compiler developers themselves. Even then, developers who know how to use a debugger may not be able to use it since the source model is needed. Very often our users do not want to give us their complete models since they are classified. This requires the user to simplify his model to the minimal case that still shows the error, which is a very tedious process. If instead there was a tool that could localize the error, it would make the process of submitting good bug reports much easier.

Even if a compiler developer has the full model input, it is often problematic to use a traditional statement-oriented debugger simply due to the size of the equation systems that are being debugged as it becomes harder and harder to set up the debugger to only look at a single specific variable or equation. This often makes these traditional instruction-level debuggers unsuitable also from a compiler developer’s point of view.

Since equation-based modelling languages are quite different from general-purpose programming languages like C, it is understandable that debuggers made for conventional imperative languages do not work as well in the EOO language domain. What is needed is a domain-specific debugger that can show the user the flow of transformations and calculations. This is analogous to how a tool such as ANTLRWorks [10] can show which rules were used to parse some input given a certain ANTLR grammar, where the language to write ANTLR grammars is a domain-specific language.

We propose an approach to debugging the symbolic optimisation and transformation of equation systems that is usable by tool developers and modellers alike. The approach can be used for debugging during translation (compilation), statically after translation and dynamically during simulation runtime. The focus is on the method for producing the information required to later implement a user-friendly graphical user interface for the debugger.

3.1.2 Sources of Errors and Faults

There are a number of sources of errors and faults in a simulation environment. Some errors can be recovered automatically by the system. Others
should be reported and allow users to enter debugging mode. An error can also be a wrong value pointed out manually by a user.

Every solver employed within a simulation system at all levels should be equipped with an error reporting mechanism, allowing error recovery by the master solver, or error reporting to the end-user in case of irrecoverable error:

- the ODE solvers
- the functions computing the derivatives and the algebraic functions given the states, time, and inputs
- the functions computing the initial states and the values of parameters
- the linear equation solvers
- the nonlinear equation solvers

If some equation can be solved symbolically, without resorting to numerical solvers, then the symbolic solution code, for example a Modelica backend, should be equipped with diagnostics to handle errors as well. Below we give some causes of errors that can appear during the model simulation.

3.1.2.1 Errors in the evaluation of expressions

During the evaluation of expressions, faults may occur. Some of the causes may be:

- Division by zero.
- Functions called outside their domain, for example \( \sqrt{-1} \), \( \log(-3) \), \( \sin(2) \), \( \text{pow}(-2, 0.5) \).
- Wrong values of some variable(s) being computed, where the error is manually pointed out by a user or automatically detected as being outside min/max bounds.

3.1.2.2 Assertion violations in models

During translation, initialisation or simulation, assertions inside models can be triggered when the condition being asserted becomes false or when variables go outside min/max bounds.

3.1.2.3 Errors in the solution of implicit algebraic equations

During initialisation or simulation of DAE systems, implicit systems of equations must be solved. In the case of linear systems, the solver might fail because there is some error in evaluating the coefficients of the \( A \) matrix and of the \( b \) vector of the linear equation \( A \times x = b \), or because said problem
is singular. In the case of nonlinear equations \( f(x) = 0 \), the solver might fail for several reasons: the evaluation of the residual \( f(x) \) or of its Jacobian gives errors; the Jacobian becomes singular; the solver fails to converge after a maximum number of iterations.

### 3.1.2.4 Errors in the integration of the ODEs

In OpenModelica, the DAEs are brought to index-1 ODE form by symbolic and numerical transformation, and these equations are then solved by an ODE solver, which iteratively computes the next state given the current state. During the computation of the next state, for example by using Euler, Runge-Kutta, or a BDF algorithm such as DASSL [59], errors might occur. These errors may be errors from Sections 3.1.2.1–3.1.2.3. Additionally, the solver might fail because of singularity in the ODE, as in the case of finite escape time solutions, if the requested error tolerance cannot be achieved, or of discontinuities leading to chattering.

### 3.1.3 Common Symbolic Operations

In order to create a debugger adapted for debugging the symbolic transformations performed on equation systems, we need to state its requirements. There are many symbolic operations that may be performed on continuous-time or hybrid equation systems. The description of operations described below also includes a rationale for each operation since it is not always apparent why such operations are performed. There are of course many more operations which may be performed than the ones listed below. The operations shown below are the ones we judge are the most important and the ones that our debugger for models translated by the OpenModelica Compiler [30] should be able to trace.

#### 3.1.3.1 Variable aliasing

An optimisation that is very common in Modelica compilers is variable aliasing. This is due to the connection semantics of the Modelica language. For example, if \( a \) and \( b \) are connectors with the effort-variable \( v \) and flow-variable \( i \), a connection (3.2) will generate alias equations (3.3) and (3.4).

\[
\text{connect}(a, b) \\
\]
\[
a.v = b.v \\
\]
\[
a.i + b.i = 0 \iff b.i = -a.i
\]

In a simulation result-file, this alias relation can be stored instead of a duplicate trajectory, saving both space and computation time. In the equation system, \( b.v \) may be substituted by \( a.v \) and \( b.i \) by \( -a.i \), which may lead to further optimisations of the equations.
3.1.3.2 Known variables

Known variables are similar to aliased variables in that the compiler may perform variable substitutions on the rest of the equation system if it finds such an occurrence. For example, (3.5) and (3.6) can be combined into (3.7). In the result-file, there is no longer a need to store the value of $a$ at each time step; once is enough for known variables, which in Modelica are parameters and constants.

\[ a = 4.0 \] (3.5)

\[ b = 4.0 - a + c \] (3.6)

\[ b = 4.0 - 4.0 + c \] (3.7)

3.1.3.3 Equation Solving

If the tool has determined that $x$ needs to be solved for in (3.8), it is needed to symbolically solve the equation to produce a simple equation with $x$ on one side as in (3.9). Solving for $x$ is not always straight-forward and it is not always possible to invert user-defined functions such as (3.10). Since $x$ is present in the call arguments and the tool cannot invert or inline the function, it fails to solve the equation symbolically and instead solves it numerically using a non-linear solver during runtime.

\[ 15.0 = 3.0 \times (x + y) \] (3.8)

\[ x = \frac{15.0}{3.0} - y \] (3.9)

\[ 0 = f(3 \times x) \] (3.10)

3.1.3.4 Expression Simplification

Expression simplification is a symbolic operation that does not change the meaning of the expression, while making it faster to calculate. It is related to many different optimisation techniques such as constant folding. It is possible to change the order in which arguments are evaluated (3.11). Constant sub-expressions are evaluated during compile-time (3.12). Regarding Modelica models it is also allowed to rewrite non-constant sub-expressions (3.13) and choose to evaluate functions fewer times than in the original expression (3.14) since functions may not have side-effects. It is also possible for the compiler to use knowledge about the execution model in order to make expressions run faster (3.15) and (3.16).

\[ \text{and}(a, \text{false}, b) \Rightarrow \text{and}(\text{false}, a, b) \Rightarrow \text{false} \] (3.11)

\[ 4.0 - 4.0 + c \Rightarrow c \] (3.12)
\[ \max(a, b, 7.5, a, 15.0) \Rightarrow \max(a, b, 15.0) \]  
(3.13)

\[ f(x) + f(x) + f(x) \Rightarrow 3 \ast f(x) \]  
(3.14)

\[ \text{if cond then } a \text{ else } a \Rightarrow a \]  
(3.15)

\[ \text{if not cond then false else true } \Rightarrow \text{cond} \]  
(3.16)

### 3.1.3.5 Equation System Simplification

It is of course also possible to solve some equation systems statically. For example a linear system of equations with constant coefficients (3.17) can be solved using one step of symbolic Gaussian elimination (3.18), generating two separate equations that can be solved individually after causalisation (3.19). A simple linear equation system such as (3.17) may also be solved numerically using for example LAPACK [2] routines.

\[
\begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 4 \\ 5 \end{pmatrix}
\]  
(3.17)

\[
\begin{pmatrix} 1 & 2 \\ 0 & -3 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 4 \\ -3 \end{pmatrix}
\]  
(3.18)

\[
x = 2
\]

\[
y = 1
\]  
(3.19)

### 3.1.3.6 Differentiation

Symbolic differentiation [22] is used for many purposes. It is used to symbolically expand known derivatives (3.20) or as an operation during index reduction. Symbolic Jacobian matrices consisting of derivatives have many applications, for example to speed up simulation runtime [12]. If there is no symbolic Jacobian available, a numerical one might instead be estimated by the numerical solvers. Such a matrix is often computed using automatic differentiation [22] which combines symbolic and/or automatic differentiation with other techniques to achieve fast computation.

\[
\frac{\partial}{\partial t} t^2 \Rightarrow 2t
\]  
(3.20)

### 3.1.3.7 Index reduction

In order to solve (hybrid) differential algebraic equations (DAEs) numerically, simulation tools use discretisation techniques and methods to numerically compute derivatives and solve differential equations. These parts of the tools are often referred to as solvers. Certain DAEs need to be differentiated symbolically to enable stable numeric solution. The differential index of a general DAE system is the minimum number of times that certain equations in the system need to be differentiated to reduce the system to a set of
ODEs, which can then be solved by the usual ODE solvers, [28]. While there are techniques to solve DAEs of higher index than 1, most of them require index-1 DAEs or ODEs (no second derivatives). A common index-reduction technique uses dummy derivatives [50]; the OpenModelica Compiler default method currently combines the Pantelides method for index reduction [57, 69] with dynamic state selection [48, 49, 50].

3.1.3.8 Function inlining

Writing functions to do common operations is a great way to reduce the burden of maintaining code since each operation is defined by a function in only one place. The problem is that for function calls there is some overhead. This becomes a noticeable fraction of the computational cost for the whole invocation and computation for small functions. By inlining a function call (3.21), it is treated as a macro expansion (3.22) which avoids the overhead of calling the function and may increase the number of symbolical manipulations that can performed by the compiler on expressions such as (3.23).

In Modelica, the compiler may inline the call before or after index reduction. Both methods have advantages and disadvantages. Doing it after index reductions may provide a better result if the modeller has provided an analytic solution in the form of a derivative function. This causes a smaller expression to be differentiated if index reduction is required.

\[
2 \times f(x, y)/\pi \\
2 \times \pi \times (\sin(x + y) + \cos(x + y - y))/\pi \\
2 \times (\sin(x + y) + \cos(x))
\]

(3.21)  
(3.22)  
(3.23)

3.1.3.9 Scalarisation

Scalarisation is the process of expanding array equations into a number of scalar equations, usually one equation for each element of the corresponding array. By keeping array equations together instead of scalarising them early, the compiler backend saves time since it needs to perform a symbolic operation on only one equation instead of \(n\) equations for an array of size \(n\). However, if enough information is known about an equation (3.24), it can be beneficial to split it into scalar equations, one for each array element (3.25).

\[
(a, b, c) = (x, y, z) \\
a = x \quad b = y \quad c = z
\]

(3.24)  
(3.25)
3.2 Debugging Equation-Based Models

The choice of techniques for implementation of a debugger depends on where and for what it is intended to be used. Translation and optimisation of large application models can be very time-consuming. It would be good if the approach has such a low overhead that it can be enabled by default without further increasing the translation time. It would also be good if error messages from the runtime could use the debug information from the translation and optimisation stages to give more understandable and informative messages to the user (see Figure 3.1). A technique that is commonly used to help debugging is tracing (sometimes called logging depending on context). Two programs that output a trace in order to help in the debugging of C applications are `strace` and `valgrind`. The `strace` program (`truss` on some Unix systems) logs all system calls and signals that the attached process receives. Valgrind is a suite of tools for debugging (and profiling) programs. Its default tool, `MEMCHECK`, helps C programmers that use pointers in unintended ways. It is capable of reporting unreachable blocks of data, double free, accessing recently freed data, accessing non-allocated data and calling standard C functions with input that has undefined result. Both `strace` and `valgrind` display the information to the user by streaming text.

The simplest way of implementing tracing is to print a message to the terminal or file in order to log the operations that are performed. The problem here is that if an operation is rolled back, the log-file will still contain the operation that was rolled back\(^1\). Moreover, there is also a need to

---

\(^1\)In the OpenModelica Compiler, basic control flow uses backtracking which is due to the MetaModelica `matchcontinue` operator. The control flow is such that the compiler tries to perform one operation and if it fails half-way through, the result is discarded and
post-process the data if it is required that the operations should be grouped by equation.

