Tools and Methods for Analysis, Debugging, and Performance Improvement of Equation-Based Models

by

Martin Sjölund
Abstract

Equation-based object-oriented (Eoo) modeling languages such as Modelica 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 of paramount importance that such tools should 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.

As part of this work, methods have been developed and explored where the Eoo tool, an enhanced Modelica compiler, records the transformations during the translation process in order to provide better diagnostics, explanations, and analysis. This information is used to generate better error-messages during translation. It is also 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 essential to identify possible bottlenecks and either simplify the model or give hints to the compiler that enable 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 model executes slowly.

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. A tool with a graphical user interface has been developed to make debugging and
performance profiling easier. Both debugging and profiling have been combined into a single view so that performance metrics are mapped to equations, which are mapped to debugging information.

The algorithmic part of Modelica was extended with meta-modeling constructs (MetaModelica) for language modeling. In this context a quite general approach to debugging and compilation from (extended) Modelica to C code was developed. That makes it possible to use the same executable format for simulation executables as for compiler bootstrapping when the compiler written in MetaModelica compiles itself.

Finally, a method and tool prototype suitable for speeding up simulations has been developed. It works by partitioning the model at appropriate places and compiling a simulation executable for a suitable parallel platform.
Populärvetenskaplig sammanfattning

Som ingenjör lär man sig att förstå världen genom ekvationer. För att beskriva eller modellera ett fysikaliskt system, till exempel vid utveckling av en produkt, vill ingenjören använda sig av de ekvationer som han eller hon lärt sig från böcker eller föreläsningar. Vid produktutveckling kan ett av målen med modellen vara att hitta de bästa parametrarna för att få den snabbaste, mest hållbara eller billigaste lösningen.

Man vill således använda en dator för att göra en simulering av modellen, men de instruktioner en dator tar körer i sekvens och gör i grund och botten bara en massa beräkningar och tilldelningar till olika minnespositioner. De klassiska programmeringsparadigmerna, exempelvis imperative programmering, fungerar på liknande sätt – man skriver en algoritm som körs på uppifrån och ner.


Om ett fel inträffar vid simulering av modellen så vill man få reda på var felet kommer ifrån. Tidigare har detta varit inte varit möjligt, men i avhandlingen introduceras en metod som sparar tillräckligt med information för att visa vilken ekvation felet inträffade i och alla de symboliska operationer som utförts på ekvationen. Informationen kan sedan visas i en grafisk debugger.

För att kunna simulera i realtid, till exempel en körsimulatör där
en människa kör en simulerad bil, så måste datorn kunna lösa ekvationer runt 100 gånger i sekunden för att illusionen av att det är en simulerad bil ska förstärkas. Om ett ekvationssystem är för detaljerat så tar det för lång tid att köra det. I avhandlingen kombineras möjligheten att mäta körtid av ekvationer med den grafiska debuggaren. Då ser man inte endast vilka ekvationer som tar lång tid att köra, utan även varför de tar lång tid att köra. Ekvationerna kan sedan förenklas för att kunna köra simuleningen i realtid.
Acknowledgements

Thank you everyone who helped make this thesis come true. The list of everyone involved would be very long indeed.

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 obtained 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, although some of this took place during the slower parts of meetings.

Thanks to all my former and present colleagues at IDA. Especially to PELAB for being there during 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 the people providing bug fixes and improved the parts of the OpenModelica kernel I most often use: Adrian Pop, Per Östlund, Jens Frenkel, Willi Braun, Lennart Ochel, and Mahder Gebremedhin. Adeel Asghar for developing the graphical user interfaces, in particular for helping develop the user interface for the debuggers presented in the thesis. My work was strongly dependent on everything working together as well as it did.

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

Thanks to Bodil Mattsson Kihlström, Åsa Kärman, 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 (back
when paperword was not sent directly to the payroll department). Anne Moe, thank you for dealing with all the paperwork and tracking my progress as well as making sure I took all my courses, signed all the papers, and did all the work required to get a doctorate. It was not always easy since I was enrolled in multiple graduate schools and had multiple employers.

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.

Finally, Peter and Adrian are mentioned above. Thanks for helping me put together my thesis and getting all the pieces to fall into place. Lili Zhu, thanks for constantly reminding me when I was not working on the thesis.

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

Martin Sjölund
Linköping, April 7, 2015
## Contents

1 Introduction .................................................. 1  
1.1 Motivation ................................................. 1  
1.2 Contributions ............................................ 3  
1.3 Structure .................................................. 4  

2 Background ................................................... 11  
2.1 Modelica .................................................. 11  
2.1.1 Modelica Association ................................. 16  
2.1.2 Modelica Standard Library ......................... 17  
2.2 OpenModelica ............................................ 17  
2.3 MetaModelica ............................................. 19  
2.3.1 Pattern Matching ........................................ 20  
2.3.1.1 Semantics ......................................... 21  
2.3.1.2 Pattern Matching Fail Semantics ................ 22  
2.3.2 Data Types ............................................. 22  
2.3.2.1 Lists ................................................ 22  
2.3.2.2 Tuples .............................................. 23  
2.3.2.3 Option Types ...................................... 24  
2.3.2.4 MetaModelica Array Types ...................... 24  
2.3.2.5 Union Types ....................................... 25  
2.3.3 An Expression Interpreter ............................ 26  
2.4 Compiler Bootstrapping ................................. 29  
2.5 Symbolic Manipulation ................................... 32  
2.5.1 Variable-Equation Dependencies .................... 33  
2.5.2 Sorting and Matching ................................ 35  
2.5.3 Block Lower Triangular Matrix ..................... 37  
2.5.4 Tearing of Nonlinear Systems of Equations ....... 38
3 Debugging

3.1 Background ........................................... 41
  3.1.1 Comparison with Traditional Debugging ........ 43
  3.1.2 Sources of Errors and Faults .................. 44
    3.1.2.1 Over- and Under-constrained Systems .... 45
    3.1.2.2 Errors in the Evaluation of Expressions .... 46
    3.1.2.3 Assertion Violations in Models ............ 46
    3.1.2.4 Errors in the Solution of Implicit Algebraic Equations ........................................................................ 46
    3.1.2.5 Errors in the Integration of the ODEs .... 47
  3.1.3 Common Symbolic Operations ..................... 47
    3.1.3.1 Variable Aliasing .......................... 48
    3.1.3.2 Known Variables .......................... 48
    3.1.3.3 Equation Solving .......................... 49
    3.1.3.4 Expression Simplification .................. 49
    3.1.3.5 Equation System Simplification ............ 50
    3.1.3.6 Differentiation .......................... 51
    3.1.3.7 Index Reduction .......................... 51
    3.1.3.8 Function Inlining .......................... 52
    3.1.3.9 Scalarization .......................... 53
  3.2 Debugging Equation-based Models .................. 54
    3.2.1 Bookkeeping of Operations ................. 56
      3.2.1.1 Variable Substitution .................. 56
      3.2.1.2 Equation Solving .................. 58
      3.2.1.3 Expression Simplification ............ 58
      3.2.1.4 Equation System Simplification .... 60
      3.2.1.5 Differentiation .................. 60
      3.2.1.6 Index Reduction .................. 61
      3.2.1.7 Function Inlining .................. 61
      3.2.1.8 Scalarization .................. 61
    3.2.2 Display of Operations ....................... 61
    3.2.3 Simulation Runtime .......................... 67
    3.2.4 Implementation Results ..................... 70
    3.2.5 JSON Schema .................................. 71
  3.3 Algorithmic Code Debugger ....................... 71
    3.3.1 Algorithmic Code Debugger Implementation .... 73
  3.4 Summary and Discussion .......................... 76
CONTENTS

4 Profiling 79
4.1 Standard Profiling Tools 80
4.1.1 gprof 80
4.1.2 Valgrind 80
4.2 Connection to Debugging 83
4.3 Implementation 84
4.4 Using Profiling Data 86
4.5 Summary and Discussion 88

5 Graphical User Interface and Usage Examples 91
5.1 Introducing a Runtime Error 91
5.2 Using the Profiler 92
5.3 Inspecting Transformations 95
5.4 Use Case: Common Subexpression Elimination 95
5.5 Use Case: Finding Bugs in the Compiler 100
5.6 Using the Algorithmic Code Debugger 102
5.7 Combining the Algorithmic Code Debugger and Transformations Browser 106
5.8 Summary and Discussion 106

6 Compiler Bootstrapping 109
6.1 Introduction 110
6.1.1 Specification of Language Constructs 111
6.2 Vision – Extensible Tools 112
6.2.1 Motivation for Compiler Bootstrapping 113
6.2.2 Stages of Bootstrapping OMC 114
6.3 Implementation 115
6.3.1 Compiler 116
6.3.2 Platform Availability 118
6.3.3 Language Feature Implementations 118
6.3.3.1 Pattern Matching Implementation 118
6.3.3.2 Type Design and Implementations 121
6.3.3.3 Polymorphism 122
6.3.4 Runtime System 122
6.3.4.1 Data Layout 122
6.3.4.2 Foreign Function Interface 123
6.3.4.3 Built-in MetaModelica Functions 124
6.3.4.4 Garbage Collection 124
6.3.5 Issues 125
6.3.5.1 MMC Problems 125
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.3.5.2</td>
<td>OpenModelica Issues</td>
<td>126</td>
</tr>
<tr>
<td>6.3.5.3</td>
<td>Modelica Problems</td>
<td>126</td>
</tr>
<tr>
<td>6.4</td>
<td>Separate Compilation</td>
<td>127</td>
</tr>
<tr>
<td>6.4.1</td>
<td>Implementation</td>
<td>127</td>
</tr>
<tr>
<td>6.4.2</td>
<td>Steps in detail</td>
<td>128</td>
</tr>
<tr>
<td>6.4.3</td>
<td>Performance</td>
<td>129</td>
</tr>
<tr>
<td>6.5</td>
<td>Performance</td>
<td>130</td>
</tr>
<tr>
<td>6.5.1</td>
<td>Benchmarks</td>
<td>131</td>
</tr>
<tr>
<td>6.5.1.1</td>
<td>Comparison with Old Benchmarks</td>
<td>134</td>
</tr>
<tr>
<td>6.5.1.2</td>
<td>Comparison with Latest OMC Version</td>
<td>135</td>
</tr>
<tr>
<td>6.6</td>
<td>Algorithmic Code Debugger</td>
<td>136</td>
</tr>
<tr>
<td>6.7</td>
<td>Related Work</td>
<td>137</td>
</tr>
<tr>
<td>6.8</td>
<td>Summary and Discussion</td>
<td>139</td>
</tr>
<tr>
<td>7</td>
<td>Parallel Simulation by TLM-Style Decoupling</td>
<td>141</td>
</tr>
<tr>
<td>7.1</td>
<td>Background</td>
<td>141</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Related Work</td>
<td>144</td>
</tr>
<tr>
<td>7.1.2</td>
<td>Transmission Line Element Method</td>
<td>146</td>
</tr>
<tr>
<td>7.1.3</td>
<td>Hopsan</td>
<td>149</td>
</tr>
<tr>
<td>7.1.4</td>
<td>Hopsan-Modelica Differences</td>
<td>150</td>
</tr>
<tr>
<td>7.2</td>
<td>Example Model: Pressure Relief Valve</td>
<td>151</td>
</tr>
<tr>
<td>7.3</td>
<td>TLM in an Equation-based Language</td>
<td>153</td>
</tr>
<tr>
<td>7.4</td>
<td>Partitioning</td>
<td>158</td>
</tr>
<tr>
<td>7.4.1</td>
<td>Algorithm</td>
<td>159</td>
</tr>
<tr>
<td>7.5</td>
<td>Distributed Solver</td>
<td>160</td>
</tr>
<tr>
<td>7.6</td>
<td>Parallel Performance</td>
<td>160</td>
</tr>
<tr>
<td>7.6.1</td>
<td>Thread Safety</td>
<td>165</td>
</tr>
<tr>
<td>7.7</td>
<td>Summary and Discussion</td>
<td>165</td>
</tr>
<tr>
<td>7.7.1</td>
<td>Modelica-based Approach</td>
<td>166</td>
</tr>
<tr>
<td>7.7.2</td>
<td>FMI-based Approach</td>
<td>167</td>
</tr>
<tr>
<td>7.7.3</td>
<td>Stand-alone TLM Simulator</td>
<td>168</td>
</tr>
<tr>
<td>8</td>
<td>Conclusion</td>
<td>169</td>
</tr>
<tr>
<td>8.1</td>
<td>Future Work</td>
<td>171</td>
</tr>
<tr>
<td>A</td>
<td>Flat Modelica</td>
<td>173</td>
</tr>
<tr>
<td>B</td>
<td>Translated Code</td>
<td>179</td>
</tr>
<tr>
<td>C</td>
<td>Generated Code Fragments</td>
<td>191</td>
</tr>
<tr>
<td>CONTENTS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>-----</td>
<td></td>
</tr>
<tr>
<td>D  Transformations Browser JSON Schema</td>
<td>195</td>
<td></td>
</tr>
<tr>
<td>E  Debugging Package</td>
<td>199</td>
<td></td>
</tr>
<tr>
<td>F  MetaModelica Example: Expression Interpreter</td>
<td>211</td>
<td></td>
</tr>
<tr>
<td>G  TLM Example</td>
<td>217</td>
<td></td>
</tr>
<tr>
<td>References</td>
<td>223</td>
<td></td>
</tr>
</tbody>
</table>
List of Figures