A more elegant technique is to treat operations as meta-data associated to equations, variables or equation systems. Other meta-data that should already be propagated from source code to runtime include the name of the model component that an equation is part of, which file, line, and column that the equation originates from, and more. Whenever an operation is performed, the operation kind and input/output data is stored inside the equation as a list of operations. If the structure used to store equations is persistent this also works if the tool needs to roll back execution to an earlier state.

The cost of adding this meta-data is a constant runtime factor from storing a new head in the list. The memory cost depends on the language and programming style that the compiler itself was written in. If garbage collection or reference counting is used, the only memory cost is a small amount of memory to describe the operation (typically an integer and some pointers to the expressions involved in the operation). For garbage collection there may also be a performance penalty from calculating the transitive closure of active memory and copying the memory around that would otherwise be free space. If these expressions now referenced by being stored as meta-data would normally become garbage and subsequently be freed, the memory usage increases, but this is a small amount even for large models.

In OpenModelica, we disabled collection of the debug information to see the actual effects of this overhead in a real-world implementation. The results shown in Table 3.1 is the mean over nine runs of translating several models, excluding code generation, in the standard library [53]. Multiple EngineV6 components were used to increase the number of equations in the last models. Because OpenModelica uses garbage collection, the code sometimes runs faster if we do collect the debug information than if we do not. The reason is possibly that the garbage collector elects to allocate more memory sooner when it cannot free as much memory. The overhead cost, if there is one, is so low that collection of debug information is done by default in OpenModelica.

Table 3.1: Mean overhead cost of adding debug information

<table>
<thead>
<tr>
<th>Model</th>
<th>Eqs</th>
<th>With debug info</th>
<th>Without</th>
</tr>
</thead>
<tbody>
<tr>
<td>DoublePendulum</td>
<td>1298</td>
<td>5.39s</td>
<td>465MB</td>
</tr>
<tr>
<td>EngineV6</td>
<td>11116</td>
<td>46.2s</td>
<td>863MB</td>
</tr>
<tr>
<td>EngineV6*3</td>
<td>33348</td>
<td>55.3s</td>
<td>926MB</td>
</tr>
<tr>
<td>EngineV6*10</td>
<td>111160</td>
<td>196s</td>
<td>2021MB</td>
</tr>
</tbody>
</table>

the program flow returns to the old state but this time it does not try to perform the failed operation.
3.2.1  Bookkeeping of Operations

3.2.1.1  Variable Substitution

Here we will consider the elimination of variable aliasing and elimination of variables with known values (constants) as the same operation that can be done within a single phase. This can be performed by a fixed-point algorithm where substitutions are collected and recorded if any change was made (stop if no substitution is performed or no new substitution can be collected). For each alias or known variable, merge the operations stored in the simple equation $x = y$ before removing it from the equation system. For each successful substitution, record it in the list of operations for the equation.

The history of the variable $a$ in the equation system (3.26) could be represented as a more detailed version (3.27) instead of the shorter (3.28) depending on the order in which the substitutions were performed.

\[
\begin{align*}
a &= b, b = -c, c = 4.5 & \text{(3.26)} \\
a &= b \Rightarrow a = -c \Rightarrow a = -4.5 & \text{(3.27)} \\
a &= b \Rightarrow a = -4.5 & \text{(3.28)}
\end{align*}
\]

In equation systems that originate from a Modelica model it is preferable to view a substitution as a single operation rather than as a long chain of operations (chains of 50 cascading substitutions are not unheard of and makes it hard to get an overview of the operations performed on the equation, even though sometimes all the steps are necessary to understand the reason for the final substitution).

The OpenModelica Compiler tries to collect sets of aliases and select a single variable (doing everything in one operation) in order to make substitutions more efficient. However, alias elimination may still cascade due to simplification rules (3.29), which means that there is a need for a workaround for substitutions performed in a suboptimal order.

\[
a = b - c + d \Rightarrow a = b - b + d \Rightarrow a = d \quad \text{(3.29)}
\]

To efficiently handle this case, the previous operation is compared with the new one and if a link in the chain is detected, this relation is stored. When displaying the operations of an equation system for the user, it is then possible to expand and collapse the chain depending on the user’s needs.

3.2.1.2  Equation Solving

Some equations are only valid for a certain range of input. When solving an equation like (3.30), it is assumed by the compiler that the divisor is non-zero and it is eliminated in order to solve for $x$. The compiler records a list of such implicit assertions made (and their data sources for traceability).
Such an assertion may be removed if it is later determined that it always holds or if it overlaps with another assertion (3.31).

\[
x/y = 1 \Rightarrow x = y \quad (y \neq 0)
\]

\[
y \neq 0, 46.0 < y < 173.3 \Rightarrow 46.0 < y < 173.3
\]

### 3.2.1.3 Expression Simplification

Tracking changes to an expression is easy if there is a canonical form for expression simplification (record a simplification operation if the simplification algorithm says that the expression changed). However, if your simplification algorithm oscillates (as in 3.32) there is no canonical form and it is hard to use the simplification procedure as a fixed-point algorithm, which simplifies until nothing changes in the next iteration.

\[
2 \ast x \Rightarrow x \ast 2 \Rightarrow 2 \ast x
\]

The simple solution is to use an algorithm that assumes a canonical form for a fixed-point expression, that is, an expression for which expression simplification will terminate and not oscillate. Alternatively, use an algorithm that is is conservative in the sense that it does not save trace information that a change was made if it performs changes that may cause oscillating behaviour. Finding where this behaviour occurs is not hard for a compiler developer (for example, simply print an error message after 10 iterations).

If it is hard to detect that a change has actually occurred (due to changing data representation to use more advanced techniques), there may be a need to compare the input and output expressions in order to determine if the operation should be recorded. While comparing large expressions may be expensive, it is often possible to let the simplification routine keep track of any changes at a smaller cost, for example by assigning to a static variable when a change was performed.

We estimated the overhead cost of comparing the input and output expressions after each simplification operation by modifying the OpenModelica Compiler to do this extra check. The times without overhead includes all phases of translation excluding parsing and code generation. The input models are all from the standard library [53].

The result in Table 3.2 suggests that the overhead cost increases slightly as the model size increases. As the model size that increases, the redundancy in the system increases. This enables a number of optimisations most of which cause expression simplification to be run. It is previously known that the time Modelica backends take to translate a model do not scale linearly to the size of the equations [26], so the findings above should not be surprising.

### 3.2.1.4 Equation System Simplification

It is possible to store these operations as pointers to a shared and more "global" operation or as many individual copies of the same operation. We
Table 3.2: Overhead cost of comparing expressions

<table>
<thead>
<tr>
<th>Model</th>
<th>Eqs</th>
<th>Time</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>MultiBody EngineV6</td>
<td>11116</td>
<td>21.1</td>
<td>0.627 2.97%</td>
</tr>
<tr>
<td>Spice3 FOURBIT</td>
<td>10154</td>
<td>49.9</td>
<td>1.34 2.69%</td>
</tr>
<tr>
<td>MultiBody DoublePendulum</td>
<td>1298</td>
<td>1.66</td>
<td>0.050 3.19%</td>
</tr>
<tr>
<td>Fluid ControlledTanks</td>
<td>492</td>
<td>2.00</td>
<td>0.0428 2.14%</td>
</tr>
<tr>
<td>Digital Counter</td>
<td>282</td>
<td>1.14</td>
<td>0.0108 0.95%</td>
</tr>
<tr>
<td>Thermal IndirectCooling</td>
<td>163</td>
<td>0.700</td>
<td>0.0129 1.84%</td>
</tr>
<tr>
<td>Rotational Friction</td>
<td>98</td>
<td>0.593</td>
<td>0.00477 0.80%</td>
</tr>
<tr>
<td>Analog ChuaCircuit</td>
<td>44</td>
<td>0.517</td>
<td>0.00150 0.29%</td>
</tr>
<tr>
<td>Blocks Filter</td>
<td>34</td>
<td>0.652</td>
<td>0.00509 0.78%</td>
</tr>
</tbody>
</table>

would also recommend to store reverse pointers (or indices) from the global operation back to each individual operation as well, so that reverse lookup can be performed at a low cost.

Since the OpenModelica Compiler only performs limited simplification of strongly connected components, it is currently limited to only recording evaluation of constant linear systems. As more of these optimisations, for example solving for $y$ in (3.33), are added to the compiler, they will also need to be traced and support for them added in the debugger. Another example would be tracing the tearing operation [23], which causes the solution of a nonlinear system to be found more efficiently. Support for tearing was recently added to the OpenModelica Compiler but is not yet part of the trace.

\[
\begin{pmatrix}
1 & 1 & 2 \\
1 & i & 1 \\
-i & 1
\end{pmatrix}
\begin{pmatrix}
x \\
y \\
z
\end{pmatrix} =
\begin{pmatrix}
15 \\
18 \\
18
\end{pmatrix} \tag{3.33}
\]

### 3.2.1.5 Differentiation

Whenever the compiler performs symbolic differentiation in an expression, for example to expand known derivatives (3.34), this operation is recorded and associated to the equation in the internal representation. Currently the state variable is not eliminated as in (3.35), but if it would be done that operation would also be recorded.

\[
der(x) = der(time) \Rightarrow der(x) = 1.0 \tag{3.34}
\]

\[
der(x) = 1.0 \Rightarrow x = time + (x_{start} - time_{start}) \tag{3.35}
\]

### 3.2.1.6 Index reduction

For the used index reduction algorithm, dummy derivatives [50], any substitutions made are recorded, source position information is added to the
new dummy variable, as well as the operations performed on the affected equations. As an example for the dummy derivatives algorithm, this includes differentiation of the Cartesian coordinates \((x, y)\) of a pendulum with length \(L\) (3.36) into (3.37) and (3.38). After the index reduction is complete, further optimisations such as variable substitution (3.39), are performed to reduce the complexity of the complete system.

\[
x^2 + y^2 = L^2 \quad (3.36)
\]

\[
der(x^2 + y^2) \Rightarrow 2 \cdot (der(x) \cdot x + der(y) \cdot y) \quad (3.37)
\]

\[
der(L^2) \Rightarrow 0.0 \quad (3.38)
\]

\[
2 \cdot (der(x) \cdot x + der(y) \cdot y) \Rightarrow 2 \cdot (u \cdot x + v \cdot y) \quad (3.39)
\]

### 3.2.1.7 Function inlining

Since inlining functions may cause one or more new function calls to be added to the expression, functions are inlined until there are no more functions to inline in the expression or a maximum recursion depth is reached when dealing with recursive functions. Expressions are also simplified in order to reduce the size of the final expression as well as cope with a few recursive functions that have a known depth after inlining. When the compiler has completed inlining of calls in an equation, this is recorded as an inline operation together with the expression before and after the operation.

### 3.2.1.8 Scalarisation

When the compiler expands an equation into scalar equations, it stores the initial array expression, the index of the new equation, and the new expression.

### 3.2.2 Presentation of Operations

Until now we have focused on collecting operations and recording those as meta-data structures associated to equations in the system of equations. What can be done with this information? During the model translation phase it can be used for directly presenting information to the user. Assuming that the data is well structured it is possible to store it externally in a static database (for example an SQL database) or simply as structured data (for example an XML text file). That way the data can be accessed by various applications and presented in different ways according to the user needs for all of them. Our OpenModelica prototype outputs XML at present; this is sufficient for a proof of concept since the execution speed is sufficient.
Listing 3.1: Alias Model with Poor Scaling

```model AliasClass_N
constant Integer N=60;
Real a[N];
equation
der(a[1]) = 1.0;
a[2] = a[1];
for i in 3:N loop
   a[i] = i*a[i-1]-sum(a[j] for j in 1:i-1);
end for;
end AliasClass_N;
```

Our prototype provides an XSL transform to HTML+JavaScript to get a human-readable document.