2.1 Simple RC circuit. ........................................... 13
2.2 Simple RC circuit simulation with input square wave
pv.v, capacitor voltage c.v, and resistor voltage r.v. . 16
2.3 OpenModelica Connection Editor (OMEdit) displaying
an example model from the Modelica Standard Li-
brary (MSL). The libraries browser (left), the diagram
view (middle), and the documentation view (right). . 18
2.4 OpenModelica compiler workflow. ....................... 19
2.5 Dependencies as a bipartite graph. ............... 33
2.6 Bipartite graph using adjacency matrix representation. 34
2.7 Bipartite graphs using biadjacency matrix representa-
tion. .......................................................... 34
2.8 A lower triangular matrix is also a BLT matrix. . 37
2.9 BLT matrix representation. E0 and E3 (V0 and V1)
is a strongly connected component that needs to be
solved simultaneously as a linear or nonlinear system
of equations. ................................................. 38
2.10 Figure 2.9 after collapsing strongly connected compo-
nents. ......................................................... 38

3.1 Using information from the translation in subsequent
phases. ......................................................... 54
3.2 The number of symbolic operations performed on each
equation in the EngineV6 model. ......................... 66
3.3 OpenModelica trace (redacted for brevity). .......... 71
3.4 Algorithmic code debugger flow of control. .......... 72
4.1 KCachegrind graph when examining eqFunction, showing clock cycles of the Virtual Machine (VM) spent calling each function as well as its call count. The residual function is called 2,387,425 times for 3,917 calls to the function solving the equation block. This means the nonlinear solver is calling the residual function on average 609 times in order to solve this strongly connected component in the BLT matrix. 82

4.2 Number of calls to the function per time step. 88

5.1 Runtime error message from model with chattering behavior. 92

5.2 Transformations browser opened for runtime error message from model with chattering behavior. When opened using the link from the simulation warning, the problematic equation is selected. 93

5.3 Setting up the simulation to use the equation profiler is as simple as selecting which class of equations should be profiled. Choosing “all” will profile all equations and functions. Choosing “blocks” will profile only strongly connected components and functions. 93

5.4 Profiling results of the Modelica standard library DoublePendulum example, sorted by execution time. 94

5.5 Inspecting the operations (transformations) performed on a single equation in the Modelica standard library DoublePendulum example. The diff before-after is color-coded (red is replaced text, green is new text). 96

5.6 Debugger showing the most expensive equations to calculate when simulating CISESim120501. The most expensive nonlinear system has been expanded and the figure cropped to focus only on this part. 97

5.7 Debugger showing the most expensive equations to calculate when simulating CISESim120501 when applying CSE. The most expensive nonlinear system has been expanded and the figure cropped to focus only on this part. Applying the Common Subexpression Elimination (CSE) optimization has reduced the time taken for this nonlinear system compared to Figure 5.6. 98
5.8 Clicking on the variable cse43 in CISESim120501 displays which equations the variable is defined in (assigned to) as well as which equations use the variable. Equation 1542 (from Figure 5.7) uses this equation and it is through that equation variable cse43 was clicked. The variable is defined in equation 1498, and clicking it shows the equation and symbolic operations performed on this equation. Note that the execution count of the equation is much lower than it is for the equations in the nonlinear system shown in Figure 5.7.

5.9 Runtime domain error from EngineV6 model with a compiler generating bad code.

5.10 The debug view of the new efficient algorithmic code debugger in the MDT Eclipse plugin.

5.11 A preview of the upcoming OMEdit version of the algorithmic code debugger.

5.12 The stack frames view of the debugger.

5.13 The variables view of the new debugger.

6.1 Adding record information to the data structure to simplify the work of the debugger with small memory overhead (marked with a circle).

7.1 Transmission line components calculate wave propagation through a line using a physically correct separation in time.

7.2 The example system consists of a volume and a pressure relief valve. Boundary conditions are represented by a constant flow source and a constant pressure source.

7.3 A pressure relief valve is designed to protect a hydraulic system by opening at a specified maximum pressure.

7.4 Pressure increases until the reference pressure of 10 MPa is reached, at which the relief valve opens.

7.5 Comparison of spool position using different TLM implementations. The curves for delay, der, and sample overlap almost completely. The HOPSAN model is slightly different from the Modelica model, which causes the curves to also be slightly different.
7.6 Comparison of system pressure using different TLM implementations. The curves for delay, der, and sample overlap almost completely. The HOPSAN model is slightly different from the Modelica model, which causes the curves to also be slightly different. 156

7.7 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 results file. 157

7.8 Adjacency matrices in lower triangular form. 159

7.9 Speedup graph for parallelization using static scheduling. 163

7.10 Comparison of spool position using a volume split into more segments. 163

7.11 Comparison of system pressure using a volume split into more segments. There is a numerical error that is amplified by splitting the volume into several segments. 164
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Mean overhead cost of adding debug information.</td>
<td>56</td>
</tr>
<tr>
<td>3.2</td>
<td>Overhead cost of comparing expressions.</td>
<td>59</td>
</tr>
<tr>
<td>5.1</td>
<td>Variables introduced by common subexpressions used in the analyzed nonlinear system of CISESim120501 shown in Figure 5.7. Each of these variables introduced an additional equation somewhere in the model that can be found by searching for the variable and listing the equations that assign to the variable. The variable cse20 is defined in an equation that is part of the nonlinear system; because the equation is part of the nonlinear system it will be executed multiple times in each evaluation of the system. The other uses of CSE variables come from outside the nonlinear system, which means they should be executed fewer times than without CSE (since the function call would still be part of the system).</td>
<td>100</td>
</tr>
<tr>
<td>6.1</td>
<td>Sizes of OMC compiler phases, lines of code.</td>
<td>116</td>
</tr>
<tr>
<td>6.2</td>
<td>Sizes of OMC parser and runtime.</td>
<td>118</td>
</tr>
<tr>
<td>6.3</td>
<td>Total compile-times of files with different file sizes.</td>
<td>130</td>
</tr>
<tr>
<td>6.4</td>
<td>Compiler performance, parsing of MSL 3.2.1.</td>
<td>132</td>
</tr>
<tr>
<td>6.5</td>
<td>Compiler performance, checkModel() on the large HumMod and analytic EngineV6 models, time spent and memory consumption.</td>
<td>133</td>
</tr>
<tr>
<td>6.6</td>
<td>Compiler performance, OpenModelica test suite, run in parallel. The fast version of the test suite is comprised of 1,622 simple tests, with the full version having 2,499 tests.</td>
<td>133</td>
</tr>
</tbody>
</table>
### LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.7</td>
<td>Running the new build script for compiling OpenModelica.</td>
</tr>
<tr>
<td>6.8</td>
<td>Compiling the OpenModelica source code with the old build system (MMC) and the new one (OMC).</td>
</tr>
<tr>
<td>6.9</td>
<td>Compiler performance, checkModel() on the large HumMod and analytic EngineV6 models, parsing of Modelica Standard Library (MSL) as multiple files, time spent and memory consumption. Comparing with the latest OMC version.</td>
</tr>
<tr>
<td>7.1</td>
<td>Performance comparison of methods for performing transmission line modeling in Modelica.</td>
</tr>
<tr>
<td>7.2</td>
<td>Performance of parallelization on a four-core machine.</td>
</tr>
</tbody>
</table>
5.1 Model with chattering behavior. ......................... 92
5.2 Symbolic transformations for the equation that the de-
bugger stops at after a bug is introduced through the
compiler, in diff format. ................................. 101
7.1 OpenMP code. ........................................ 161
7.2 Serial code. ........................................... 161
A.1 Flattened RC circuit. ................................... 173
B.1 Translated RC circuit. ................................. 179
C.1 OpenModelica trace (JSON fragment). ................ 191
C.2 OpenModelica ODE code fragment. .................... 193
C.3 OpenModelica residualFunc3. ......................... 193
D.1 Full JSON Schema. .................................... 195
E.1 Debugging.mo. ......................................... 199
F.1 Expression interpreter: Exp1.mo. ....................... 211
F.2 Expression interpreter: Parse.mo. ...................... 213
F.3 Expression interpreter: lexer.l. ....................... 213
F.4 Expression interpreter: parser.y. ...................... 214
G.1 TLM example. ......................................... 217
Glossary

ANTLR ANother Tool for Language Recognition. 44, 117

API Application Programming Interface. 140, 162, 165

BDF Backward Differentiation Formula. 47, 167

BLT Block Lower Triangular. viii, 37, 77, 82, 85, 158, 159, 171

CORBA Common Object Request Broker Architecture. 117

CPU Central Processing Unit. 3, 80, 85, 89, 132, 135, 139, 141, 142, 144

CSE Common Subexpression Elimination. viii, xi, 95, 97, 98, 100, 107

DAE Differential-Algebraic Equation. 12, 46, 47, 51, 146

DAEs Differential-Algebraic Equations. 146

DFA Deterministic Finite Automaton. 119–121

EOO Equation-based Object-Oriented. 1–3, 5, 11, 41–44, 56, 71, 76, 79, 91, 110, 139, 141, 169, 170

eXtensible Markup Language a verbose text-based markup language that is both human and computer-readable. xvii, 62

FMI Functional Mock-up Interface. 17, 166–168

FMUs Functional Mock-up Units. 17, 145, 166, 168, 171

GC Garbage Collection. 124, 132, 135, 165
Glossary

GCC GNU Compiler Collection. 73, 80, 134, 162

GDB the GNU project debugger. 29, 30, 43, 73–75, 130, 136

GPGPU General-Purpose computing on Graphics Processing Units. 141, 146

GUI graphical user interface. 70, 77, 137, 169

HTML HyperText Markup Language. 70, 86, 107

JavaScript Object Notation a compact text-based human and computer-readable format. The main data-type is attribute-value pairs (maps). Despite its name, the format is language-independent. xvi, 62, 83, 195

JSON JavaScript Object Notation. xvi, 62, 65, 70, 83, 84, 195, See: JavaScript Object Notation

LLVM a collection of modular and reusable compiler and toolchain technologies. 134

MDT Modelica Development Tooling. ix, 72, 76, 102, 103, 114, 136

MMC MetaModelica Compiler. 20, 29, 30, 32, 109, 110, 114, 117–119, 121–126, 130–139

MSL Modelica Standard Library. vii, xii, 16–18, 131, 132, 135

ODE Ordinary Differential Equation. xiv, 12, 33, 45, 47, 51, 52, 87, 142, 193

ODEs Ordinary Differential Equations. 146

OMC OpenModelica Compiler. 17, 19, 29, 30, 32, 47, 51, 55, 57, 59, 60, 65, 72, 73, 75, 77, 100, 109, 110, 114, 116–127, 130–139, 160, 165, 166, 168, 170, 171

OMEdit OpenModelica Connection Editor. vii, ix, 17, 18, 72, 77, 86, 91, 92, 102, 103, 106, 107, 139, 168

PDE Partial Differential Equation. 113
**Glossary**

**PELAB** Programming Environments Laboratory, Linköping University. 11, 145

**QSS** Quantized State System. 146

**RML** Relational Meta-Language. 112, 114, 118, 121, 125, 138

**SCCs** Strongly Connected Components. 35, 37, 38, 84, 85, 97

**SCM** Source Control Management. 30

**SQL** Structured Query Language. xvii, 62, *See: Structured Query Language*

**Structured Query Language** a special-purpose programming language for managing data held by databases. xvii, 62

**TLM** Transmission Line Modeling. ix, x, 143–147, 149–151, 153, 155–160, 165–168, 170, 171

**VM** Virtual Machine. viii, 81, 82

**XML** eXtensible Markup Language. xvii, 62, 65, 67, *See: eXtensible Markup Language*
Chapter 1

Introduction

Equation-based Object-Oriented (EOO) modeling languages such as Modelica 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 perform product development these days – combining 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 using a simulation tool. Once the simulation is working, optimization and design space exploration can also be performed. This makes it possible to find for example the fastest or cheapest way to build a product. Exploring many design alternatives usually requires a large number of simulations which makes fast simulation tool performance essential. Such model-based product development processes save time and money since it is possible to test many aspects of the final system without expending resources on building a physical prototype.

1.1 Motivation

EOO modeling languages are declarative and at a high level of abstraction where models express what to compute but not how to
perform the computation. This is an advantage, but today’s EOO simulation tools currently have drawbacks regarding debugging and detecting performance problems in models. Such tools typically translate high-level models into a low-level representation such as C code, which is compiled and executed to simulate the model. If there is an error in the executing simulation it is very difficult for the user to understand, based on the generated code, which part or parts of the model are causing the problem. The low-level generated code has very little resemblance to the original model.

Understanding the complex transformations an EOO compiler performs on a typical input model is a daunting task that requires more than 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 considerable time trying to figure out the sequence of symbolic transformations even for small models containing only dozens of equations. The main reason for this difficulty is the large gap in abstraction between the model and the executable simulation code.

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. Determining what goes wrong in such large models is very difficult without additional tool support. Although manually searching through all these equations is possible, it is error-prone and certainly not practical.

This situation makes it hard for the modeler to determine and analyze how a simulation is performed computationally, such as the order in which 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 performance profiling of models in EOO languages.

Simulation of equation-based models using numerical solutions is a well-developed area where reasonably efficient methods and algorithms are known. A problem for further improved simulation perfor-
1.2. CONTRIBUTIONS

Performance is that most of these algorithms were developed with uniprocessor systems in mind. During several decades, processor performance doubled approximately every 18th month (Moore’s law). Unfortunately, this hitherto periodic increase in simulation speed has slowed significantly\(^1\). To partially compensate for this lack of performance, new chips are now designed to contain an increasing number of processor cores. Unfortunately, utilizing the full computational power of multiple processors is no trivial matter in general applications. The computational tasks must be partitioned and scheduled for parallel execution on multiple cores. Developing improved methods for partitioning and scheduling of equation-based model simulation is thus highly motivated.

1.2 Contributions

The main contributions of this work are:

- A method and prototype for tracing the symbolic transformation operations performed in the backend of an EOO modeling language (Modelica) compiler to be used in debugging tool support and in generating explanations for the user.

- A schema for standardizing storage of debugging metadata.

- A design and implementation of a performance profiler for simulations. The tool has a very low overhead, making it suitable for real-time simulations. Running the profiler shows which equation systems are slow to simulate. Combined with debug information, the tool can also point out why they are slow to simulate.

- A design and implementation of a tool that can display user-friendly information for debugging or profiling equation-based model simulations.

\(^1\)As an example, compare two Intel Core i7 CPUs with the best single threaded performance at the time of their release. The i7-965 was introduced in Q4 2008 and the i7-4790K in Q2 2014. The difference in single-threaded performance is 1.85x and the time between releases is 5.5 years (11.8% per year, doubles every 6.3 years).
• A general compilation scheme including a code generator that generates efficient C code that is easy to debug and profile using standard, usually C-based, development tools.

• A general compilation scheme that efficiently supports boot-strapping in addition to standard simulation model compilation, that is, the compiler can compile (a model of) itself to generate an efficiently executing model compiler.

• A method to parallelize simulation models partly based on transmission line modeling, and partly on automatic dependency analysis and partitioning of systems of equations.

1.3 Structure

The structure of the thesis is as follows. General background information (Chapter 2) is first presented for all the chapters. Chapters 3, 4, 5, 6, and 7 correspond to the publications presented below. These chapters contain enhanced and expanded versions of the original papers and have also been reworked to provide continuity throughout the text. The thesis is concluded with Chapter 8. Moreover, some supplementary information including sections of source code have been placed in the appendices.

• Chapter 3 is based on the following publications:
  – Adrian Pop, Martin Sjölund, Adeel Ashgar, Peter Fritzson, and Francesco Casella. “Integrated Debugging of Modelica Models”. In: Modeling, Identification and Control 35.2 (2014), pp. 93–107. DOI: 10.4173/mic.2014.2.3. A combination of the 2012 and 2011 papers below. The initial submission was made by Adrian Pop and conversion from Word to LATEX done by myself.
  – Martin Sjölund, Francesco Casella, Adrian Pop, Adeel Ashgar, Peter Fritzson, Willi Braun, Lennart Ochel, and Bernhard Bachmann. “Integrated Debugging of Equation-Based Models”. In: Proceedings of the 10th International Modelica Conference. Ed. by Hubertus Tummescheit and
1.3. STRUCTURE

Karl-Erik Årzén. Lund, Sweden: Modelica Association and Linköping University Electronic Press, Mar. 2014. doi: 10.3384/ecp14096195. My contributions are in creating a computer-readable format for the debugging metadata and providing feedback and fixing bugs during the implementation of the graphical user interface now known as the transformations browser.


version of the profiling implementation.

- Chapter 5 is a more updated version of the graphical user interfaces introduced in the articles covered by the debugging chapter.

- Chapter 6 is about model compiler bootstrapping (which is briefly explained in Section 2.4). The chapter was adapted from: Martin Sjölund, Peter Fritzson, and Adrian Pop. “Bootstrapping a Compiler for an Equation-Based Object-Oriented Language”. In: *Modeling, Identification and Control* 35.1 (2014), pp. 1–19. doi: 10.4173/mic.2014.1.1. Most of the implementation work for the bootstrapped compiler was performed by myself. The exception is garbage collection. Designed parts of the MetaModelica 2.0 language. Wrote the manuscript for the article.

- Chapter 7 is a more updated version of: Martin Sjölund, Robert Braun, Peter Fritzson, and Petter Krus. “Towards Efficient Distributed Simulation 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 University Electronic Press, Oct. 2010, pp. 71–80. URL: http://www.ep.liu.se/ecp/047/ (visited on 2011-10-07). Creating the example model and writing the manuscript was in close collaboration between Robert Braun and myself since I did not know TLM and he did not know Modelica. The partitioning and parallelization was performed later as part of my licentiate thesis [124].

The following publications, sorted by publication date, are either covered by an updated publication above or not covered in this thesis. They are all related to interoperating with or enhancing the Open-Modelica modeling and simulation tool suite including its compiler and debugger:

- Arunkumar Palanisamy, Adrian Pop, Martin Sjölund, and Peter Fritzson. “Modelica Based Parser Generator with Good Er-


• 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

Before work with the Modelica language was started, PELAB was involved in developing a language and tool called ObjectMath [149], an equation-based object-oriented (EOO) specification language for mathematical modeling. Several other groups developed related languages and tools, for example Dymola [37], NMF [121], Smile [72], and gPROMS [11]. In 1996, some of these groups joined forces to create an internationally viable declarative mathematical modeling language. The result is Modelica, which is an Equation-based Object-Oriented (EOO) modeling language for declarative mathematical modeling of large and heterogeneous (multi-domain) physical systems [38, 48, 51, 94, 139, 140]. For modeling with Modelica, commercial software products such as System Modeler [52, 153], Dymola [22], SimulationX [68], MapleSim [86], IDA Simulation Environment [40], and more are available. There are also open source implementations like OpenModelica [49, 59] and JModelica.org [156]1.

2.1 Modelica

Modelica [44, 94] is an EOO language2 that has been designed to allow tools to automatically generate efficient simulation code with

1http://modelica.org has a complete list of Modelica tools.
2Object-orientation in the sense of hierarchical modeling, inheritance, and object behavior represented by equations and functions, not object-oriented programming like Java, which tends to focus on methods in classes for representing object behavior.
the important objective of facilitating exchange and reuse of models, model libraries, and simulation specifications. Modelica has a general type system that unifies object orientation, multiple inheritance, and generics templates within a single class construct. The language supports acausal modeling, 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. It is thus possible to change how a model is solved by specifying different unknown variables to solve for. For example, solving for and computing 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 were used instead, such a change in unknowns may necessitate rewriting the entire model from scratch.

Most numerical solver packages require each (state) equation to be written as an Ordinary Differential Equation (ODE). Modelica is more general and allows using higher-index Differential-Algebraic Equation (DAE) formulation, leaving it up to the Modelica tool to translate the model into an ODE formulation in order to use numerical solvers. On top of that, the Modelica language also allows DAEs to be mixed together with discrete-time equations, forming hybrid DAEs. Discrete-time variables in Modelica are variables that change only at a specific discrete point in time (think sample and hold). Modelica 3.3 [94] also added support for clocked synchronous variables, which may not be accessed outside specific clock “ticks”.

The language has a strong software component model, with constructs for creating libraries of reusable components and connecting them together. Composing a model from existing Modelica libraries targeting different domains, such as electrical and mechanical, is as easy as dragging and dropping model components from the libraries onto a diagram and connecting the models together. This gives the user the possibility to model a much larger number of systems than simulation tools specialized in only one domain. If the goal is to model a simple serial RC circuit using a rectangular pulse-voltage source, the graphical view may be as in Figure 2.1. Behind the scenes, the tex-
2.1. MODELICA

tual 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:

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;
```

Figure 2.1: Simple RC circuit.

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 before presenting it in Listing 2.1 was to remove the graphical annotations that specify the
CHAPTER 2. BACKGROUND

graphical coordinates and graphical transformations of the components in order to reduce the volume of text, just as many Modelica text editors would do.

In Modelica the variables in a standard connector are of three kinds: potential, flow, or stream. In order to get models that balance, that is, have an equal number of equations and variables, the number of potential and flow variables in a connector must be equal. This matches physical connections rather well. A flow variable is a physical variable that resembles the behavior of a current and a potential variable is something that resembles a potential quantity such as voltage. Listing 2.2 shows what a simple connector looks like, in this case an electrical pin that transfers voltage and current. 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 for electrical connections (the sum of currents flowing into a point is zero while all connected voltages are equal) and corresponding laws for connections in other domains.

Listing 2.2: A simple electrical pin in Modelica is a connector.

```modelica
connector Pin "Pin of an electrical component"
    Modelica.SIunits.Voltage v "Potential at the pin";
    flow Modelica.SIunits.Current i "Current flowing into the pin";
end Pin;
```

The connections between the capacitor, the source and the ground in this model are expanded into equations:

\[
\begin{align*}
    c.n.i + g.p.i + pv.n.i &= 0.0, \\
    c.n.v &= g.p.v, \\
    c.n.v &= pv.n.v.
\end{align*}
\] (2.1a, 2.1b, 2.1c)

The additional equations in this case are stated in a partial class like OnePort (Listing 2.3), which is a partial electrical class with one positive and one negative pin. Components like the resistor and the
Listing 2.3: A simple electrical component with two pins.