The number of operations stored for each equation varies widely. The reason is that when a known variable, for example $x$, is replaced by for example the number 0.0, the compiler may start removing subexpressions. It may then end up with a chain of operations that loops over variable substitutions and expression simplification. The number of operations performed may scale with the total number of variables in the equation system if there is no limitation of the number of iterations that the optimiser may take [26]. This makes some synthetic models very hard to debug. The example model in Listing 3.1 performs $1 + 2 + \ldots + N$ substitutions and simplifications in order to deduce that $a[1] = a[2] = \ldots = a[N]$.

When testing these methods on a real-world example, the EngineV6 model\footnote{Modelica.Mechanics.MultiBody.Examples.Loops.EngineV6 from the MSL [53]}, the majority of equations have less than 4 operations (Figure 3.2), which means most equations were very easy to solve. The highest number of operations was 16 which is a manageable number to go through when there is a need to debug the model and to understand which equations are problematic. The 16 operations still require a nice presentation, like a visual diff, to quickly get an overview of what happened and why. Note that Figure 3.2 is a cumulative graph that includes both the initial equation system, the continuous-time equation system, the discrete-time equation system, and the known variables. Since known variables were included, most of these equations will actually not be part of the generated code and will not be interesting to debug unless it is suspected that the back-end produced the wrong result for a constant equation.

### 3.2.3 Simulation Runtime

In order to produce better error messages during simulation runtime, it would be beneficial to be able to trace the source of the problem. The toy
example in Listing 3.2 is used to show the information that the augmented runtime can display when an error occurs. The user should be presented with an error message from the solver (linear, non-linear, ODE, or algebraic equation does not matter). Here, the displayed error comes from the algebraic equation handling part of the solver. It clearly shows that $\log(0.0)$ is not defined and the source position of the error in the concrete textual Modelica syntax form (the Modelica code that the user makes changes for example to fix the problem) as well as the class names of the instances at this point in the instance hierarchy (which may be used as a link by a graphical editor to quickly switch view to the diagram view of this component). The tool also displays the symbolic transformations performed on the equation, which can help in debugging additional problems with the model.

The equation that threw an exception also depends on other variables. These variable dependencies are displayed as a link (which opens up a view with the information in Listing 3.3), enabling a user to debug the input variables of the equation. Using this view, it can quickly be realised that $v$ is a constant, which causes the expression $v \ast (1.5 - v) - time$ to become 0 at some point in time. If $v$ was time-variant, it is possible that the equation could always be calculated.

If the user interface is intuitive enough this should give users all the information they need to debug their models. This includes being able to click any variable name in an equation and getting to the equation where this variable is solved as well as plots of the current as well as historic values of the variable. All possible error sources (see Section 3.1.2) should be able to display such error messages and open up a browser/debugger of
3.2. DEBUGGING EQUATION-BASED MODELS

Listing 3.2: Runtime Error

Error: At t=0.5, block1.u = 0.0 is not in the domain of log (>0)

Source equation: [Math.mo:2490:9-2490:33] y = log(u)


Flattened equation: block1.y = log(block1.u)

Manipulated equation: y = log(u)

<Operations>
  variable substitution: log(block1.u) = log(u)
<Depending on equations (from BLT)>
  u <:link>

Listing 3.3: Runtime Error Dependency


Source component: <top> (MyModel)

Flattened equation: u = v*(1.5-v)-time

Manipulated equation: u = 0.5-time

<Expand operations>
  simplify: 0.5*(1.5-0.5)-time => 0.5-time
  variable substitution: v*(1.5-v)-time => 0.5*(1.5-0.5)-time
the optimised equation system.

This information can be easily made available in generated code by loading the database containing the transformation traces and annotating the source code with the key to the correct row in the table.

3.3 Conclusions and Future Work

A prototype design and implementation for tracing symbolic transformations and operations in a compiler (the OpenModelica Compiler [30]) for an EOO language (Modelica [51]) has been constructed. It has an overhead in the order of 0.01% of the total compilation time excluding the cost of writing the information to file. Writing all the information to a verbose XML-file is cheap in the general case but may take as long as the rest of the compilation time used by the backend for some models with very large expressions where only a small part of the expression is treated in each operation. If future need should arise, the trace information could alternatively be output to a more efficient binary file format. The current implementation also considers that some operations, for example cascading alias elimination, should be considered as an expandable operation that combines a simple overview and a detailed view. Further, a design for constructing a debugger using this data has been provided. This design can be used for static debugging, runtime debugging, and to provide understandable runtime errors.

The Functional Mock-up Interface (FMI) [9] is a standard for exchanging executable models, called Functional Mock-up Units (FMUs), which OpenModelica supports. Since FMI standardises variable names and initial values of variables and parameters in an XML file, OpenModelica uses a very similar XML format for the initialisation file also when it is not producing FMUs. And since the runtime already uses XML, the debug information is provided in a separate XML-file which is also used by the runtime to get user-friendly names and dependencies of equations in order to produce better error-messages. The runtime system of OpenModelica [30] simulations needs to be extended to take full advantage of the collected information. For example, the compiler currently does not use the debugging information to give detailed runtime errors, only the name and index of the equation, but plan to report more detailed messages in the future.

Most of the sources of errors and faults in Section 3.1.2 should be possible to debug using this approach. All equations have debug information attached to them, which means it can be given to the numerical solvers. Any expression that is evaluated is part of an equation which means the same debug information can be provided. User-provided errors in calculated values should also be possible to trace back to the source. There are two problems remaining for localising the errors.

The first problem is that it is not clear how errors in the OpenModelica backend would attach the graphical debugger to it since the full debug information is not yet available. One possibility is to not launch the debugger,
3.3. CONCLUSIONS AND FUTURE WORK

\[ 0.0 = y - \sin(x \times \text{time}) \]
M.mo:6:3-6:18

\[ 0.0 = x + (-y) \times \text{time} \]
M.mo:5:3-5:15

Operation 1 Simplify: \( x = y \times \text{time} \times 1.0 \)

Operation 2 Substitution: \( y \times \text{time} \times z \)

Nonlinear System (index 3)
Solves for variables (2 variables, 2 equations): \( x, y \)

Equations #2 #1

Figure 3.3: OpenModelica Trace (Snipped for Brevity)

instead providing error-messages that use the required debug information that was already collected. This would probably be the preferred method at least for compiler warnings. The second problem is that none of the numerical solvers use the debug information yet.

A graphical debugger and browser for the transformations also needs to be designed. Without this, a regular user will not be able to take full advantage of the trace. At the time of this writing, the trace uses HTML but not all parent or child links are present. A user at intermediate skill level could still take advantage of the current trace: Figure 3.3 is the HTML version as displayed by Firefox, Listing 3.4 is the XML representation. The current trace is at least more readable than the translated source code (Listings 3.5 and 3.6) which does not include any information about the origin of the equation. The system corresponds to Listing 3.7.

What also needs to be implemented is a set of special-purpose, or domain-specific, debuggers for example to guide the user through the process of debugging nonlinear systems of equations that either fail or produce an unexpected result.

The XML format, part of which can be deduced from Listing 3.4, has a few flaws which is why the current schema is not yet published. One thing we would like to do is use our own diff algorithm to produce a format that displays a diff that has some knowledge of the expressions used represented by the tool. XML code like <diff><rm>abc</rm><add>b</add></diff> instead of <before>abc</before><after>b</after> would allow us to produce a diff like abcb instead of applying a general-purpose textual diff and ending up with ab. The former is preferred since the current diff algorithm likes to cut words in the middle which makes often makes it hard to see the actual code.

This work focused mainly on operations performed on individual equations. By analysing the BLT matrix of the equation system, a user could additionally be presented with equations that need to be solved before or
Listing 3.4: OpenModelica Trace (XML snippet)

```xml
<equation index="2">
  <residual>x + (-y) * time</residual>
  <source>
    <info file="debug/M.mo" lineStart="5" lineEnd="5" colStart="3" colEnd="15"/>
  </source>
  <operations>
    <simplify>
      <before>x = y * time * 1.0</before>
      <after>x = y * time</after>
    </simplify>
    <substitution>
      <before>y * time * z</before>
      <exp>y * time * 1.0</exp>
    </substitution>
  </operations>
</equation>

<equation index="3">
  <nonlinear indexNonlinear="0">
    <var>x</var>
    <var>y</var>
    <eq index="2"/>
    <eq index="1"/>
  </nonlinear>
</equation>
```
Listing 3.5: OpenModelica ODE code fragment

```c
/* extrapolate data */
data->simulationInfo.nonlinearSystemData[0].nlsx[0] = $Px;
data->simulationInfo.nonlinearSystemData[0].nlsxOld[0] = _$Px(1) /*old*/;
data->simulationInfo.nonlinearSystemData[0].nlsxExtrapolation[0] = extrapolate(data, _$Px(1) /*old*/, _$Px(2) /*old2*/);
data->simulationInfo.nonlinearSystemData[0].nlsx[1] = $Py;
data->simulationInfo.nonlinearSystemData[0].nlsxOld[1] = _$Py(1) /*old*/;
data->simulationInfo.nonlinearSystemData[0].nlsxExtrapolation[1] = extrapolate(data, _$Py(1) /*old*/, _$Py(2) /*old2*/);
/* NLS 0 is associated with residualFunc3 */
solve_nonlinear_system(data, 0);
/* write solution */
$Px = data->simulationInfo.nonlinearSystemData[0].nlsx[0];
$Py = data->simulationInfo.nonlinearSystemData[0].nlsx[1];
```

Listing 3.6: OpenModelica residualFunc3

```c
DATA* data = (DATA*) dataIn;
modelica_real tmp1006;
$Px = xloc[0];
$Py = xloc[1];
res[0] = ($Px + ((-$Py) * time));
tmp1006 = sin(($Px * time));
res[1] = ($Py - tmp1006);
```

Listing 3.7: Modelica non-linear system

```modelica
model M
  Real x,y,z;
equation
  z = 1.0;
x = y*time*z;
y = sin(x*time);
end M;
```
after the selected equation. Neither the BLT matrix nor dependencies for individual equations are stored in the current XML-file, but it should be easy to add since all the information is already available. With more bookkeeping it should even be possible to show how the relationships change during the optimisation process.

We also have not yet considered operations performed on algorithmic code or optimisations that can be performed on the hybrid parts of the system, for example dead code elimination of unreachable discrete events.

Information about the number of states, sizes of equation systems, and similar statistics are very useful to get an overall feeling of how efficient a model is to simulate. OpenModelica does keep track of this and is able to show the information to a user. But it is hard to obtain this information. It usually involves running the compiler one additional time, from command-line with some additional flags set, either because the information was not requested the first time or because a graphical client was used. It would be useful if the debug information included all such relevant information to give the user the complete picture.
Chapter 4

Profiling

4.1 Background

Standard computer program performance profiling is based on the premise that it is possible to tell a lot about the performance of a program by looking at some key performance indicators. The measured indicators are usually memory and CPU consumption, and they are measured dynamically on a running program. There are many ways to obtain these measurements. We will first describe two common CPU-time profilers for C-based programs below.

4.1.1 gprof

The gprof tool is a profiler that works by making the compiler perform code instrumentation (like gcc -pg). The timing measurements are gathered by interrupting the program at regular intervals and noting the function within which it currently executing. This is statistical sampling. After the program (or multiple runs of the same program) finishes, the output files can be analysed.

In our experience, (GNU) gprof is not very suitable for huge amounts of generated code since it is very common to get a measurement of no time spent in short functions. This also only works really well if every single source-file is recompiled, including supporting libraries, with the -pg flag. While running the instrumented code is fast, analysing the data afterwards is slow. An advantage is that gprof is almost universally available – GNU gprof is supported on any platform GCC runs on.

4.1.2 Valgrind

Valgrind is known as a memory debugging tool, but it is actually a framework for creating dynamic analysis tools. One such use is profilers, of
which Valgrind currently provides three different ones (massif, cachegrind, and callgrind). Of these, the functionality of callgrind is the most similar to gprof. The way it is used is quite different, however. Valgrind performs code instrumentation on the binary code and requires neither compiler help nor source code. Depending on which Valgrind tool is used, better results may be provided if the code is compiled with debug symbols (\texttt{gcc -g} gives line numbers instead of only function names) and might get more intuitive results if optimisations like function inlining are disabled. Because Valgrind is essentially a virtual machine, the executed code for callgrind runs approximately 100 times slower than usual. It is however very easy to use callgrind which in most cases compensates for the steep performance penalty. Kcachegrind, the tool that displays the result of the analysis, is very fast since the callgrind tool does more processing than gprof. In our experience, it is also more accurate than gprof.