```modelica
partial model OnePort
  "Component with two electrical pins p and n and current i from p to n"
  SI.Voltage v "Voltage drop between the two pins (= p.v - n.v)"
  SI.Current i "Current flowing from pin p to pin n"
  PositivePin p "Positive pin (potential p.v > n.v for positive voltage drop v)"
  NegativePin n "Negative pin"

equation
  v = p.v - n.v;
  0 = p.i + n.i;
  i = p.i;
end OnePort;
```
capacitor are specializations of OnePort and inherit OnePort.

Modelica has more kinds of connector classes than just the basic connectors containing flow and potential variables. For example expandable connectors for buses, input and output connectors, stream connectors for fluid systems, and over-constrained connection graphs used for example in multi-body systems. The language is thus ideally suited as an architectural description language for complex physical systems, and to some extent also for software systems and modeling of discrete controllers.

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

The result of simulating the simple RC circuit can be seen in Figure 2.2. The × signal shows the square input wave pv.v, the +
Figure 2.2: Simple RC circuit simulation with input square wave $pv.v$, capacitor voltage $c.v$, and resistor voltage $r.v$.

signal shows capacitor voltage $c.v$, and the □ signal shows the voltage over the resistor $r.v$. The input wave is a perfect square and is modeled as a discrete-time component. When used together with continuous-time components, the connected components together form a hybrid system model. Such hybrid continuous-discrete-time modeling is supported by Modelica, making the language more expressive than a purely continuous-time language.

While this short chapter is sufficient for the purpose of reading the rest of this text, Modelica has many more features from a simulation practitioner’s point of view.

To learn more about Modelica or the motivations and design goals that led to it, see books [47, 48, 140] as well as shorter overviews [38, 50, 51], and the language specification [94].

2.1.1 Modelica Association

The Modelica Association is a non-profit organization that manages a number of Modelica-related projects. The primary project is the Modelica language project, where the Modelica language is standardized, maintained, and further developed during project meetings. There is also the Modelica library project, which maintains and develops the Modelica Standard Library (MSL) and libraries that might one day be
2.2. OPENMODELICA

included in the standard library. The Functional Mock-up Interface (FMI) project is related to Modelica in the sense that most Modelica tools import or export Functional Mock-up Units (FMUs).

2.1.2 Modelica Standard Library

The Modelica Standard Library (MSL) version 3.2.1 is used in this work [93]. MSL 3.2.1 contains 1340 models from many different application domains. MSL is the best known Modelica library; it is open-source, supported and standardized by the Modelica Association and its individual members. Being a standard, models from the standard library are sometimes used as reference models for benchmarks since they are readily available and well-known in the community.

How Modelica icons are displayed is standardized, and most tools will display models similar to Figure 2.3.

2.2 OpenModelica

OpenModelica [59] is an open-source Modelica-based modeling and simulation environment that is used as a base platform in the research. However, the research in this thesis is of more general applicability and not tied to the implementation. It should be possible to use similar approaches for other EOO languages and other Modelica tools. OpenModelica consists of several subsystems.

Editing graphical and textual models is typically performed using OMEdit (Figure 2.3). The OpenModelica Compiler (OMC) performs the translation of Modelica models to executable simulations\(^3\). The compilation steps of OMC are shown in Figure 2.4. This work is primarily concerned with flattened models as they are produced by the OMC frontend (other Modelica tools have a very similar intermediate representation). A flattened model is basically the pure equation system and is similar to systems of equations in other equation-based languages, even those without object-oriented aspects (see Appendix A for an example of a flat Modelica model). It should also be possible

\(^3\)OMEdit links against OMC, so technically OMEdit can do anything OMC can. But in terms of development, OMEdit is about creating a good, user-friendly interface for modeling and analysis of simulations.
Figure 2.3: OpenModelica Connection Editor (OMEdit) displaying an example model from the Modelica Standard Library (MSL). The libraries browser (left), the diagram view (middle), and the documentation view (right).
2.3 MetaModelica

MetaModelica [54, 56, 58, 115] provides language extensions to Modelica for language modeling and model transformations. The basic language extensions are briefly described here. Some language features are defined in libraries.
2.3.1 Pattern Matching

MetaModelica pattern-matching expressions [135] may occur where expressions can be used in Modelica code. There are two kinds of pattern-matching expressions in MetaModelica, using the `match` or `matchcontinue` keywords. The syntax can be described (approximately) as follows:

```plaintext
matchcontinue (<var-list>)
local
  <var-decls>
  ...
  case (<pat-expr>)
    algorithm
    <equations>
    then <expr>;
    ...
end matchcontinue;
```

In the MetaModelica language extension only local, time-independent algorithms\footnote{The original MetaModelica design used the `equation` keyword to let the compiler sort the equations like in Modelica. The original MetaModelica Compiler (MMC) was meant to only accept single assignment equations, but did not enforce this, which led to introducing the `algorithm` keyword instead. In the future, the `equation` keyword will be supported as well.} may occur within a pattern-matching expression which must be checked by the semantic phase of the compiler. The difference between a pattern-matching expression with the keyword `match` and a pattern-matching expression with the keyword `matchcontinue` is in the fail semantics, see Section 2.3.1.2. The `matchcontinue` variant is a `match` with backtracking and continuation of the next case in the `matchcontinue` expression at failure. The `match` variant has no backtracking and makes the whole `match` expression fail if there is a failure when evaluating the matched case.

The `<pat-expr>` expression is a sequence of patterns. A pattern may be:

- A wildcard pattern, denoted `_`.
- A variable, such as `x`.
• A constant literal of built-in type such as 7, 3.14, "a string", or true.

• A variable binding pattern of the form x as pat.

• A constructor pattern of the form C(pat₁,...,patₙ), where C is a record identifier and pat₁,...,patₙ are patterns. The arguments of C may be named (for instance field₁ = pat₁) or positional but a mixture is not allowed. Constructor patterns with zero arguments are also allowed (similar to Modelica enumeration literals).

2.3.1.1 Semantics

The semantics of a pattern-matching expression are as follows: If the input variables match the pattern-expression in a case-clause, then the equations in this case-clause will be executed and the match-continue expression will return the value of the corresponding then-expression. The variables declared in the uppermost variable declaration section can be used (as local instantiations) in all case-clauses. The matching of patterns works as follows given a variable v:

• A wildcard pattern, _, will succeed matching anything.

• A variable binding pattern of the form x as pat: If the match of pat succeeds then x will be bound to the value of v.

• A variable, x, will be bound to the value of v just as x as _ would.

• A constant literal of the built-in type will be matched against v.

• A constructor pattern of the form C(pat₁,...,patₙ): v will be matched against C and the subpatterns will be matched (recursively) against parts of v.

As a small example showing subpatterns, see Listing 2.4. In the example, the expression that is the addition of the integer 1 to any real number is matched by the pattern. When an input is matched by
the pattern, then \(\text{exp, i, lhs, and rhs}\) are bound to their respective values.

Listing 2.4: MetaModelica example showing subpattern matching.

```modelica
case \(\text{exp}\) as \(\text{ADD}(\text{lhs}=\text{lhs}\) as \(\text{INTEGER}(\text{i}\) as \(1), \text{rhs}=\text{rhs}\) as \(\text{REAL}())\)
    then ...;
```

### 2.3.1.2 Pattern Matching Fail Semantics

If a `case`-clause fails in an expression with the keyword `matchcontinue` then an attempt to match the subsequent `case`-clause will take place. If there is an expression with the keyword `match`, however, then the whole expression will fail if there is a failure in one of the `case`-clauses. This work primarily deals with `matchcontinue` expressions because this is the somewhat more complicated general form, whereas `match` can be viewed as a slightly more efficient special case.

### 2.3.2 Data Types

`List`, `Tuple` and `Option` are built-in algebraic data types that are common in many languages used for meta-programming and symbolic programming. The `uniontype` is a recursive parametrized type typically used to represent trees and directed acyclic graphs, which is similar to algebraic data types in functional languages such as SML [91] and Haskell [112], and case classes in Scala [101].

#### 2.3.2.1 Lists

The following operations allow creation of lists and addition of new elements in front of lists in a declarative way. Extracting elements is accomplished through pattern-matching in `match`-expressions shown earlier.

- \(\text{List(e1, e2, e3, \ldots)}\) creates a list of elements of identical type. Examples: \(\text{List()}\) – the empty list, \(\text{List(2,3,4)}\) – a list of integers.

- \(\{\}\) – denotes an reference to an empty list.
2.3. **METAMODELICA**

- The call `cons(element, lst)` adds an element in front of the list and returns the resulting list. Also available as a new built-in operator `::` (coloncolon), for example: `element:::lst`.

Types of lists and list variables can be specified as follows:

```plaintext
type RealList = List<Real>;
```

Or directly in a declaration of a variable `rlist` that denotes a list of real numbers:

```plaintext
List<Real> rlist;
```

### 2.3.2.2 Tuples

Tuples can be viewed as instances of anonymous records. The syntax is a parenthesized list. The same syntax is used in the extended Modelica (that is, MetaModelica) presented here, and is in fact already present in standard Modelica as a receiver of values for functions returning multiple results.

- An example of a tuple literal: `(1, 2.5, "cc")`

- A tuple with a single element has a comma in order to have different syntax compared to a parenthesized expression: `(a,)`

- A tuple can be seen as being returned from a function with multiple results in standard Modelica:

  ```plaintext
  (a, b, c) := foo(x, 2, 3, 5);
  ```

The main reason to introduce tuples is for convenience of notation. It is possible to use them directly without explicit declaration of a corresponding record type. MetaModelica does not yet support polymorphic records, so tuples are also easier to use together with higher-order functions. Tuples using this syntax are already present in the major functional programming languages.

A tuple will of course also have a type. When tuple variable types are needed, they can for example be declared using the following notation:

```plaintext
type VarBND = Tuple<Ident, Integer>;
```
Or directly in a declaration of a variable `bnd`:

```plaintext
Tuple<Ident, Integer> bnd;
```

### 2.3.2.3 Option Types

Option types have been introduced in MetaModelica to provide a type-safe way of representing the common situation where a data item is optionally present in a data structure. In C-like and Java-like languages this is often represented by NULL pointers that are not type-safe and may cause program crashes even when the data has been initialized by the programmer. Examples include double-free, or usage of a copy of a previously freed pointer.

- `NONE()` represents no data present
- `SOME(e)` represents that the data item `e` is present

The option type is declared analogous to the list type:

```plaintext
type MaybeResult = Option<Result>;
```

### 2.3.2.4 MetaModelica Array Types

There is also an additional array type in MetaModelica, which is different from the standard Modelica rectangular array type. A MetaModelica array can be used to represent ragged arrays, that is, arrays of arrays that may have unequal dimension sizes. More importantly from the point of view of performance is that standard Modelica arrays need a deep copy when assigning one array to another. This means it allocates new memory and copies all the data content from the old array to the new, a linear-time operation. In MetaModelica, copying of an array is accomplished by copying a reference to the array, a constant-time operation. New arrays have to be explicitly created. The main use of arrays is currently for side effects that increase performance, for example caching partial results even if a function fails.
2.3.2.5 Union Types

The `uniontype` declaration in MetaModelica is used to introduce union types. This is similar to the algebraic data types represented by the `datatype` construct in the ML family of languages \[91\]. Consider for example the `Number` type below, which can be used to represent several kinds of number types such as integers, rational numbers, real, and complex within the same type:

```modelica
uniontype Number
  record INT
    Integer int;
  end INT;
  record RATIONAL
    Integer dividend, divisor;
  end RATIONAL;
  record REAL
    Real real;
  end REAL;
  record COMPLEX
    Real re,im;
  end COMPLEX;
end Number;
```

The most frequent use of the union type is for representation of types for trees or directed acyclic graphs. A tree is a recursive data type, and representing these is as simple as using the name of the union type as the type of a field in a record that is part of the union type. There are no restrictions or special syntax required to define mutually dependent union types as shown by `Expression.VAR` and `Subscript.SUBSCRIPT`:

```modelica
uniontype Expression
  record RCONST "A real constant"
    Real r;
  end RCONST;
  record ADD "lhs + rhs"
    Expression lhs, rhs;
  end ADD;
```
2.3.3 An Expression Interpreter

A small example describing how to evaluate is given by the function Exp1.eval in Listing 2.5. The language is trivial: binary or unary expressions with integer literals as leaf nodes. Evaluation is performed by a depth-first traversal. The full example is part of Appendix F.

Listing 2.5: Expression interpreter: Exp1.mo.

```plaintext
package Exp1

uniontype Exp
    record INTconst
        Integer integer;
    end INTconst;

    record ADDop
```
Exp exp1;
Exp exp2;
end ADDop;

record SUBop
  Exp exp1;
  Exp exp2;
end SUBop;

record MULop
  Exp exp1;
  Exp exp2;
end MULop;

record DIVop
  Exp exp1;
  Exp exp2;
end DIVop;

record NEGop
  Exp exp;
end NEGop;
end Exp;

function eval "Abstract syntax of the language
  Exp1: Evaluation semantics of Exp1"
  input Exp inExp;
  output Integer outInteger;
algorithm
  outInteger := match (inExp)
    local
      Integer ival,v1,v2;
      Exp e1,e2,e;
    /* evaluation of an integer node is the
      integer itself */
    case (INTconst(integer = ival)) then ival;
Evaluation of an addition node PLUSop is $v1 + v2$, where $v1$ and $v2$ are the evaluated results of its children $e1$ and $e2$. Subtraction, multiplication, division operators have similar specs.

```
case (ADDop (exp1 = e1, exp2 = e2))
  algorithm
  v1 := eval(e1);
  v2 := eval(e2);
  then v1 + v2;
  case (SUBop (exp1 = e1, exp2 = e2))
    algorithm
    v1 := eval(e1);
    v2 := eval(e2);
    then v1 - v2;
    case (MULop (exp1 = e1, exp2 = e2))
      algorithm
      v1 := eval(e1);
      v2 := eval(e2);
      then v1*v2;
      case (DIVop (exp1 = e1, exp2 = e2))
        algorithm
        v1 := eval(e1);
        v2 := eval(e2);
        then intDiv(v1, v2);
      case (NEGop (exp = e))
        algorithm
        v1 := eval(e);
        then -v1;
      end match;
  end match;
end eval;
end Exp1;
```
2.4 Compiler Bootstrapping

Historically, the Modelica language was primarily aimed at one main application: the modeling of physical systems. Event handling and events were also present in rather early versions of Modelica but had limitations for efficient real-time applications. With the introduction of the clocked synchronous language extensions in Modelica 3.3 [94], the language has evolved to better handle discrete-time models. In the future, the Modelica language may also gain support for variables that are able to represent tree structures. When or if this happens, the Modelica language would be well on its way to becoming a full modeling and programming language, even for modeling software artifacts, instead of only a modeling language for mainly physical systems.

The OpenModelica project could not wait for standardization of tree structures in Modelica and instead developed the MetaModelica language (Section 2.3) as a Modelica language extension. One of the reasons for this extension is to enable Modelica to model languages, including Modelica, and thus be able to model itself. Another reason is use for applications involving symbolic transformations, not just numeric computations. OMC is written in the MetaModelica language and used as a very large test case for the MetaModelica language and its compiler. The current OMC can compile both MetaModelica models and standard Modelica models to efficient C code with essentially the same structure for both kinds of models. The frontend of OMC is to a large extent the same when compiling MetaModelica and standard Modelica models. Since the OpenModelica developers use the compiler to develop and compile MetaModelica code, this means that the developers have an interest in keeping error messages useful for MetaModelica (which in turn also causes the compiler to produce useful error messages for standard Modelica models). Similarly, by improving the quality of the Modelica model compilation, the MetaModelica compiler improves; by introducing algorithmic code debuggers for MetaModelica models, debugging support of Modelica models improves as well. The debugging aspects are very important for developers; the old MMC lacked efficient tools for debugging and profiling – the bootstrapped compiler, compiling to well-structured C code, can use standard tools like the the GNU project debugger.
(GDB) [133] and Valgrind [96] in addition the Modelica-specific debugger which has been built on top of GDB [118]. For more details on the differences between OMC and MMC, see Chapter 6.

The real benefit comes with bootstrapping OMC (see Chapter 6 for more details). Since the OpenModelica source code is written in MetaModelica, and OpenModelica can compile MetaModelica source code, it can compile itself. OpenModelica generates C source code from the MetaModelica code. An old copy of this C source code is stored in the Source Control Management (SCM) system, along with the newer MetaModelica source code. In order to compile OpenModelica, an old version of OMC (compiled to C source code) will be compiled to an executable, which will then be used to compile the newer version of OpenModelica. This completes the bootstrapping since OpenModelica has now compiled itself.

Since bootstrapping of OMC was completed and the old MMC was abandoned (2014-10-25), it has been possible to use all Modelica features not supported by the old MMC as well as several new language extensions. Some of these newer language extensions produce higher quality generated code, and/or enable more readable source code. Listings 2.6 and 2.7 show the code used for a simple list mapping function in MetaModelica. Note that the built-in array comprehension in Modelica has been extended in the MetaModelica languages supported by OpenModelica to also handle lists. The built-in version is also faster than the MetaModelica 1.0 approach because it does not need to reverse a list in order to become tail recursive.

Listing 2.6: List mapping in current MetaModelica is built into the language.

```plaintext
list(1+i for i in lst);
```

Listing 2.7: List mapping in MetaModelica 1.0 requires writing auxiliary functions.

```plaintext
map1(lst, realAdd, 1.0);
```

```plaintext
function realAdd
    input Real x1;
    input Real x2;
```
2.4. COMPILER BOOTSTRAPPING

```plaintext
input Real y;
algorithm
  y := x1 + x2;
end realAdd;

function map1
  "Takes a list, a function and one extra argument, and creates a new list by applying the function to each element of the list."
  input List<ElementInType> inList;
  input MapFunc inFunc;
  input ArgType1 inArg1;
  output List<ElementOutType> outList;

  partial function MapFunc
    input ElementInType inElement;
    input ArgType1 inArg1;
    output ElementOutType outElement;
  end MapFunc;
algorithm
  outList := listReverse(map1_tail(inList, inFunc, inArg1, {}));
end map1;

function map1_tail
  "Tail-recursive implementation of map1"
  input List<ElementInType> inList;
  input MapFunc inFunc;
  input ArgType1 inArg1;
  input List<ElementOutType> inAccumList;
  output List<ElementOutType> outList;