Figure 4.1 shows the kind of information KCachegrind displays when examining \texttt{eqFunction.564} in the generated code of Modelica.Fluid.Examples.ControlledTankSystem.ControlledTanks from the standard library [53]. The total execution times of the called child functions are only displayed if \texttt{eqFunction.564} is a parent in the call chain, and vice versa for parent functions. Any other nonlinear code that uses the same solvers are not included. The user interface of KCachegrind displays a little more information than it exports; it can show that \texttt{eqFunction.564} uses 94.76\% of the total execution time instead of the absolute time it takes to execute. Since \texttt{eqFunction.564} uses up such a high proportion of the execution time, the function is by far the most important one to investigate. A quick analysis of the graph shows that the generated residual function (\texttt{residualFunc564}) that is called from within the nonlinear solver (\texttt{solveHybrd}) is fast compared to the other functions used by the nonlinear solver – it uses less than 10\% of the time.

The problem here is that the residual function needed to be called 2.4 million times during the simulation because the solver had difficulty solving it. In order to improve performance of the model some optimisation is required to make it easier for the nonlinear system solver to find the roots. An alternative would be to modify the model to become linear instead.

One disadvantage of Valgrind is that it tied to Linux. There exists ports for OSX and FreeBSD but these are not as well supported. Since Windows is not supported, it is not a tool that could be integrated on all supported platforms in the OpenModelica suite.

4.2 Implementation

The OpenModelica profiler uses compiler-assisted source code instrumentation. It is implemented in the code generation as a macro set, which means zero overhead unless profiling is enabled. Even with profiling enabled, the overhead cost is low compared to the cost of solving most non-linear systems of equations. The instrumentation is performed by compiling a model with
Figure 4.1: KCachegrind graph when examining eqFunction_564.
a preprocessor macro set enabled and running the executable with the time measurement flag. The time measurement uses the best real-time clock available on the platform used\(^1\). Windows, Linux, and OSX are all supported but only Linux has a choice of measuring either CPU-time or real-time (wall time). All measurements are accumulated using integer operations, which means that they are fast and no information is truncated which might be the case if floating-point operations would be used. The output is written to a file at each time step.

The implementation is suitable for deployment on real-time applications since the overhead cost of execution is very low. The code also needs to run only once unlike statistical profilers. Unlike gprof every entry of a measured block is recorded. The advantage to using a statistical approach is that accurate average numbers are provided, but for real-time systems worst-case execution time needs to be considered to also meet deadlines. Some statistical methods can provide this – gprof cannot since it does not record execution times of function calls. Due to the way simulations are performed, our method can be used together with statistical methods. Simply put, the numerical solvers perform many time steps in order to get accurate simulation results. Analyse this data to calculate an estimate of worst and average cases.

The code runs on all platforms supported by OpenModelica and is not limited to real-time platforms. Profiling is of general interest because small changes to a model may have a large impact on the simulation time. By providing a profiler to both developers and users of a simulation tool, performance issues can more easily be discovered. In OpenModelica, use the command `simulate(Model, measureTime=true)` to enable profiling or check the performance profiling box in the OpenModelica Connection Editor (OMEdit).

### Listing 4.1: Simple non-linear equation

```model SimpleNonLinear
    Real x = cos(x);
end SimpleNonLinear;
```

As a result it should be easy to improve the quality of the tool. In OpenModelica, profiling has detected many scaling issues and detected poor code generation for many operations that work on arrays. By observing the output of the profiler, it can be seen that in the SimpleNonLinear model example (Listing 4.1), \( \cos(x) \) will be called 3 times in each time step. While it is possible to determine the value of \( x \) during compile-time or initialisation, OpenModelica does not yet perform these optimisations. It is thus important to model it as an initial equation if the result is needed for a real-time simulation.

The profiling also works for any user-defined function that is called. In the ArrayCall example (Listing4.2), tenCos is called a total of 593 times for

\(^1\)OpenModelica uses `clock_gettime`, `QueryPerformanceCounter`, or `mach_absolute_time` depending on what is available.
Listing 4.2: Binding equation is an array

model ArrayCall
  function tenCos
    input Real r;
    output Real array [10];
  algorithm
    array := cos(r*(1.0:10.0));
  end tenCos;
  Real x[10] = tenCos(time);
  Real y;
  equation
    der(y) = x[3];
  end ArrayCall;

Figure 4.2: Number of calls to the function per time step.
499 time steps. If the model is changed so that \( \text{der}(y) \) does not depend on \( x \), tenCos is called exactly once per time step in addition to 4 times during initialisation. While this behaviour is intended when using the default DAE solver, \text{dassl} \text{, it makes the graph} (Figure 4.2) \text{look a little odd.} \text{Dassl} \text{performs long steps and interpolates output points if the approach is still numerically stable.}

In order to determine the length of the step to take, the states are calculated a couple of times before the decision is made. This results in tenCos being called between 2 and 4 times for some output points, when \text{dassl} performs some internal calculations, and 1 time for the steps it determined were safe to take without choosing a smaller step size. If real-time simulations were the goal, a solver with a fixed step size would be selected since the number of times each function was called would be known.

### 4.3 Conclusions and Future Work

By using profiling techniques, it is possible to detect which equations cause low performance in a simulation. By using simple techniques to sample and log high-precision clocks at each time step, it is possible to see how much time is spent calculating each strongly connected component. This essentially means that the profiling returns a set of equations that calculates a set of variables. But these are variables in the optimised equation system which means that it is hard to know their origin [13]. By combining the profiling information and the debugging information from Chapter 3 it would be possible to not only see which systems were performing slowly, but also why they were performing slowly.

Measured functions should have an option to measure each function call separately. Currently only the top-most function is measured for efficiency but we have determined that it is often functions called from within this function that take the longest to execute. There is currently no support to find out which function this is, which means another profiler, like Valgrind, has to be used to find the function that actually takes most time. Adding this support is a very minor extension to the existing profiler that should only slightly impact performance.

The simulation runtime keeps track of miscellaneous statistics that could be usable for understanding other performance problems. One such statistic is the number of state events that are triggered. In the future, the profiling information should include all such relevant statistics.

All these proposed additions to the profiler are items for which information can be collected efficiently. There will be no major impact on real-time performance. All information that could be useful for simulations in general cannot be collected if real-time performance is required. This information could be collected if a flag was set to collect additional information.

One alternative approach to consider is creating our own replacement to \text{KCacheGrind} \text{that combines the information from callgrind and debug infor-}
mation to get a more accurate result than in the version we have presented, for example collecting execution times of all scalar equations. The obvious disadvantage would be that the performance penalty is so great that it could not be used for real-time profiling.
Chapter 5

Parallel Simulation

This chapter is based on the work achieved in [65]. The initial paper did not cover parallelisation, but gave the basic framework to perform parallelisation. Here, we present a more complete solution to the problem of parallelisation of equation-based models.

5.1 Background

An increasingly important way of creating efficient computations is to use parallel computing, that is, dividing the computational work onto multiple processors which are available in multi-core systems. Such systems may use either a central processing unit (CPU) [36] or a general-purpose graphics processing unit (GPGPU) [40, 76]. Since multi-core processors are becoming more common than single-core processors, it is becoming important to utilise such resources. Taking full advantage of the new parallel architectures requires support in compilers and development tools.

However, while parallelisation of models expressed in EOO languages is not an easily solved task, the increased performance if successful is very important. A hardware-in-the-loop real-time simulator using detailed computationally intensive models certainly needs the performance to keep short real-time deadlines, as do large models that take days or weeks to simulate. There are a few common approaches to parallelism in programming:

- No parallelism in the programming language, but accessible via library calls. The computational work can be divided by executing several processes or jobs at once, each utilising one CPU core.

- Explicit parallelism in the language. Language constructs are introduced so that the programmer can express parallel computations using several CPU cores.

- Automatic parallelisation. The compiler itself analyses the program or model, partitions the work, and automatically produces parallel code.
Automatic parallelisation is the preferred way because the users do not need to learn how to do parallel programming, which is often error-prone and time-consuming. This is even more true in the world of equation-based languages because the modeller can be a systems designer or modeller with no real knowledge of programming or algorithms.

It is not trivial to achieve automatic parallelisation of models in equation-based languages. Not only is it needed to decide which processor should perform a particular operation; it is also needed to determine in which order to schedule computations needed to solve the equation system.

The scheduling problem can become quite difficult and computationally expensive for large equation systems. It might also be hard to split the sequence of operations into several separate threads due to dependencies between the equations [3].

There are methods that can make automatic parallelisation easier by introducing parallelism over time, for example distributing solver work over time [62]. However, parallelism over time gives very limited speedup for typical ODE systems of equations. Pipelining, inlining and distributing the solver as phases on different processors [45, 46] is a method that removes the central solver and gives some additional speedup. However, the potential for splitting a solver into pieces is usually limited. In [46], splitting a Runge-Kutta solver into three phases gives additional speedup of a factor of two.

A single centralised solver is the normal approach to simulation in most of today’s state-of-the-art open-source and commercial simulation tools. Although great advances have been made in the development of algorithms and software, this approach suffers from inherent poor scaling. That is, execution time grows more than linearly with system size.

By contrast, distributed modelling, where solvers can be associated with or embedded in subsystems, and even component models, has almost linear scaling properties. Special considerations are needed, however, to connect the subsystems to each other in a way that maintains important properties such as numerical stability and robustness without introducing unwanted numerical effects. Technologies based on bilateral delay lines [6], also called transmission line modelling (TLM), have been developed for a long time at Linköping University.

This method has been successfully implemented in the HOPSAN [44] simulation package, which is currently almost the only simulation package that regularly utilises this technology, within applications in mechanical engineering and fluid power. Recent versions of HOPSAN include multi-core support [11]. TLM has also been demonstrated in [42] and subsequently in [14]. Although the method has its roots already in the sixties, it has never been widely adopted, probably because its advantages are not evident for small applications, and that wave-propagation is regarded as a marginal phenomenon in most areas, and thus not well understood. Moreover, the limited adoption may also have been caused by the fact that the general applicability of the method to different application areas and models is not
well understood.

In this chapter we focus on introducing distributed simulation based on TLM technology in Modelica and combining this with solver inlining which further contributes to avoid the centralised solver bottleneck. We also demonstrate these techniques work in the favour of parallelism when performing simulations.

Summarising the main contents of this chapter:

- We propose using a structured way of modelling with model partitioning using transmission lines in Modelica that is compatible with existing Modelica tools (Section 5.3).

- We show that such a system can use distributed solvers and may contain subsystems with different time steps, which may improve simulation performance dramatically (Section 5.5).

- We demonstrate that distributed simulations using TLM can be combined and that the resulting simulation results are essentially identical to those obtained using the HOPSAN simulation package.

- We show that a simple partitioning algorithm can indeed be used to parallelise a Modelica model using transmission lines.

5.1.1 Related Work

Several people have performed work on parallelisation of Modelica models [3, 16, 18, 45, 46, 47, 55, 71, 78], but there are still many unsolved problems to address.

The work most similar to ours is [55], where Nyström uses transmission lines to perform model partitioning for parallelisation of Modelica simulations using computer clusters. The problem with clusters is the communication overhead, which is huge if communication is performed over a network. Real-time scheduling is also a bit hard to reason about if cluster nodes are connected through TCP/IP. Today, there is an increasing need to parallelise simulations on a single computer because most CPUs are multi-core. One major benefit is the decrease of the communication costs; shared memory may be used with low overhead in inter-processor communication. If the program fits in the cache, the overhead is even lower. The OpenModelica implementation also does not lock memory which should slightly improve performance of simulations.

Another difference between the two implementations is the way TLM is modelled. We use regular Modelica models without function calls for communication between model elements. Nyström used an external function interface to do server-client communication. His method is a more explicit way of parallelisation, since he looks for the sub-models that the user created to achieve a kind of co-simulation. However, Nyström did not develop a fully
general solution to the TLM-based parallelisation for Modelica, the approach was only demonstrated for a very small electrical model.