  partial function MapFunc
    input ElementInType inElement;
    input ArgType1 inArg1;
```
output ElementOutType outElement;  
end MapFunc;  

algorithm  
outList := match(inList, inFunc, inArg1, inAccumList)  
local  
   ElementInType head;  
   ElementOutType new_head;  
   List<ElementInType> rest;  
   List<ElementOutType> accum;  

case (\{\}, _, _, _) then inAccumList;  

case (head :: rest, _, _, _)  
   equation  
      new_head = inFunc(head, inArg1);  
      accum = map1_tail(rest, inFunc, inArg1, new_head :: inAccumList);  
   then  
      accum;  
end match;  
end map1_tail;  

Bootstrapping of OMC was not necessary for implementation of the transformations browser, the profiler, or the parallelization – the generated files are the same regardless of which MetaModelica compiler is used to compile OpenModelica. However, it proved to be very useful and accelerated the process of development and maintainability of OpenModelica, allowing the developers to use language features like loops and if-statements that were not supported by the older MMC.

2.5 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 of the system (the current values of the state, non-state variables, and more). In order to simulate models correctly, equation-
2.5. SYMBOLIC MANIPULATION

based simulation tools usually first perform a series of symbolic operations on the system of equations. An introduction to this topic can be found in [44].

It suffices to say that these operations are performed in order to transform the equation system to 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 preprocessing, as well as improving numerical stability of the solution process. How to efficiently save a trace of the symbolic operations performed by the compiler is the main topic of the debugging chapter (Chapter 3).

2.5.1 Variable-Equation Dependencies

A Modelica compiler backend (recall Figure 2.4) is mainly concerned with variables and equations. In order to speed up the different analyses of the systems of equations, the backend needs to keep track of which variables belong to which equation or equations. The relation between equations and variables is represented as a bipartite graph (Figure 2.5). Bipartite graphs contain two sets of nodes that are disjoint. In the case of Modelica, variables belong to one set and equation nodes to another. An edge between an equation node and a variable node 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 in
Figure 2.6: Bipartite graph using adjacency matrix representation.

$$
\begin{bmatrix}
E_0 & E_1 & E_2 & E_3 & V_0 & V_1 & V_2 & V_3 \\
E_0 & * & & & & & & \\
E_1 & * & * & * & & & & \\
E_2 & & & X & * & & & \\
E_3 & & & * & * & & & \\
V_0 & * & * & & & & & \\
V_1 & * & X & & & & & \\
V_2 & * & * & & & & & \\
V_3 & * & * & & & & & \\
\end{bmatrix}
$$

Figure 2.7: Bipartite graphs using biadjacency matrix representation.

Figure 2.6. The matrix corresponds to the same graph as Figure 2.5, where the dotted edge is represented with $X$ and the other edges with $\ast$. For any bipartite graph, the adjacency matrix has the structure of:

$$
\begin{bmatrix}
0 & B \\
B^\top & 0 \\
\end{bmatrix}.
$$

(2.2)

It is always possible to use the more compact biadjacency matrix representation ($B$ or $B^\top$) for a bipartite graph (Figure 2.7).

While the biadjacency matrices in Figure 2.7 look dense, most Modelica models give rise to matrices with a high sparsity. In Open-Modelica, the internal data structure representation uses arrays of lists to allow efficient access while keeping memory usage low. The structure also stores information if it is the state variable or the derivative of the state variable that is used in an equation (the derivative of the state variable must be assigned in exactly one equation while the state variable must not be assigned in any equation). Both the $B$ and $B^\top$ matrices are stored since it is then possible to query which variables are used in a given equation or which equations use a given
variable. In this thesis, the biadjacency matrix will be referred to as the adjacency matrix even though the full adjacency matrix is never constructed.

2.5.2 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 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 (SCCs).

If there is a strong connection (cycle) between components, they must be solved simultaneously either as a linear system of equations:

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

or as a nonlinear system of equations:

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

The SCCs are condensed into blocks of equations in order to make the graph free of cycles and thus possible to sort topologically. As an example of what the output of the sorting and matching is, see Listings 2.8 and 2.9. This is the actual output of the OpenModelica backend with the RC circuit as input. Listing 2.9 is the output of the SCCs 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.

Keep in mind that some of the variables in the system are calcu-
Listing 2.8: 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

Listing 2.9: 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)
2.5. SYMBOLIC MANIPULATION

A sorted and matched adjacency matrix has no SCCs if it is a lower triangular matrix (there are no entries in the upper triangle). In Modelica, the adjacency matrix might be a Block Lower Triangular (BLT) matrix since SCCs are allowed. By combining the equations and variables in each SCC into a single conceptual equation and variable, the adjacency matrix becomes a lower triangular matrix without changing the order of any row or column.

The reason this is of interest 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 matrix.

Figure 2.8 shows that the execution order of equations (2, 0, 3, 1) is safe. Another valid sorting would have been (2, 0, 1, 3). Figure 2.8 is both a BLT matrix and a lower triangular matrix. The case where SCCs need to be combined (a BLT matrix that is not a lower triangular matrix) is shown in Figure 2.9. Combining the two nodes in the SCC into a single node results in the matrix shown in Figure 2.10. Here, the valid sorting is (2, (0, 3), 1) – this means equations 0 and 3 need to be solved simultaneously at runtime (or compile-time) as a
2.5.4 Tearing of Nonlinear Systems of Equations

A nonlinear system of equations is hard to solve. The formulation used is an optimization problem, trying to guess values and minimizing the error. It is especially hard to solve nonlinear equations as a black box, so symbolic preprocessing is used to make the systems easier to evaluate. One such symbolic method is tearing [39]. The
nonlinear system:

\[ 0 = f(x, y), \quad (2.5a) \]
\[ 0 = x - 2 * y \quad (2.5b) \]

has iteration variables \( x \) and \( y \). The system can be solved by guessing the values of these iteration variables and checking if the right sides of the equations evaluate to zero (the left side is by convention always zero to make the right side the residual). With tearing, as few variables as possible are chosen as iteration variables. If \( y \) is chosen as iteration variable, the system becomes:

\[ x := 2 * y, \quad (2.6a) \]
\[ 0 = f(x, y). \quad (2.6b) \]

The solution processing now becomes: Guess the iteration variable \( y \), use that guess value to calculate \( x \). Then verify the system by checking if the right side of the final equation is close to zero. Having fewer iteration variables usually makes convergence (reaching the solution) faster and more reliable.
Chapter 3

Debugging

The primary focus of this chapter is debugging of equations in an Equation-based Object-Oriented (EOO) language, which is focused on the transformations performed by the compiler. Section 3.3 describes debugging of algorithmic code that is related to traditional instruction-level debugging, used to for example step through a function call execution where the function is possibly called from an equation. The results of the two approaches are the transformations browser and algorithmic code debugger in Chapter 5.

3.1 Background

EOO modeling languages such as Modelica have significant advantages in describing large models since it is easy to construct large hierarchical models by connecting model components from a library.

However, in order to simulate such models efficiently, EOO simulation tools perform a large number of symbolic manipulations 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 could be functions that are called with the same input (caching, or common subexpression elimination).

One of the drawbacks of this symbolic manipulation is that error-messages are not understandable due to extensive symbolic manipulation. For example, the following error message says nothing about the
variables involved (there is no variable $x$ or residual in the model):

Error solving nonlinear system 2

\begin{verbatim}
  time = 0.002
  residual[0] = 0.288956
  x[0] = 1.105149
  residual[1] = 17.000400
  x[1] = 1.248448
  ...
\end{verbatim}

It is usually very hard for a typical user of an EOO tool to determine which symbolic manipulations have been performed and why. If the tool only generates a binary executable this is virtually 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 modeling errors or to improve the speed and stability of the simulation.

While some tools allow exporting a description of the translated system of equations \cite{107}, this is not enough because after symbolic manipulation the resulting equations no longer need to contain the same variables or have the same structure as the original equations.

There are two main aspects of debugging application models. One is debugging the simulation executable itself \cite{114}. 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 or causes of slow simulations by studying only result files.

One would need to look earlier in the compilation process where the equation system is symbolically optimized. Moreover, to understand the causes of possible erroneous variable values, the user also needs 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 the application model as input, or alternatively as tracing and understanding how a modeling error affects the generated actual system of equations to be solved during simulation.
3.1. BACKGROUND

3.1.1 Comparison with Traditional Debugging

When most developers hear the word debugging, they think of a statement or instruction-level debugger such as the GNU project debugger (GDB) [133] 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 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. Some set-up is required to make sure that a debugger stops at the correct location and execution count (or some other trigger). Another issue is that debuggers often change the behavior of the executable. For example, the debugger might initialize 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.

There is a fundamental difference between the needs of a debugger for an EOO language and those 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 optimizations 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 in addition to requiring compiler sources and a version of the compiler compiled with debugging symbols. 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 phases 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 users do not want to give the compiler developers their complete models since the models are confidential. This requires the user to simplify his or her model to a minimal case without confidential parts that still shows the error, which is a tedious process. If there was instead 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 also makes these traditional instruction-level debuggers unsuitable from a compiler developer’s point of view.

Since equation-based modeling 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 [17] 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.

The approach suggested in this work is debugging the symbolic optimization and transformation of equation systems. The approach is usable by tool developers and modelers alike, and 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 tool. Some modeling errors can be detected automatically at compile-time. 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 should be equipped with an error reporting mechanism at all levels, allowing error recovery by the master solver, or error reporting to the end-user in case of an irrecoverable error:

- The Ordinary Differential Equation (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
- Other numerical solvers

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

### 3.1.2.1 Over- and Under-constrained Systems

A common modeling error in Modelica is that the model ends up with too many or too few equations. The user then needs to identify which equation is either missing or needs to be removed. Bunas deals with this topic in his thesis [24].

One of the reasons this used to be a big problem was that earlier versions of the Modelica language allowed local classes to be unbalanced. The current version of the Modelica language [94] requires that local classes are balanced and includes rules for how to count the equations, which solves part of the problem. The user still needs to add or remove equations, but the error-messages are localized to the class that is unbalanced.
3.1.2.2 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 \( \sqrt{-1} \), \( \log(-3) \), \( \arcsin(2) \), \( \text{pow}(-2, 0.5) \).
- Wrong values of some variable or variables being computed, where the error is manually pointed out by a user or automatically detected as being outside the bounds given by the attributes on primitive types in Modelica (exemplified in Listing 3.1).

Listing 3.1: Example of minimum and maximum attributes in Modelica.

\[
\text{Real } x(\text{min}=0.0, \text{max}=100.0) = f(\text{time}, y);
\]

3.1.2.3 Assertion Violations in Models

During translation, initialization, or simulation, assertions inside models can be triggered when the condition being asserted becomes false or when variables go outside minimum or maximum bounds. For the initialization problem and nonlinear systems, there are guesses involved in the numerical solution of the problem. During integration, the numerical integration algorithm makes guesses to determine the longest time step it can take to evaluate the right side of the equation system as few times as possible. When triggering an error during such a guess, there will be a number of retries until a step is accepted or the assertion is actually triggered and the simulation fails.

3.1.2.4 Errors in the Solution of Implicit Algebraic Equations

During initialization or simulation of Differential-Algebraic Equation (DAE) systems, implicit systems of equations must be solved. In the
3.1. BACKGROUND

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 $A \cdot x = b$, or because said problem is singular (the square matrix $A$ cannot be inverted). 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.

3.1.2.5 Errors in the Integration of the ODEs

In OpenModelica, the DAEs are brought to 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 Backward Differentiation Formula (BDF) algorithm such as dassl [111], errors might occur. These errors may be errors from Sections 3.1.2.2–3.1.2.4. 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, its requirements need to be stated. 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 that may be performed other than the ones listed below. The operations shown below are the ones judged to be the most important and the ones that the debugger for models translated by the OpenModelica Compiler (OMC) [59] should be able to trace.
3.1.3.1 Variable Aliasing

An optimization that is 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 potential variable $v$ and flow variable $i$, a connection:

$$\text{connect}(a,b) \quad (3.1)$$

will generate alias equations:

$$a.v = b.v, \quad (3.2)$$

$$a.i + b.i = 0 \iff b.i = -a.i. \quad (3.3)$$

In a simulation results 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 optimizations 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, equations:

$$a = 4.0, \quad (3.4)$$

$$b = 4.0 - a + c \quad (3.5)$$

can be combined into:

$$b = 4.0 - 4.0 + c. \quad (3.6)$$

There is no longer a need to store the value of $a$ at each time step in the results file; it is enough to store it once for known variables, which in Modelica are parameters and constants.
3.1.3.3 Equation Solving

If the tool has determined that \( x \) needs to be solved for (in terms of the other variables) in the equation:

\[ 15.0 = 3.0 \times (x + y) \]  
(3.7)

it is needed to symbolically solve the equation to produce a simple equation with \( x \) on the left side as in:

\[ x = 15.0/3.0 - y. \]  
(3.8)

Solving for \( x \) is not always straightforward and it is not always possible to invert user-defined functions such as:

\[ 0 = f(3 \times x). \]  
(3.9)

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 nonlinear solver during runtime.

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. Simplification is related to many different optimization techniques such as constant folding. It is possible to change the order in which arguments are evaluated:

\[ \text{and}(a, false, b) \Rightarrow \text{and}(false, a, b) \Rightarrow false. \]  
(3.10)

Constant subexpressions are evaluated during compile-time:

\[ 4.0 - 4.0 + c \Rightarrow c. \]  
(3.11)

In Modelica it is also allowed to rewrite non-constant subexpressions such as:

\[ \text{max}(a, b, 7.5, a, 15.0) \Rightarrow \text{max}(a, b, 15.0). \]  
(3.12)
CHAPTER 3. DEBUGGING

It is allowed to choose to evaluate functions fewer times than in the original expression since pure functions (the default in Modelica) may not have side-effects:

\[ f(x) + f(x) + f(x) \Rightarrow 3 \times f(x). \quad (3.13) \]

It is also possible for the compiler to use knowledge about the execution model in order to make expressions run faster:

\[ \text{if cond then } a \text{ else } a \Rightarrow a, \quad (3.14) \]
\[ \text{if not cond then } false \text{ else } true \Rightarrow \text{cond}. \quad (3.15) \]

3.1.3.5 Equation System Simplification

It is of course also possible to solve some equation systems statically at compile-time instead of evaluating the system at runtime. For example a linear system of equations with constant coefficients:

\[
\begin{pmatrix}
1 & 2 \\
2 & 1
\end{pmatrix}
\begin{pmatrix}
x \\
y
\end{pmatrix}
=
\begin{pmatrix}
4 \\
5
\end{pmatrix}
\quad (3.16)
\]

can be solved using one step of symbolic Gaussian elimination:

\[
\begin{pmatrix}
1 & 2 \\
0 & -3
\end{pmatrix}
\begin{pmatrix}
x \\
y
\end{pmatrix}
=
\begin{pmatrix}
4 \\
-3
\end{pmatrix}
\quad (3.17)
\]

generating two separate equations that can be solved individually after causalization:

\[ x = 2, \quad (3.18a) \]
\[ y = 1. \quad (3.18b) \]

A simple linear equation system such as (3.16) may also be solved numerically at compile-time using for example LAPACK [3] routines.
3.1. BACKGROUND

3.1.3.6 Differentiation

Symbolic differentiation [36] is used for many purposes. It is used to symbolically expand known derivatives:

\[ \frac{\partial}{\partial t} t^2 \Rightarrow 2 \ast t \]  

(3.19)

or as an operation during index reduction. Symbolic Jacobian matrices consisting of derivatives have many applications, for example to speed up simulation runtime [19]. Such a matrix is often computed using automatic differentiation [36] that combines symbolic differentiation with other techniques to achieve fast computation. If there is no symbolic Jacobian available, a numerical Jacobian might instead be estimated by the numerical solvers.

3.1.3.7 Index Reduction

In order to solve (hybrid) DAE’s numerically, simulation tools use discretization 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 [44]. While there are techniques to solve DAEs of higher index than 1, most of them require index-1 DAEs or ODEs (no second derivatives). The OMC default method currently combines Pantelides’s method for index reduction (which is a systematic graph-theoretical approach that adds and differentiates existing equations in the system) [105, 132] with dynamic state selection (which replaces some of the derivatives with algebraic variables – the dummy derivatives method) [87, 88, 89].

As an example of index reduction, an equation system was taken from [154]. Consider the example DAE below:
There are two variables \( x \) and \( y \), and each variable needs to be solved in one equation. Because \( y \) is only present in one equation, it would be natural to assume \( y \) is solved in that equation:

\[
x(t) = \cos(t),
\]
(3.22)

\[
y(t) = \cos(t) - x''(t).
\]
(3.23)

The problem with this formulation is that the second derivative of \( x \) is now present on the right side. Solving the second equation will only give values for \( x \); not its first or second derivative. The solution is to for example differentiate the second equation twice:

\[
x''(t) + y(t) = \cos(t),
\]
(3.24)

\[
x''(t) = -\cos(t).
\]
(3.25)

The equations can be symbolically simplified for better numerical precision:

\[
x''(t) = -\cos(t),
\]
(3.26)

\[
y(t) = 2 \times \cos(t).
\]
(3.27)

This system can be solved efficiently using ODE solvers.

### 3.1.3.8 Function Inlining

Writing functions to perform common operations is a great way to reduce the burden of maintaining code since each operation is defined by a function at only one place. The problem is that a function call causes some overhead. This becomes a noticeable fraction of the computational cost for the whole invocation and computation for small functions. When inlining a function call such as:

\[
2 \ast f(x, y, y)/\pi
\]
(3.28)
given a known function \( f \):

\[
f(x, y, z) = \pi \ast (\sin(x + y) + \cos(x + y - z))
\]

the compiler can treat the function call as a macro expansion, resulting in:

\[
2 \ast \pi \ast (\sin(x + y) + \cos(x + y - y)) / \pi
\]

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:

\[
2 \ast (\sin(x + y) + \cos(x)).
\]

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 modeler 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.

### 3.1.3.9 Scalarization

Scalarization 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 scalarizing 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, such as knowing that the left side and right side are arrays of the same size:

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

then it can be beneficial to split the array equation into one scalar equation for each array element:

\[
a = x \quad b = y \quad c = z.
\]
CHAPTER 3. DEBUGGING

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 optimization of large application models can be 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 optimization stages to provide 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 the context). Two programs that output a trace in order to help in the debugging of C applications are strace and valgrind [96]. The strace program (truss on some UNIX systems) logs all system calls and signals that the attached process is receiving. Valgrind is a suite of tools for debugging (and profiling) programs. Its default tool MEMCHECK
helps C programmers understand errors due to bad usage of point-
ers. 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 results.
Both strace and valgrind display the information to the user with
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 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 metadata asso-
ciated to equations, variables or equation systems. Other metadata
that should already be propagated from source code to runtime in-
clude 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 metadata is a constant runtime factor from
storing a new head in the list. The memory cost depends on the lan-
guage 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 opera-
tion). 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 ex-
pressions now referenced by being stored as metadata would normally
become garbage and subsequently freed, memory usage increases, but

\(^1\)In OMC, basic control flow uses backtracking which is due to the MetaMod-
elica 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
the program flow returns to the old state but this time it does not try to perform
the failed operation.
CHAPTER 3. DEBUGGING

<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>1,298</td>
<td>5.39s</td>
<td>465 MB</td>
</tr>
<tr>
<td>EngineV6</td>
<td>11,116</td>
<td>46.2s</td>
<td>863 MB</td>
</tr>
<tr>
<td>EngineV6*3</td>
<td>33,348</td>
<td>55.3s</td>
<td>926 MB</td>
</tr>
<tr>
<td>EngineV6*10</td>
<td>11,160</td>
<td>196s</td>
<td>2021 MB</td>
</tr>
</tbody>
</table>

Table 3.1: Mean overhead cost of adding debug information.

In OpenModelica, collection of the debug information was disabled to see the actual effects of this overhead in a real-world implementation. The result shown in Table 3.1 is the mean over nine runs of translating several models, excluding code generation, in the standard library [93]. Multiple EngineV6 components were used for increasing the number of equations in the last models. Because OpenModelica uses garbage collection to reclaim unused heap memory, the code sometimes runs faster and uses less memory if the debug information is collected than if it is not. The reason is possibly that the garbage collector elects to allocate more memory sooner when it cannot free as much memory. There is naturally an overhead cost of collecting additional data. But with a garbage collector, this cost is so low, typically less than one percent of compilation time, that collection of debug information is performed by default in OpenModelica.

3.2.1 Bookkeeping of Operations

Each different kind of symbolic operation eventually needs to be recorded in a data structure. Because the equation systems handled by EOO tools can be very large, this needs to be done efficiently. How to efficiently collect the operations is discussed below.

3.2.1.1 Variable Substitution

The elimination of variable aliasing and elimination of variables with known values (constants) is considered here. Aliasing and elimination of known variables is treated as the same operation and can be done within the same 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:

\[
\begin{align*}
    a &= b, \\
    b &= -c, \\
    c &= 4.5
\end{align*}
\]

could be represented as a more detailed version:

\[
\begin{align*}
    a &= b \Rightarrow a = -c \Rightarrow a = -4.5
\end{align*}
\]

instead of the shorter:

\[
\begin{align*}
    a &= b \Rightarrow a = -4.5
\end{align*}
\]

depending on the order in which the substitutions were performed.

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).

OMC 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 such as:

\[
\begin{align*}
    a &= b - c + d \Rightarrow a = b - b + d \Rightarrow a = d
\end{align*}
\]

which means that there is a need for a work-around for substitutions performed in a suboptimal order.

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

\[ \frac{x}{y} = 1 \Rightarrow x = y \ (y \neq 0) \] (3.38)

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:

\[ y \neq 0, \ 46.0 < y < 173.3 \Rightarrow 46.0 < y < 173.3 \] (3.39)

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 the simplification algorithm oscillates as in:

\[ 2 \times x \Rightarrow x \times 2 \Rightarrow 2 \times x \] (3.40)

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.

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 conservative in the sense that it does not save trace information that a change was made if it performs changes that may cause oscillating behavior. Finding where this behavior occurs is not hard for a compiler developer (for example, simply print an error message after 10 iterations).
3.2. DEBUGGING EQUATION-BASED MODELS

<table>
<thead>
<tr>
<th>Model</th>
<th>Eqs</th>
<th>Time</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>MultiBody EngineV6</td>
<td>11,116</td>
<td>21.1</td>
<td>0.627</td>
</tr>
<tr>
<td>Spice3 FOURBIT</td>
<td>10,154</td>
<td>49.9</td>
<td>1.34</td>
</tr>
<tr>
<td>MultiBody DoublePendulum</td>
<td>1,298</td>
<td>1.66</td>
<td>0.0530</td>
</tr>
<tr>
<td>Fluid Controlled Tanks</td>
<td>492</td>
<td>2.00</td>
<td>0.0428</td>
</tr>
<tr>
<td>Digital Counter</td>
<td>282</td>
<td>1.14</td>
<td>0.0108</td>
</tr>
<tr>
<td>Thermal Indirect Cooling</td>
<td>163</td>
<td>0.700</td>
<td>0.0129</td>
</tr>
<tr>
<td>Rotational Friction</td>
<td>98</td>
<td>0.593</td>
<td>0.00477</td>
</tr>
<tr>
<td>Analog ChuaCircuit</td>
<td>44</td>
<td>0.517</td>
<td>0.00150</td>
</tr>
<tr>
<td>Blocks Filter</td>
<td>34</td>
<td>0.652</td>
<td>0.00509</td>
</tr>
</tbody>
</table>

Table 3.2: Overhead cost of comparing expressions.

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 before and after 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.

The overhead cost of comparing the input and output expressions after each simplification operation has been estimated by modifying OMC to make this extra check. The times without overhead include all phases of translation excluding parsing and code generation. The input models are all from the standard library [93].

The results in Table 3.2 suggest that the overhead cost increases slightly as the model size increases. As the model size increases, the redundancy in the system increases. This enables a number of optimizations most of which cause expression simplification to be run. It is previously known that the time Modelica backends take to translate a model does not scale linearly to the size of the equations [43], 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. It is recommended 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 OMC only performs limited simplification of strongly connected components, it is currently limited to only recording evaluation of constant linear systems. As more of these optimizations are added to the compiler, they will also need to be traced and support for them added in the debugger. One example is solving for \( y \) in:

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

Another example would be tracing the tearing operation, which causes the solution of a nonlinear system to be found more efficiently (recall Section 2.5.4). Support for tearing was recently added to OMC but is not yet part of the trace.

3.2.1.5 Differentiation

Whenever the compiler performs symbolic differentiation in an expression, for example to expand known derivatives such as:

\[
x' = time' \implies x' = 1.0
\]

(3.42)

this operation is recorded and associated to the equation in the internal representation. Currently OMC does not eliminate the state variable in:

\[
x' = 1.0 \implies x = time + (x_{start} - time_{start})
\]

(3.43)

but if it was performed, that operation would also be recorded.
3.2. DEBUGGING EQUATION-BASED MODELS

3.2.1.6 Index Reduction

For the used index reduction algorithm (dummy derivatives):

- Record any substitutions that are made (dummy derivatives replace states with new algebraic variables).
- Add source position information to the new dummy variables.
- Add the operations performed on the affected equations (symbolic differentiation, expression simplification, additional variable substitutions).

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 Scalarization

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 Display of Operations

Until now the focus has been on collecting symbolic operations and recording these as metadata structures associated with 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
a Structured Query Language (SQL) database) or simply as structured data (for example as eXtensible Markup Language (XML) or JavaScript Object Notation (JSON) text files). In this way the data can be accessed by various applications and presented in different ways according to user needs for all of them.

SQL was dismissed as it was believed the performance gain from using a database instead of a text file would not be worth it. Text files are easy to read and write, making them simple to implement. There are also many JSON and XML serialization and de-serialization packages, making it possible to use built-in data types in the languages to map debugging metadata to for example Python dictionaries and lists.

The OpenModelica compiler stores metadata for each equation and variable in the translated model. Part of this metadata is the operations performed on the equation, stored as a list of symbolic operations. The list is treated as a stack and when some operations are pushed onto the stack, they are merged with the top element so that they look like a single operation (for example variable substitutions, which in OpenModelica is performed in many steps). Listing 3.2 shows the internal MetaModelica representation of the symbolic operations (see Section 2.3 for an overview of MetaModelica data types):

Listing 3.2: Data type for symbolic operations.