It is also possible to use multi-core co-simulation where the master algorithm is able to schedule each task on a separate thread. This approach was used by xMOD [8], which imports and co-simulates FMUs [9]. Parallelism is introduced into the model by using TLM techniques. The difference from our approach is that it all seems to be done manually. One advantage of this approach is that since it can co-simulate FMUs, it is also possible to combine Modelica model components with models expressed in other languages.

Inlining solvers have also been used in the past to introduce parallelism in simulations, for example [45, 46].

Casella [16] proposed an algorithm that causes the compiler to utilise a certain structure common in some domains in order to parallelise them. It is yet unclear what the performance benefits can be expected.

Quantised state system (QSS) methods [18] belong to a different family of ODE solvers than the solvers used in this work. QSS algorithms are directly distributed and work asynchronously. Communication is done only at event times, and only with neighbouring integrators.

Parallelisation of Modelica-based simulation on GPUs has been explored by Stavåker [71] and Östlund [78].

5.1.2 Transmission Line Element Method

A computer simulation model is basically a representation of a system of equations that model some physical phenomena. The goal of simulation software is to solve this system of equations in an efficient, accurate and robust way. In order to achieve such a goal the by far most common approach is to use a centralised solver algorithm which puts all equations together into a DAE or an ODE. The system is then solved using matrix operations and numeric integration methods. One disadvantage of this approach is that it often introduces data dependencies between the central solver and the equation system, making it difficult to parallelise the equations for simulation on multi-core platforms. Another problem is that the stability of the numerical solver depends on the simulation step size.

An alternative approach is to let each component in the simulation model solve its own equations, that is, a distributed solver approach. This allows each component to have its own fixed time step in its solvers. A special case where this is especially suitable is the transmission line element method. Such a simulator has numerically highly robust properties and a high potential for taking advantage of multi-core platforms [41]. Despite these advantages, distributed solvers have never been widely adopted and centralised solvers have remained the de facto strategy on the simulation software market for DAE-based languages. One reason is perhaps the rapid increase in processor speed, which for many years has made multi-core systems unnecessary and reduced the priority of increasing simulation performance. The
other reason is that languages such as Modelica use implicit, centralised solvers since these solvers are very easy to use. Typically the simulation is robust without a single parameter being tweaked.

Modelling for multi-core-based simulation also requires applications of significant size in order for advantages to become significant. With the recent development towards an increase in the number of processor cores rather than an increase in speed of each core, distributed solvers will play a more important role in the near future. Another reason for the fact that distributed solvers have not yet been widely adopted is the limited tool support and the fact that the modeller has been required to manually perform a partitioning and decoupling of model parts.

The fundamental idea behind the TLM method is to model a system in a way such that components can be somewhat numerically isolated from each other. This allows each component to solve its own equations independently of the rest of the system. This is achieved by replacing capacitive components (for example volumes in hydraulic systems) with transmission line elements of a length for which the physical propagation time corresponds to at least one simulation time step. In this way a time delay is introduced between the resistive components (for example orifices in hydraulic systems). The result is a physically accurate description of wave propagation in the system [41]. The transmission line element method originates from the method of characteristics used in HYTRAN [1] and from Transmission Line Modelling [39], both developed back in the nineteen sixties [6]. Today it is used in the HOPSAN simulation package for fluid power and mechanical systems, see Section 5.1.3, and in the SKF TLM-based co-simulation package [63].

Mathematically, a transmission line can be described in the frequency domain by the four pole equation [77]. Assuming that friction can be neglected and transforming these equations to the time domain, they can be described according to equation 5.1 and 5.2.

\[
p_1(t) = p_2(t - T) + Z_c q_1(t) + Z_c q_2(t - T) \quad (5.1)
\]

\[
p_2(t) = p_1(t - T) + Z_c q_2(t) + Z_c q_1(t - T) \quad (5.2)
\]

Here \(p\) equals the pressure before and after the transmission line, \(q\) equals the volume flow and \(Z_c\) represents the characteristic impedance. The main property of these equations is the time delay they introduce, representing the communication delay between the ends of the transmission line, see Figure 5.1. In order to solve these equations explicitly, two auxiliary variables are introduced, see equations 5.3 and 5.4.

\[
c_1(t) = p_2(t - T) + Z_c q_2(t - T) \quad (5.3)
\]

\[
c_2(t) = p_1(t - T) + Z_c q_1(t - T) \quad (5.4)
\]

These variables are called wave variables or wave characteristics and they represent the delayed communication between the end nodes. Putting equations 5.1 to 5.4 together will yield the final relationships between flow and
pressure in equations 5.5 and 5.6.

\[ p_1(t) = c_1 + Z_c q_1(t) \]  \hspace{1cm} (5.5)

\[ p_2(t) = c_2 + Z_c q_2(t) \]  \hspace{1cm} (5.6)

These equations can now be solved using boundary conditions. These are provided by adjacent (resistive) components. In the same way, the resistive components get their boundary conditions from the transmission line (capacitive) components.

One noteworthy property with this method is that the time delay represents a physically correct separation in time between components of the model. Since the wave propagation speed (speed of sound) in a certain liquid can be calculated, the conclusion is that the physical length of the line is directly proportional to the delay time used when simulating the component, see equation 5.7. Note that this delay time is a parameter in the component and can very well differ from the time step used by the simulation engine. Keeping the delay in the transmission line larger than the simulation solver time step is important, to avoid extrapolation of delayed values. This means that a minimum time delay of the same size as the time step is required, introducing a modelling error for very short transmission lines.

\[ l = h a = \sqrt{\frac{\beta}{\rho}} \]  \hspace{1cm} (5.7)

Here, \( h \) represents the time delay and \( a \) the wave propagation speed, while \( \beta \) and \( \rho \) are the bulk modulus and the density of the liquid. With typical values for the latter two, the wave propagation speed will be approximately 1000 m/s, which means that a time delay of 1 ms will represent a length of 1 m. [42]

It should be noted that by using TLM, the fixed delay time corresponds to using explicit solvers in a simulation tool. Some state variables have been implicitly inlined and eliminated. While it is possible to use implicit solvers, such as DASSL, within the submodels between the transmission lines, the method is more suitable for use with explicit solvers. TLM is especially suitable for real-time simulation since explicit solvers are easier to predict than implicit ones.
5.1.3 Hopsan

HOPSAN is a simulation software for simulation and optimization of fluid power and mechanical systems. This software was first developed at Linköping University in the late 1970’s [24]. The simulation engine is based on the transmission line element method described in Section 5.1.2, with transmission lines (called C-type components) and restrictive components (called Q-type) [44]. In the current version, the solver algorithms are distributed so that each component uses its own local solvers, although many common algorithms are placed in centralised libraries.

In the new version of HOPSAN, which is currently under development, all equation solvers will be completely distributed as a result of an object-oriented programming approach [7]. Numerical algorithms in HOPSAN are always discrete. Derivatives are implemented by first or second order filters, that is, a low-order rational polynomial expression as approximation, and using bi-linear transforms, that is, the trapezoid rule, for numerical integration. Support for built-in compatibility between HOPSAN and Modelica is also being investigated. Performance measurements of the new HOPSAN implementation can be found in [11].

5.1.4 Modelica

Modelica supports event handling as well as delayed expressions in equations. We will use these properties later in our implementation of a distributed TLM-style solver. It is worth mentioning that HOPSAN may access the value of a state variable, for example $x$, from the previous time step. This value may then be used to calculate derivatives or do filtering since the length of time steps is fixed.

In standard Modelica it is possible to access the previous value before an event using the `pre()` operator, but impossible to access solver time-step related values, since a Modelica model is independent of the choice of solver. This is where sampling and delaying expressions comes into play.

In particular, this work depends on the Modelica compiler to perform some optimisations regarding the `delay()` operator. Without special treatment of the operator, data can be extrapolated from the two previous values, which is a poor approximation of the actual value. The more accurate way of calculating the delayed value is by interpolation of the current and previous value. If it is hard to understand why this is true, consider the expression `delay(x,0)` – a delay-time of zero is obviously the expression itself. The dependence on the expression itself may forms an algebraic loop that makes simulation very slow and in many cases the numerical solution does not converge.

What is needed for this work is for the `delay()` operator to break algebraic loops. In order to achieve this, the step size of the integrator is forced to be lower than the minimum delay-time of all delay-expressions in the system. Compare this to the restriction in (section 5.1.2), where it is stated
that the delay time is at least as large as the time step. This is no coinci-
dence and the only real restriction of this method is that the Modelica tool
used must perform this optimisation, which is optional in the specification
[52].

5.2 Example Model with Pressure Relief Valve

The example model used here for comparing TLM implementations is a
simple hydraulic system consisting of a volume with a pressure relief valve,
as can be seen in Figure 5.2. A pressure relief valve is a safety component,
with a spring at one end of the spool and the upstream pressure, that is, the
pressure at the side of the component where the flow is into the component,
acting on the other end, see Figure 5.3. The preload of the spring will make
sure that the valve is closed until the upstream pressure reaches a certain
level, when the force from the pressure exceeds that of the spring. The valve
then opens, reducing the pressure to protect the system.

In this system the boundary conditions are given by a constant prescribed
flow source into the volume and a constant pressure source at the other end
of the pressure relief valve representing the tank. As oil flows into the volume
the pressure will increase at a constant rate until the reference pressure of
the relief valve is reached. The valve then opens and after some oscillations
(transitory state), a steady state pressure level will appear.

A pressure relief valve is a very suitable example model when comparing
simulation tools. The reason is that it is based on dynamic equations and
also includes several non-linearities, making it an interesting component to
study. It also includes multiple physical domains, namely hydraulics and
mechanics. The opening of a relief valve can be represented as a step or
ramp response, which can be analysed by frequency analysis techniques,
for example using bode plots or Fourier transforms. It also includes sev-
eral physical phenomena useful for comparisons, such as wave propagation,
damping and self oscillations. If the complete set of equations is used, it
will also produce non-linear phenomena such as cavitation and hysteresis,
although these are not included here.
5.2. EXAMPLE MODEL WITH PRESSURE RELIEF VALVE

The volume is modelled as a transmission line, in HOPSAN known as a C-type component. In practice this means that it will receive values for pressure and flow from its neighbouring components (flow source and pressure relief valve) and return characteristic variables and impedance. The impedance is calculated from bulk modulus, volume and time step, and is in turn used to calculate the characteristic variables together with pressures and flows. There is also a low-pass damping coefficient called $\alpha$, which is set to zero and thereby not used in this example.

$$mZc = \frac{mBulkmodulus}{mVolume} * mTimestep;$$
$$c10 = p2 + mZc * q2;$$
$$c20 = p1 + mZc * q1;$$
$$c1 = mAlpha * c1 + (1.0 - mAlpha) * c10;$$
$$c2 = mAlpha * c2 + (1.0 - mAlpha) * c20;$$

The pressure relief valve is a restrictive component, known as Q-type, which means that it receives characteristic variables and impedance from its neighbouring components and returns flow and pressure. Advanced models of pressure relief valves are normally performance oriented. This means that parameters that a user normally has little or no knowledge about, such as the inertia of the spool or the stiffness of the spring are not needed as input parameters but are instead implicitly included in the model. Such a way of handling parameters is however complicated and not very intuitive, thus a simpler model was created for this example. It is basically a first-order force equilibrium equation with a mass, a spring and a force from the pressure. Hysteresis and cavitation phenomena are also excluded from the model.

The first three equations below calculate the total force acting on the spool. By using a second-order filter, the x position can be received from Newton’s second law. The position is used to retrieve the flow coefficient of the valve, which in turn is used to calculate the flow using a turbulent flow algorithm. Pressure can then be calculated from impedance and characteristic variables according to transmission line modelling.
\[
m_{Fs} = m_{PilotArea} \times m_{Pref}; \\
p_1 = c_1 + q_1 \times Zc_1; \\
F_{tot} = p_1 \times m_{PilotArea} - m_{Fs}; \\
x_0 = m_{Filter}.value(F_{tot}); \\
m_{Turb}.setFlowCoefficient(m_{Cq} \times m_{W} \times x_0); \\
q_2 = m_{Turb}.getFlow(c_1, c_2, Zc_1, Zc_2); \\
q_1 = -q_2; \\
p_1 = c_1 + Zc_1 \times q_1; \\
p_2 = c_2 + Zc_2 \times q_2;
\]

5.3 TLM in an Equation-Based Language

There are some issues when trying to use transmission lines in an equation-based language.

TLM has been proven to work well with numerical solvers using fixed time steps. In Modelica however, events can happen at any time. When an event is triggered due to an event-inducing expression changing sign (zero-crossing), the continuous-time solver is temporarily stopped and a root-finding solution process is started in order to find the point in time where the event occurs. If the event occurs for example in the middle of a fixed time step, the solver will need to take a smaller time step when restarted, that is, some solvers may take extra time steps if the specified tolerance is not reached. However, this occurs only for hybrid models. For pure continuous-time models which do not induce events, fixed steps will be kept when using a fixed step solver.

The delay in the transmission line can be implemented in several ways. One possibility is to implement a delayed expression as a periodically sampled expression. Sampling works fine in Modelica, but requires an efficient Modelica tool since the system is typically sampled quite frequently. An example usage of the Modelica \texttt{sample()} built-in function is shown below. Variables defined within when-equations in Modelica as below will have discrete-time variability.

\[
\begin{array}{l}
\text{when sample}(-T,T) \text{ then} \\
\quad \text{left} . c = \text{pre(right} . c) + 2 \times Zc \times \text{pre(right} . q); \\
\quad \text{right} . c = \text{pre(left} . c) + 2 \times Zc \times \text{pre(left} . q); \\
\text{end when;}
\end{array}
\]

Modelica tools also offer the possibility to use delays instead of sampling. If delays are used, this gives rise to continuous-time variables instead of discrete-time ones. The methods are numerically very similar, but because the variables are continuous-time when delays are used, the curve will look smoother than when the sampled system is plotted with the discrete signal as a staircase.
5.3. TLM IN AN EQUATION-BASED LANGUAGE

Figure 5.4: Pressure increases until the reference pressure of 10 MPa is reached, where the relief valve opens.

```
left.c = delay(right.c + 2 * Zc * right.q, T);
right.c = delay(left.c + 2 * Zc * left.q, T);
```

Finally, it is possible to explicitly specify a derivative rather than obtaining it implicitly by difference computations relating to previous values (delays or sampling). This then becomes a transmission line without delay, which is a good reference system.

```
der(left.p) = (left.q + right.q)/C;
der(right.p) = der(left.p);
```

Figure 5.4 contains the results of simulating our example system, that is, the pressure relief valve from section 5.2. Figures 5.5 and 5.6 are magnified versions that show the difference between our different TLM implementations, using the delay, sample, or der operators in Modelica. The models used to create the Figures, are part of the Modelica package in Appendix B, with the problemSize parameter set to 1.

Note how similar the curves of the delay implementation and the ideal der operator are. It looks like × and + are combined into *. They are basically the same and a step size of $10^{-4}$ was used, which is the longest step size allowed without extrapolation. The curve that does not fit quite as well as the others is the HOPSAN implementation, which does not use the Modelica model at all. Instead it uses custom C++ code which uses other integrators and approximations.

To verify that different Modelica tools produce the same results, a comparison was made between Dymola [21] and OpenModelica (Figure 5.7). It shows that both OpenModelica and Dymola give almost exactly the same
Figure 5.5: Comparison of spool position using different TLM implementations. The curves for delay, der, and sample overlaps almost completely.

Figure 5.6: Comparison of system pressure using different TLM implementations. The curves for delay, der, and sample overlaps almost completely.
5.3. TLM IN AN EQUATION-BASED LANGUAGE

Figure 5.7: Comparison of spool position in OpenModelica and Dymola. The curves overlap almost completely. The markers are in different positions because OpenModelica does not round or interpolate the output points in the result-file.

results. Comparisons of the delay and sample operators are not provided because OpenModelica performs optimisations essential for models written using TLM. Dymola can not simulate the version that uses the delay operator due to chattering when the pressure of $10^7$ is reached. The tool also had trouble detecting all time-events with the sample operator which gives unexpected results.

The OpenModelica DASSL solver cannot be used in all of these models due to an incompatibility with the delay() operator since the solver does not limit its step size as it should. For this reason, the RK4\(^1\) solver was used throughout since it provides better numerical stability than Euler.

Table 5.1 contains average performance numbers on the models used based on 20 runs. Only one segment was used for the pipe and the simulation did not write a result-file which means the simulations run really fast. The only real surprise here is that sampling in OpenModelica improved in performance by a few orders of magnitude since [67]. This is due to time-events being implemented in the last few years in OpenModelica which means exact times for events can be found without searching for it in the zero-crossing function. While sampling is now rather fast, at least for small systems, the OpenModelica prototype cannot parallelise a sampled system since our parallelisation scheme does not consider discrete events yet.

We have only made single-core simulations so far for a small problem size that cannot really be parallelised.

\(^1\)Explicit Runge-Kutta solver of order 4, with root finding of zero-crossings.
Table 5.1: Performance comparison of methods to model TLM in Modelica

<table>
<thead>
<tr>
<th>Method</th>
<th>Min (s)</th>
<th>Mean (s)</th>
<th>Max (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delay</td>
<td>0.02280</td>
<td>0.03010</td>
<td>0.03750</td>
</tr>
<tr>
<td>Ideal</td>
<td>0.1260</td>
<td>0.1320</td>
<td>0.1410</td>
</tr>
<tr>
<td>Sample</td>
<td>0.08630</td>
<td>0.09312</td>
<td>0.10600</td>
</tr>
</tbody>
</table>

\[
\begin{bmatrix}
\ast & \ast & \ast \\
\ast & X & \ast \\
\ast & \ast & \ast
\end{bmatrix}
\begin{bmatrix}
\ast & \ast & \ast \\
\ast & \ast & \ast \\
\ast & \ast & \ast
\end{bmatrix}
\begin{bmatrix}
\ast & \ast & \ast \\
\ast & \ast & \ast \\
\ast & \ast & \ast
\end{bmatrix}
\]

(a) Regular system (b) TLM system (c) Partitioned systems

Figure 5.8: Adjacency matrices in lower triangular form.

The current OpenModelica simulation runtime system implementation does not have special efficient implementations for delayed expressions but is still very fast compared to the alternatives.

The `-noemit` flag in OpenModelica to disable generation of result files. Generating them takes between 20% and 90% of the total simulation runtime depending on solver and how many events are generated. Especially the sampled implementation benefits from not generating a result file since the simulation creates extra output points before and after each event.

5.4 Partitioning

The main reason for using TLM is that a coarse-grained parallelisation of the system is implicitly gained. We present a general approach to partitioning a system of equations that utilises the time-delay introduced by TLM.

Each partition of the equation system will be independent from any other within the current time step. This means they can be parallelised by synchronising between time steps.

To illustrate what our algorithm does, consider the standard approach to causalise an equation system (Section 2.2.2). The equation system is transformed into block lower triangle (BLT) form, where each BLT block corresponds to either a single equation from the original system of equations or a strongly connected component (several equations) in the strongly connected components dependency graph of the system of equations. This means that the system can be solved sequentially.

What happens when TLM is used to model the system is that some entries in the adjacency matrix disappear since delay expressions are allowed to decouple the system if they only access data in former time steps. In
Figure 5.8b, the entry using $X$ has been removed from Figure 5.8a by using a TLM delay. The new matrix can be cut into two separate ones and solved sequentially or in parallel since the two blocks are now totally independent from each other (Figure 5.8c).

5.4.1 Algorithm

The basic data structure needed to perform the partitioning analysis is the adjacency matrix. Neither the BLT matrix nor the sorted system are needed since it is possible to use the adjacency matrix alone to determine if two equations are totally connected in the graph. The benefit of only looking at the adjacency matrix is that the equation system can be partitioned before optimisations are performed, some of which are costly to perform on large systems since they do not have linear time complexity.

Since not all nodes are connected to each other, the graph is not a tree, but a forest. The goal is to find all trees in the forest since these are possible to run in parallel. There are many ways to do this operation fast. Which one to choose depends mostly on the data representation that is used. Cormen [20] contains an interesting algorithm using disjoint sets. Our approach instead uses a depth-first search, marking all reachable nodes then choosing the next unmarked vertex and repeating the algorithm until all trees have been found. The algorithm has a complexity of $O(|V| + |E|)$, as any connected components algorithm should.

5.5 Distributed Solver

The partitioned system can also be handled using distributed solvers. It may then use different time steps in the different partitions, which means a system can be simulated using a very small time step only for certain components. The advantage of such a distributed system becomes apparent in [42].

In the current OpenModelica implementation this is not yet taken advantage of since a global solver is always used. It is also needed to specify which solvers each partition should use. Modelica 3.3 [52] introduced synchronous language features and annotations that can be used to select solvers and step sizes for the newly introduced clocked partitions. It is not yet supported in OpenModelica, and only with limited support (beta) in Dymola [21].

5.6 Runtime

To generate parallel code for an equation system with TLM connections is very simple. As we have already shown, the partitioning algorithm works in linear time and produces the sets of equation to be run in parallel. The OpenModelica Compiler currently provides a very simple code generation
for the partitioned systems of equations. The generated code has the general structure shown in Listing 5.1. The serial code is provided as a reference point (Listing 5.2). We can generate code for both dynamic and static scheduling. With static scheduling, OpenModelica determines the estimated execution time of each system and assigns it to a core using a greedy scheduling algorithm. The execution time estimates are too simplistic to make static scheduling a real choice using our current compiler prototype.

To test the partitioning, this simplistic code generator was used and a version of the pressure relief valve with a tunable number of volumes and orifices to simulate. This is basically breaking down one pipe into multiple segments in order to test the parallelisation.

The results of 20 runs with the volume split into 20 segments (19 orifices) using an RK4 integrator and a step size of $5 \times 10^{-6}$ are shown in Table 5.2.
Table 5.2: Performance of parallelisation

<table>
<thead>
<tr>
<th></th>
<th>Min (s)</th>
<th>Mean (s)</th>
<th>Max (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>serial</td>
<td>5.380</td>
<td>5.593</td>
<td>5.770</td>
</tr>
<tr>
<td>dynamic</td>
<td>4.130</td>
<td>4.582</td>
<td>6.660</td>
</tr>
<tr>
<td>static 2 threads</td>
<td>8.220</td>
<td>9.256</td>
<td>12.200</td>
</tr>
<tr>
<td>static 3 threads</td>
<td>6.320</td>
<td>7.138</td>
<td>8.400</td>
</tr>
<tr>
<td>static 4 threads</td>
<td>5.410</td>
<td>5.912</td>
<td>6.550</td>
</tr>
<tr>
<td>static 5 threads</td>
<td>4.690</td>
<td>5.178</td>
<td>5.610</td>
</tr>
<tr>
<td>static 6 threads</td>
<td>4.720</td>
<td>5.128</td>
<td>5.950</td>
</tr>
<tr>
<td>static 7 threads</td>
<td>4.130</td>
<td>4.481</td>
<td>5.200</td>
</tr>
<tr>
<td>static 8 threads</td>
<td>4.200</td>
<td>4.600</td>
<td>5.130</td>
</tr>
</tbody>
</table>

This is equivalent to one volume with a step size of $10^{-4}$, which was used earlier. Figures 5.9 and 5.10 show a comparison of the volume modelled using one or twenty segments. The results are the same, with the main difference being that the volume split into more segments being less smooth. There exists a small numerical error that is probably amplified due to the large number of segments the pipe was split into. Simulation results from running the parallel tests can be verified by producing a result-file using a serial implementation and comparing the files using the `compareSimulationResults()` API in OpenModelica. These tests gave the exact same results.

A four-core i7-3820 was used for performance measurements using gcc 4.4 on Linux. Some speedup was obtained with static scheduling only if hyper-threading was used. The reason is that the rough estimates of execution time are quite a bit off from the real world. Dynamic scheduling works as well as static scheduling, but neither was very much faster than the serial implementation. Part of the reason is that the model was not split into very many different parts. Scaling this particular model into more parts involves cutting the volumes into more parts, which shortens the delay line and forces us to use a smaller step size. This causes a quadratic growth of simulation time, which in turn limits the maximum size that can be used while still obtaining results in a reasonable time frame. Twenty segments was a small enough size that still shows the model does scale with more cores was used.