```modelica
uniontype SymbolicOperation "Symbolic operations"
  record FLATTEN "From one equation/statement to an element"
    SCode.EEquation scode;
    Option<Element> dae;
  end FLATTEN;

  record SIMPLIFY "Before and after expression is equivalent"
    EquationExp before;
    EquationExp after;
  end SIMPLIFY;
```
3.2. DEBUGGING EQUATION-BASED MODELS

record SUBSTITUTION "A chain of substitutions"
  list<Exp> substitutions;
  Exp source;
end SUBSTITUTION;

record OP_INLINE "Before and after inlining of function calls"
  EquationExp before;
  EquationExp after;
end OP_INLINE;

record OP_SCALARIZE "Convert array equation into scalar equations; \(x = \{1,2\}, \[1\] \Rightarrow x[1] = \{1\}\)"
  EquationExp before;
  Integer index;
  EquationExp after;
end OP_SCALARIZE;

record OP_DIFFERENTIATE "Differentiate w.r.t. \(cr\)"
  ComponentRef cr;
  Exp before;
  Exp after;
end OP_DIFFERENTIATE;

record SOLVE "Solve equation, \(exp1 = exp2 \Rightarrow cr = exp\); note that assertions may have been generated for example in case of divisions"
  ComponentRef cr;
  Exp exp1;
  Exp exp2;
  Exp res;
  list<Exp> assertConds;
end SOLVE;

record SOLVED "Equation is solved"
    ComponentRef cr;
    Exp exp;
end SOLVED;

record LINEAR_SOLVED "Solved linear system of equations"
    list<ComponentRef> vars;
    list<list<Real>> jac;
    list<Real> rhs;
    list<Real> result;
end LINEAR_SOLVED;

record NEW_DUMMY_DER "Introduced a dummy derivative (from index reduction)"
    ComponentRef chosen;
    list<ComponentRef> candidates;
end NEW_DUMMY_DER;

record OP_RESIDUAL "Converted the equation into residual form, to use nonlinear equation solvers 0=e (0=e1-e2)"
    Exp e1;
    Exp e2;
    Exp e;
end OP_RESIDUAL;
end SymbolicOperation;

public uniontype EquationExp "An equation on residual or equality form has 1 or 2 expressions. For use with symbolic operation tracing."
record PARTIAL_EQUATION "An expression that is part of the whole equation"
OMC outputs JSON [35] at present (see Section 3.2.5 for the schema); previous versions output XML or unstructured text. MessagePack [61], a binary format similar to JSON, was considered but dismissed as an option since JSON files are not much larger since most data is stored as strings and text-based files are much easier to edit and inspect. The JSON implementation is faster than the XML version since the text is less verbose. It is also easier to read JSON files since it nicely maps to common data types in most programming languages. The debugging information is presented by a tool with a graphical user interface, the transformations browser, see Chapter 5. The format assumes that the equations can be represented as plain text in the JSON format. It also assumes that the source code for the model is a textual file and that equations can map to a position in this source file. If this is not the case for the targeted language and compiler, other references can be inserted and you lose a part of the debugging functionality in tools that assume a textual source file. If the source file is in a binary format, a graphical debugger could also be updated to understand how to display such equations.

The number of operations stored for each equation varies widely. The reason is that when a known variable, for example \( x \), is replaced by the number 0.0 for example, 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 optimizer may make [43]. This makes some synthetic models hard to debug. The example model in Listing 3.3
performs $1 + 2 + \ldots + N$ substitutions and simplifications in order to deduce that $a[1] = a[2] = \ldots = a[n]$:

Listing 3.3: Alias model with poor scaling.

```plaintext
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;
```

When testing these methods on a real-world example (the EngineV6 model), the majority of equations have less than 4 operations (Figure 3.2), which means most equations were easy to solve. The highest number of operations was 16 which is a manageable number to go
3.2. DEBUGGING EQUATION-BASED MODELS

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 includes 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 of interest to debug unless it is suspected that the backend produced the wrong result for a constant equation.

3.2.3 Simulation Runtime

Runtime error messages need to provide enough information to be able to understand the source of the problem. Adding the equation index or indexes associated with the error source is sufficient. OpenModelica simulation executables emit each message as structured data (XML) including these equation indexes. In this way, any error message related to a numerical solver, assertion, domain error, or similar is able to give an initial hint of where to look for problems. The transformations browser uses this information to select the initial view when opening the debugger after an error was triggered.

The error and equation info from a toy example Listing 3.4 is shown in Listings 3.5 and 3.6:

Listing 3.4: Toy example.