While good performance is our target, the goal so far has been to prove that the generated simulation code behaves as expected and produces correct results. Note that while in the best case, minimum time, parallelisation does become faster and faster. But it also becomes more unpredictable if too many threads are used as can be seen for the case of dynamic scheduling. It is always good to keep one thread open for other tasks in the operating system since there is always something going on. Dynamic scheduling is only on par with serial code in the worst case even though it performs really well in the best and average case.

The chosen example is a simple one for a good reason: the OpenModelica
CHAPTER 5. PARALLEL SIMULATION

Figure 5.9: Comparison of spool position using a volume split into more segments.

Figure 5.10: Comparison of system pressure using a volume split into more segments.
runtime code is not thread-safe in the general case. The OpenModelica Compiler backend has optimised the model far enough that no memory allocation, like creating a temporary array, needs to be performed during a time-step. Since the code works as expected, we can proceed with improving the performance of the generated code in addition to making the runtime system thread-safe.

5.7 Future Work

To make performance even faster we could either optimise the usage of the delay() operator or extend Modelica to allow access to the current step size and value of a variable in the previous time step. This should be faster than using the delay() operator as well as improve stability.

Since our current prototype partitions the Modelica model we would be able to simulate subsystems using distributed solvers with different step sizes [38, 42], also referred to as multi-rate integration [17, 43]. This requires rewriting the main loop in the simulator runtime system to handle different solvers and step sizes in addition to moving logic for parallel systems to the runtime system instead of being in the model itself.

Small tasks need to be merged to avoid synchronisation overhead. We also need to rewrite the runtime system to be totally thread-safe so that more complex models and alternative parallelisation schemes can be tested. Such parallelisation schemes include evaluating the method proposed by Casella [16], as well as parallelising implicit numerical solvers.

Parallelising at the solver level may be useful in the general case since implicit solvers call $f(x)$ many times for one step. It would also be very easy enable as a user. The problem with this approach is that it increases the sizes of work arrays, which reduces locality and may cause more cache misses. Another disadvantage is that only the calculation of states is parallelised – dependent variables are calculated after the states and would then still be calculated in serial unless methods were combined.

It would be interesting to use clocked partitions in Modelica 3.3 [52] instead of delays to model TLM. These language elements were introduced with real-time systems in mind and support use of different solvers at fixed steps. Support for the synchronous language features introduced in Modelica 3.3 would then first need to be implemented in OpenModelica.

There is a need to write more Modelica models using TLM. Currently we provide only one model from the Fluid domain. While this model was chosen because it contains all the necessary elements for modelling using TLM, more advanced models are needed to test parallelisation.
5.8 Conclusions

We conclude that all implementations (using der(), sampling, and delay()) of our test model are accurate in Modelica. The delay() implementation is fast enough that using it will not cancel out any benefits from parallelising the model.

The code can be scaled up to demonstrate that the parallelisation algorithm works fine. Code generation for OpenMP is working, but does not yet provide good speed-up.
Chapter 6

Conclusion

Debug information can be collected without much overhead. Using this information it is possible to produce a machine-readable XML-file as well as a human-readable HTML-file with a visual diff that makes it possible to use it for manual debugging. There is still a problem with this approach, that is, too much of the work is still done by the user. The reason is that there is not yet any smart enough tool that knows enough about the structure of equation-based models to guide the user. We plan to provide an improved version of our current simple tool in the future, which will have uses beyond merely debugging a user model. One such use is profiling simulations of models for real-time targets coupled to mapping the source positions of problematic parts of the generated simulation code to the original model.

For real-time, hardware-in-the-loop, we provide an implementation of a profiler. That is, we can measure the CPU-time that certain blocks of equations and function calls take to execute. This enables us to find out which equations are worthwhile to simplify in order to speed up simulation. This also gives us a possibility to identify subsystems suitable for parallelisation.

By decoupling two parts of a model using a bilateral delay line, it is possible to simulate these parts of the model in parallel. Moreover, if this change is physically motivated, correctness of the simulation result is not sacrificed. We have tested a model that uses delay lines and can be scaled to increase the computational load. The algorithm we provide coupled with a simple modification of generated code produces an executable that gets a slight speedup compared to serial code.

6.1 Future Work

So far we have only broached the subject. There is a lot more to be done especially for debugging equation-based models. In all the future work listed in the previous chapters, the most pressing matter is producing a graphical debugger. The reason is that the intended workflow very much depends on
this tool becoming a reality.

Of course there are more things to be done especially considering things that are missing in our OpenModelica prototypes. Integration of the debugger and profiler is one goal. Improved code generation and runtime system for parallel code is another. Future work for the OpenModelica Compiler is by no means limited to work done in this thesis. General improvements to the compiler are of great interest especially for parallelisation part. The reason is that serial performance also impacts parallel performance. Further improvements will also add support for more Modelica libraries and thus more models that we can use as a basis for debugging and parallelisation.

There is currently no special support to do profiling on parallel simulations. Thus, an obvious extension would be to include thread activity over time, average waiting times for synchronisation, and more.
Flat Modelica is type-checked Modelica code where the object-oriented parts have been flattened out and connections have been expanded. The following is the flattened RC circuit from Chapter 2

Listing A.1: Flattened RC Circuit

```modelica
class RC
    Real c.v (quantity = "ElectricPotential", unit = "V", start = 0.0) "Voltage drop between the two pins (= p.v - n.v)";
    Real c.i (quantity = "ElectricCurrent", unit = "A") "Current flowing from pin p to pin n";
    Real c.p.v (quantity = "ElectricPotential", unit = "V") "Potential at the pin";
    Real c.p.i (quantity = "ElectricCurrent", unit = "A") "Potential at the pin";
    Real c.n.v (quantity = "ElectricPotential", unit = "V") "Potential at the pin";
    Real c.n.i (quantity = "ElectricCurrent", unit = "A") "Current flowing into the pin";
    parameter Real c.C (quantity = "Capacitance", unit = "F", min = 0.0, start = 1.0) = 0.000001 "Capacitance";
    Real r.v (quantity = "ElectricPotential", unit = "V") "Voltage drop between the two pins (= p.v - n.v)";
    Real r.i (quantity = "ElectricCurrent", unit = "A") "Current flowing from pin p to pin n";
    Real r.p.v (quantity = "ElectricPotential", unit = "V") "Potential at the pin";
    Real r.p.i (quantity = "ElectricCurrent", unit = "A") "Current flowing into the pin";
```

73
Real r.n.v(quantity = "ElectricPotential", unit = "V") "Potential at the pin";
Real r.n.i(quantity = "ElectricCurrent", unit = "A") "Current flowing into the pin";
parameter Boolean r.useHeatPort = false "=true, if HeatPort is enabled";
Real r.LossPower(quantity = "Power", unit = "W") "Loss power leaving component via HeatPort";
Real r.T_heatPort(quantity = "ThermodynamicTemperature", unit = "K",
displayUnit = "degC", min = 0.0, start = 288.15, nominal = 300.0) "Temperature of HeatPort";
parameter Real r.R(quantity = "Resistance", unit = "Ohm", start = 1.0) = 1000000.0 "Resistance at temperature T_ref";
parameter Real r.T_ref(quantity = "ThermodynamicTemperature", unit = "K",
displayUnit = "degC", min = 0.0, start = 288.15, nominal = 300.0) = 300.15 "Reference temperature";
parameter Real r.alpha(quantity = "LinearTemperatureCoefficient", unit = "1/K") = 0.0 "Temperature coefficient of resistance (R_actual = R*(1 + alpha*(T_heatPort - T_ref))"");
Real r.R_actual(quantity = "Resistance", unit = "Ohm") "Actual resistance = R*(1 + alpha*(T_heatPort - T_ref))";
parameter Real r.T(quantity = "ThermodynamicTemperature", unit = "K",
displayUnit = "degC", min = 0.0, start = 288.15, nominal = 300.0) = r.T_ref "Fixed device temperature if useHeatPort = false";
Real g.p.v(quantity = "ElectricPotential", unit = "V") "Potential at the pin";
Real g.p.i(quantity = "ElectricCurrent", unit = "A") "Current flowing into the pin";
Real pv.v(quantity = "ElectricPotential", unit = "V") "Voltage drop between the two pins (= p.v - n.v)";
Real pv.i(quantity = "ElectricCurrent", unit = "A") "Current flowing from pin p to pin n";
Real pv.p.v(quantity = "ElectricPotential", unit = "V") "Potential at the pin";
Real pv.p.i(quantity = "ElectricCurrent", unit = "A") "Current flowing into the pin";
Real pv.n.v(quantity = "ElectricPotential", unit = "V") "Potential at the pin";
Real pv.n.i(quantity = "ElectricCurrent", unit = "A") "Current flowing into the pin";
parameter Real pv.offset(quantity = "ElectricPotential", unit = "V") = 0.0 "Voltage offset";
parameter Real pv.startTime(quantity = "Time", unit = "s") = 0.0 "Time offset";
parameter Real pv.V(quantity = "ElectricPotential", unit = "V", start = 1.0) = 240.0 "Amplitude of pulse";
parameter Real pv.width(min = 1e-60, max = 100.0, start = 50.0) "Width of pulse in % of period";
parameter Real pv.period(quantity = "Time", unit = "s", min = 1e-60, start = 1.0) = pv.period "Time for one period";
output Real pv.signalSource.y "Connector of Real output signal";
parameter Real pv.signalSource.amplitude = pv.V "Amplitude of pulse";
parameter Real pv.signalSource.width(min = 1e-60, max = 100.0) = pv.width "Width of pulse in % of period";
parameter Real pv.signalSource.period(quantity = "Time", unit = "s", min = 1e-60, start = 1.0) = pv.period "Time for one period";
parameter Integer pv.signalSource.nperiod = -1 "Number of periods (< 0 means infinite number of periods)";
parameter Real pv.signalSource.offset = pv.offset "Offset of output signals";
parameter Real pv.signalSource.startTime(quantity = "Time", unit = "s") = pv.startTime "Output = offset for time < startTime";
protected Real pv.signalSource.T_width(quantity = "Time", unit = "s") = pv.signalSource.period * pv.signalSource.width / 100.0;
protected Real pv.signalSource.T_start(quantity = "Time", unit = "s") "Start time of current period";
protected Integer pv.signalSource.count "Period count";

initial algorithm
pv.signalSource.count := integer((time -
   pv.signalSource.startTime) /
   pv.signalSource.period);

pv.signalSource.T_start :=
   pv.signalSource.startTime + /*Real*/
   (pv.signalSource.count) *
   pv.signalSource.period;

equation
   c.i = c.C * der(c.v);
   c.v = c.p.v - c.n.v;
   0.0 = c.p.i + c.n.i;
   c.i = c.p.i;
   assert(1.0 + r.alpha * (r.T_heatPort - r.T_ref)
      >= 0.000000000000001,"Temperature outside
      scope of model!");

   r.R_actual = r.R * (1.0 + r.alpha *
      (r.T_heatPort - r.T_ref));

   r.v = r.R_actual * r.i;
   r.LossPower = r.v * r.i;
   r.v = r.p.v - r.n.v;
   0.0 = r.p.i + r.n.i;
   r.i = r.p.i;
   r.T_heatPort = r.T;

   g.p.v = 0.0;

when integer((time - pv.signalSource.startTime)
   / pv.signalSource.period) > pre(
   pv.signalSource.count) then
pv.signalSource.count = 1 + pre(
   pv.signalSource.count);
pv.signalSource.T_start = time;
end when;

pv.signalSource.y = pv.signalSource.offset + (if
   time < pv.signalSource.startTime or
   pv.signalSource.nperiod == 0 or
   pv.signalSource.nperiod > 0 and
   pv.signalSource.count >=
   pv.signalSource.nperiod then 0.0 else if time
   < pv.signalSource.T_start +
   pv.signalSource.T_width then
   pv.signalSource.amplitude else 0.0);

pv.v = pv.signalSource.y;

pv.v = pv.p.v - pv.n.v;
0.0 = pv.p.i + pv.n.i;
pv.i = pv.p.i;
c.p.i + r.n.i = 0.0;
c.n.i + g.p.i + pv.n.i = 0.0;
r.p.i + pv.p.i = 0.0;
c.n.v = g.p.v;
c.n.v = pv.n.v;
pv.p.v = r.p.v;
c.p.v = r.n.v;
end RC;
Appendix B

TLM Example

The following is the example model used in Chapter 5\(^1\).

Listing B.1: TLM Example

```plaintext
package TLM

connector Connector_Q
    output Real p;
    output Real q;
    input Real c;
    input Real Zc;
end Connector_Q;

connector Connector_C
    input Real p;
    input Real q;
    output Real c;
    output Real Zc;
end Connector_C;

model FlowSource
    Connector_Q source;
    parameter Real flowVal;
    equation
        source.q = flowVal;
        source.p = source.c + source.q*source.Zc;
end FlowSource;

model PressureSource
```

\(^1\)The latest version can currently be found in the OpenModelica sources. Since it is prone to move, I’ll leave it as an exercise to the reader to find it.
Connector_C pressure;
  parameter Real P;
end PressureSource;

model HydraulicAlternativePRV
  Connector_Q left;
  Connector_Q right;

  parameter Real Pref = 20e6 "Reference Opening Pressure";
  parameter Real cq = 0.67 "Flow Coefficient";
  parameter Real spooldiameter = 0.01 "Spool Diameter";
  parameter Real frac = 1.0 "Fraction of Spool Circumference that is Opening";
  parameter Real W = spooldiameter*frac;
  parameter Real pilotarea = 0.001 "Working Area of Pilot Pressure";
  parameter Real k = 1e6 "Steady State Characteristics of Spring";
  parameter Real c = 1000 "Steady State Damping Coefficient";
  parameter Real m = 0.01 "Mass";
  parameter Real xhyst = 0.0 "Hysteresis of Spool Position";
  constant Real xmax = 0.001 "Maximum Spool Position";
  constant Real xmin = 0 "Minimum Spool Position";

  parameter Real T;
  parameter Real Fs = pilotarea*Pref;

  Real Ftot = left.p*pilotarea - Fs;
  Real Ks = cq*W*x;
  Real x(start = xmin, min = xmin, max = xmax);

  Real xfrac = x*Pref/xmax;
  Real v = (x-delay(x,T))/T "better than der(xtmp) and der(x) does not have a good derivative";
  Real a = (v-delay(v,T))/T "der(v)";
  Real v2 = c*v;
  Real x2 = k*x;
Real xtmp;

equation
  left.p = left.c + left.Zc*left.q;
  right.p = right.c + right.Zc*right.q;

  left.q = -right.q;
  right.q = sign(left.c-right.c) * Ks * (noEvent(
    sqrt(abs(left.c-right.c)+((left.Zc+right.Zc)*
    Ks)^2/4)) - Ks*(left.Zc+right.Zc)/2);

  xtmp = (Ftot - c*v - m*a)/k;
  x = if noEvent(xtmp < xmin) then xmin else if
      noEvent(xtmp > xmax) then xmax else xtmp;
end HydraulicAlternativePRV;

replaceable model Volume
  parameter Real V;
  parameter Real Be;
  final parameter Real Zc = Be*T/V;
  parameter Real T;

  Connector_C left;
  Connector_C right;

equation
  left.Zc = Zc;
  right.Zc = Zc;
  left.c = if initial() then 0 else delay(right.c
    +2*Zc*right.q,T);
  right.c = if initial() then 0 else delay(left.c
    +2*Zc*left.q,T);
end Volume;

model VolumeDer
  parameter Real V;
  parameter Real Be;
  parameter Real Zc = Be*T/V;
  parameter Real T;
  parameter Real C = V/Be;

  Connector_C left(Zc = Zc);
  Connector_C right(Zc = Zc);
protected
  Real derleftp;

equation
  derleftp = (left.q+right.q)/C;

derleftp = \text{der}(\text{left}.p);
derleftp = \text{der}(\text{right}.p);
end VolumeDer;

model VolumeSample
parameter Real V;
parameter Real Be;
parameter Real Zc = Be*T/V;
parameter Real T = 0.01;

Connector\_C left(Zc = Zc);
Connector\_C right(Zc = Zc);
equation
when sample(-T,T) then
  left.c = pre(right.c) + 2*Zc*pre(right.q);
  right.c = pre(left.c) + 2*Zc*pre(left.q);
end when;
end VolumeSample;

model Orifice
parameter Real K;

Connector\_Q left;
Connector\_Q right;
equation
  left.q = (right.p - left.p)*K;
  right.q = -left.q;
  left.p = left.c + left.Zc*left.q;
  right.p = right.c + right.Zc*right.q;
end Orifice;

end TLM;

model PRVSystem
extends TLM;
parameter Real T = 1e-4;
// Chain of volumes + orifices to scale up the problem size
parameter Integer problemSize = 24;
Volume volumes[problemSize](each final V=1e-3 / problemSize, each final Be=1e9, each final T=T/problemSize);
Orifice orifices[problemSize-1](each final K=1);
FlowSource flowSource(flowVal = 1e-5);
PressureSource pressureSource(P = 1e5);
HydraulicAlternativePRV hydr(Pref=1e7,cq=0.67,
spooldiameter=0.0025,frac=1.0,pilotarea=5
×e-5,xmax=0.015,m=0.12,c=400,k=150000,T=T);

equation
connect(flowSource.source,volumes[1].left);
for i in 1:(problemSize-1) loop
  connect(volumes[i].right,orifices[i].left);
  connect(orifices[i].right,volumes[i+1].left);
end for;
connect(volumes[problemSize].right,hydr.left);
connect(hydr.right,pressureSource.pressure);
end PRVSystem;

model PRVSystemDer "Using der() instead of delay()"
  extends PRVSystem(redeclare model Volume =
  TLM.VolumeDer);
end PRVSystemDer;

model PRVSystemSample "Using sample instead of
delay()"
  extends PRVSystem(redeclare model Volume =
  TLM.VolumeSample);
end PRVSystemSample;
References


86

REFERENCES


REFERENCES


REFERENCES


Department of Computer and Information Science
Linköpings universitet

Licentiate Theses

Linköpings Studies in Science and Technology
Faculty of Arts and Sciences

No 17
Vojin Plavsic: Interleaved Processing of Non-Numerical Data Stored on a Cyclic Memory. (Available at: FOA, Box 1165, S-581 11 Linköping, Sweden. FOA Report B30062E)

No 28

No 29

No 48

No 52

No 60

No 71

No 72

No 73
Ola Strömberg: A Structure Editor for Documents and Programs, 1986.

No 74

No 104

No 109

No 111

No 113

No 118

No 126

No 127

No 139

No 140

No 146

No 150

No 166

No 174

No 177
Peter Åberg: Design of a Multiple View Presentation and Interaction Manager, 1989.

No 230

No 248

No 253

No 272

No 328

No 332

No 335

No 348

No 352

No 371

No 378
No 380 Johan Ringström: Compiler Generation for Parallel Languages from Denotational Specifications, 1993.
No 414 Peter Carlsson: Separation av företagsledning och finansiering - fallstudier av företagsledarutkört ur ett agent-
FHS 3/94 Owen Eriksson: Informationssystem med verksamhetskvalitet - utvärdering baserat på ett verksamhetsinriktat och
FHS 4/94 Karin Pettersson: Informationsystemstrukturering, ansvarsfördelning och användarinflytande - En komparativ
studie med utgångspunkt i två informationssystemstrategier, 1994.
No 455 Pär Carlshamre: A Collaborative Approach to Usability Engineering: Technical Communicators and System
FHS 5/94 Stefan Cronholm: Varför CASE-verktyg i systemutveckling? - En motiv- och konsekvensstudie avseende
No 482 Eva L. Ragnemalm: Towards Student Modelling through Collaborative Dialogue with a Learning Companion,
1995.
No 488 Eva Toller: Contributions to Parallel Multiparadigm Languages: Combining Object-Oriented and Rule-Based
No 498 Stefan Svenberg: Structure-Driven Derivation of Inter-lingual Functor-Argument Trees for Multi-Lingual
FHS 8/95 Dan Fristedt: Metoder i användning - mot förbättring av systemutveckling genom situationell metodkunskap och
FHS 9/95 Malin Bergvall: Systemförvaltning i praktiken - en kvalitativ studie avseende centrala begrepp, aktiviteter och
ansvarsroller, 1995.
No 538 Staffan Flodin: Efficient Management of Object-Oriented Queries with Late Binding, 1996.
No 545 Vadim Engelson: An Approach to Automatic Construction of Graphical User Interfaces for Applications in
No 546 Magnus Werner: Multidatabase Integration using Polymorphic Queries and Views, 1996.
FiFa 1/96 Mikael Lind: Affärsprocessinriktad förändringsanalys - utveckling och tillämpning av synsätt och metod, 1996.
No 550 Kristina Larsen: Försättnings- och begränsningar för arbete på distans - erfarenheter från fyra svenska företag.
1996.
No 558 Patrik Nordling: The Simulation of Rolling Bearing Dynamics on Parallel Computers, 1996.
No 563 Niclas Andersson: Compilation of Mathematical Models to Parallel Code, 1996.
Jörgen Lindström: Chefers användning av kommunikationsteknik, 1996.
Niclas Wahllöf: A Default Extension to Description Logics and its Applications, 1996.
Fredrik Eklund: Declarative Error Diagnosis of GAPLog Programs, 1997.
Mats Gustafsson: Bringing Role-Based Access Control to Distributed Systems, 1997.
Bengt E W Andersson: Samverkande informationssystem mellan aktörer i offentliga åtaganden - En teori om aktörsarena i samverkan om utbyte av information, 1998.
Paweł Pietrzak: Static Incorrectness Diagnosis of CLP (FD), 1999.


Charlotte Björkegren: Learning for the next project - Bearers and barriers in knowledge transfer within an organisation, 1999.


Svein Bergum: Managerial communication in telework, 2000.


Jean Paul Meynard: Control of industrial robots through high-level task programming, 2000.


Magnus Kald: The role of management control systems in strategic business units, 2000.


Fredrik Elg: Ett dynamiskt perspektiv på individuella skillnader av heuristisk kompetens, intelligens, mentala modeller, mål och konflikter inom mikrovärlden Moro, 2002.


Martin V. Howard: Debugging and Structural Analysis of Declarative Equation-Based Languages, 2002.


Charlotte Stoltz: Calling for Call Centres - A Study of Call Centre Locations in a Swedish Rural Region, 2004.


Genevieve Gorrell: Rule Extraction - the Key to Accurate and Comprehensible Data Mining Models, 2004.


Per-Ola Kristensson: Large Vocabulary Shorthand Writing on Stylus Keyboard, 2004.


Vaida Jakoniené: A Study in Integrating Multiple Biological Data Sources, 2005.

Abdil Rashid Mohamed: High-Level Techniques for Built-In Self-Test Resources Optimization, 2005.


David Dinka: Role and Identity - Experience of technology in professional settings, 2005.
Andreas Hansson: Increasing the Storage Capacity of Recursive Auto-associative Memory by Segmenting Data, 2005.


Andreas Käll: Översättningar av en managementmodell - En studie av införandet av Balanced Scorecard i ett landsting, 2005.

He Tan: Aligning and Merging Biomedical Ontologies, 2006.


FiF-a 90


FiF-a 91


Jiri Trnka: Prerequisites for data sharing in emergency management, 2007.


Erik Kuiper: Mobility and Routing in a Delay-tolerant Network of Unmanned Aerial Vehicles, 2008.


Martin Karresand: Completing the Picture – Fragments and Back Again, 2008.


Mirko Thorstenson: Using Observers for Model Based Data Collection in Distributed Tactical Operations, 2008.


Mohammad Saifullah: Exploring Biologically Inspired Interactive Networks for Object Recognition, 2011.


Anna Vapen: Contributions to Web Authentication for Untrusted Computers, 2011.


Kristian Stavåker: Contributions to Parallel Simulation of Equation-Based Models on Graphics Processing Units, 2011.