```model Model
  Real x(fixed=true, start=1.0);
  Real y;
  equation
    der(x) = -2*time;
    y = log(x - 0.5);
end Model;
```
CHAPTER 3. DEBUGGING

Listing 3.5: Runtime Error.

```xml
<message stream="assert" type="warning" text="The following assertion has been violated at time 0.708000"/>
<message stream="assert" type="debug" text="Model error: Argument of log(x + -0.5) was -0.001264 should be &gt; 0">
<used index="4"/>
</message>
```

Listing 3.6: Corresponding equation information to runtime error.

```xml
{
    "eqIndex": 4,
    "section": "regular", "tag": "assign",
    "defines": ["y"],
    "uses": ["x"],
    "equation": ["log(x + -0.5)"]
}
```

```xml
"source": {
    "info": {"file": "Model.mo", "lineStart": 6, "lineEnd": 6, "colStart": 3, "colEnd": 19},
    "within": ["Model"],
    "operations": [
        {
            "op": "info", "display": "solved",
            "data": ["y = log(x + -0.5)"]
        },
        {
            "op": "before-after", "display": "flattening",
            "data": [
                "y = log(x - 0.5);",
                "y = log(x + -0.5);"
            ]
        }
    ]
}
```
The listings show how a debugger can combine the information from the simulation runtime with the static debug information provided by the tool. Here, the displayed error comes from the algebraic equation handling part of the solver. It clearly shows that \( \log(0) \) is not defined and the source position of the error in the concrete textual Modelica syntax form (the Modelica code that the user 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 the view to the diagram view of this component). The symbolic transformations performed on the equation are also present, which can help in debugging additional problems with the model.

Equations might depend on one or more variables. These variable dependencies are also shown and can be used to navigate the equation system. In particular, it is possible to get the variable view for the dependent variable \( x \), shown in Listing 3.7:

Listing 3.7: Corresponding variable information.

```
"x": {
  "comment": "",
  "kind": "state",
  "type": "Real",
  "unit": "", "displayUnit": "",
  "source": {...},
  "within": ["Real"], "typeLst": ["M", "Real"]
}
```

It is also possible to query the equation that defines \( x \) in the section for regular equations, shown in Listing 3.8:

Listing 3.8: Equation information for equation solving for \( x \).

```
{
  "eqIndex": 5,
  "section": "regular", "tag": "assign",
  "defines": ["der(x)"],
  "uses": ["x"],
  "equation": ["-x"],
  "source": {...}
}
```
CHAPTER 3. DEBUGGING

Using the equation view, it can quickly be realized that \( x \) will converge towards 0.0. Since \( x \) starts at value 1.0, this will cause the expression \( 0.5 - x \) to become 0 at some point in time.

The user interface performs these lookup operations behind the scenes. All the user needs to do is to click his or her way through the equation system and read the information provided. 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 the required equation indexes to open up the transformations browser of the optimized equation system.

3.2.4 Implementation Results

The model used in this section corresponds to Listing 3.9:

```
model M
  Real x,y,z;
equation
  z = 1.0;
  x = y*time*z;
  y = sin(x*time);
end M;
```

Listing 3.9: Modelica nonlinear system.

Translation of this model results in a JSON file (Listing C.1, in Appendix C). While it is possible for the user to inspect this file and detect diffs manually, it is tedious and error-prone to do so. Instead a graphical user interface (GUI) of some kind is preferable. It could be an HTML report or the GUI presented in Chapter 5, and the user would be provided with color-coded differences of transformations such as those shown in Figure 3.3. Even so, if no GUI was available, the debugging information is much more accessible than trying to decipher the translated simulation source code (Listings C.2 and C.3, in Appendix C) which does not include any information about the origin of the equation.
3.3. Algorithmic Code Debugger

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

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

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

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 (redacted for brevity).

3.2.5 JSON Schema

Having a (de facto) standardized format for the debugging metadata was one of the most requested requirements since the first publication introducing the concept of debugging EOO models. The reason is that other tool vendors and users would like to be able to either extract information from the metadata or create their own metadata files. The schema was thus chosen to keep common operations simple and extensible so that tools other than OpenModelica would be able to add new operations that can still be displayed in the existing debugger. An example of this is the before-after operation shown in Listing 3.10:

Listing 3.10: JSON Schema; before-after operations.

```
{
"op": "before-after",
"display": "simplify",
"data": [
"a = 1.0 * fn(x)",
"a = fn(x)"
]
}
{
"op": "before-after",
"display": "inline",
"data": [
"a = fn(x)",
"a = 273.15 + x"
]
}
```

The entire schema can be found in Appendix D.

3.3 Algorithmic Code Debugger

The information regarding the algorithmic code debugger used for debugging dynamic programs at runtime is spread throughout the
This section contains information regarding the implementation of the debugger in general. Section 6.6 has the parts specific to the bootstrapped version of OMC (MetaModelica data types). Section 5.6 presents the user of the algorithmic debugger. This section refers to the implementation of the debugger for Modelica Development Tooling (MDT), which is an Eclipse plugin. The implementation is the same for the OpenModelica Connection Editor (OMEdit) counterpart of the algorithmic code debugger which can be launched from the usual simulation dialog box. Both implementations of the algorithmic code debuggers, MDT and OMEdit, have the same features.

The user interface for the algorithmic code debugger has been developed by Pop and Asghar [118]. The debugger uses the debugging information that was added to OpenModelica-generated C code during the development of the bootstrapped OpenModelica compiler (Chapter 6). From being a necessity to debug the compiler using a command line interface to the debugger, it has grown into a debugger with a graphical user interface for any algorithmic Modelica or MetaModelica code.

Figure 3.4: Algorithmic code debugger flow of control.
3.3. Algorithmic Code Debugger

3.3.1 Algorithmic Code Debugger Implementation

In order to keep track of Modelica source code positions, the Modelica source code line numbers are inserted into the transformed C source code. This information is used by the GNU Compiler Collection (GCC) to create the debugging symbols that can be read by the GDB [133].

Figure 3.4 shows how the bootstrapped OMC (presented in Chapter 6) propagates the line number information all the way from the high level Modelica representation to the low level intermediate representation and the generated code.

Consider the Modelica code shown in Listing 3.11:

Listing 3.11: Modelica Code: HelloWorld.mo.

```modelica
function HelloWorld
    input Real x;
    output Real y;
algorithm
    y := sin(2*x+1/x);
end HelloWorld;
```

OMC compiles this HelloWorld function into intermediate C source code:

Listing 3.12: Generated C source code, HelloWorld.c, before post-processing line directives.

```c
1 double omc_HelloWorld ( threadData_t * threadData ,
2                     double _x )
3 {
4     double _y ;
5     double tmp1 ;
6     _tailrecursive : OMC_LABEL_UNUSED
7     /*# modelicaLine [ HelloWorld .mo :5:3 -5:20]*/
8     tmp1 = _x ;
9     if ( tmp1 == 0 ) { throwStreamPrint ( threadData ,
10                     "Division by zero \%s", "1.0 / x");}
11     _y = sin ( ((2.0 * _x) + (1.0 / _x)) ) ;
12     /*#endModelicaLine*/
13     _return : OMC_LABEL_UNUSED
14     return _y ;
15 }
```
CHAPTER 3. DEBUGGING

The generated code contains blocks that represent the Modelica code lines. The generated intermediate C source code is converted in a streaming filter while outputting the data structure to a file. During this step, the modelicaLine directives are converted to standard C line preprocessor directives:

Listing 3.13: Generated C source code, HelloWorld.c, after post-processing line directives to standard C.

```c
#define OMC_FILE "HelloWorld.c"

double omc_HelloWorld(threadData_t *threadData, double _x)
{
    double _y;
    double tmp1;
    _tailrecursive: OMC_LABEL_UNUSED
    #line 5 "HelloWorld.mo"
    tmp1 = _x;
    #line 5 "HelloWorld.mo"
    if (tmp1 == 0) { throwStreamPrint(threadData, "Division by zero %s", "1.0 / x");}
    #line 5 "HelloWorld.mo"
    _y = sin(((2.0 * _x) + (1.0 / _x)));  
    #line 14 OMC_FILE
    _return: OMC_LABEL_UNUSED
    return _y;
}
```

Note that one file preprocessor directive is required for each line in the C-file, even though it corresponds to only one line in the Modelica code. Since the OpenModelica text template language (also called Susan [60]) used to specify the C code generator does not have support for automatically inserting line preprocessor directives on every new line in its output stream, the additional post-processing is necessary.

The executable is created from the converted C source code and is debugged from the Eclipse-based Modelica debugger which converts Modelica-related commands to low-level GDB commands at the C code level.

The Eclipse interface allows adding and removing breakpoints. The breakpoints are created by sending the following command to
GDB: \(<\text{-break-insert filename:linenumber}>\)

At the moment only line number based breakpoints are supported. Other alternatives to set the breakpoints are:
\(<\text{-break-insert function}>\)
\(<\text{-break-insert filename:function}>\)

These program execution commands are asynchronous because they do not send back any acknowledgement. However, GDB raises signals:

- As a response to those asynchronous commands.
- For notifying program state.

The debugger uses the following signals to perform specific actions:

- \textit{breakpoint-hit} – raised when a breakpoint is reached.
- \textit{end-stepping-range} – raised when step-into or step-over operations are finished.
- \textit{function-finished} – raised when a step-return operation is finished.

These signals are utilized by the debugger to extract the line number information and highlight the line in the source code editor. They are also used as notifications for the debugger to start the routines to fetch the new values of the variables.

The suspend functionality which interrupts the running program is implemented in the following way: on Windows, GDB interrupts do not work. A small program called \texttt{BreakProcess} was therefore written to allow interrupts on Windows. The debugger calls \texttt{BreakProcess} by passing it the process ID of the debugged program. \texttt{BreakProcess} then sends the \texttt{SIGTRAP} signal to the debugged program so that it will be interrupted. Interrupts on Linux and OS X work by default.

The algorithmic code debugger is operational and works without performance degradation on large algorithmic Modelica and Meta-Modelica applications such as OMC, with more than 330,000 lines of code\textsuperscript{2}.

\textsuperscript{2}For more details on code size and implementation details of OMC, see Section 6.3.1.
The algorithmic code debugging framework graphical user interface is developed in Eclipse as a plugin that is integrated into the existing OpenModelica MDT. The tracking of line number information and the runtime part of the debugging framework is implemented as part of the OpenModelica compiler and its simulation runtime.

The algorithmic code debugger currently supports the standard Modelica data types including arrays and records as well as all the additional MetaModelica data types such as ragged arrays, lists, and tree data types. It supports algorithmic code debugging of both simulation code and MetaModelica code.

Furthermore, in order to make the debugging practical (as a function could be evaluated in a time step several hundred times) the debugger supports conditional breakpoints based on the time variable or hit count.

The algorithmic code debugger can be invoked from the MDT model evaluation browser and breaks at the execution of the selected function to allow the user to debug its execution.

3.4 Summary and Discussion

A design and implementation for tracing symbolic transformations and operations in a compiler (the OpenModelica compiler [59]) for an EOO language (Modelica [94]) has been constructed. Collecting the symbolic operations (excluding writing them to file) has a tiny overhead compared to the total compilation time. Writing all the information to a verbose text-based file is cheap enough in the general case but may take as long as the rest of the compilation time used by the backend for some models with huge 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 such as a database. The current implementation also takes consideration to that some operations, such as cascading alias elimination, should be seen as an expandable operation that combines a simple overview and a detailed view.

The debug information is provided in a separate file that is also
used by the runtime to obtain user-friendly names and dependencies of equations in order to produce better error-messages. The runtime system of OpenModelica simulations has been extended to take full advantage of the debug information.

The file format for the debug information has been standardized in order to make it easy for other tools to either generate debug metadata that OMEdit can open, or to create a user interface that can take the debugging metadata that OMC generates as input.

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.

A graphical debugger and browser for the transformations has been designed (see Chapter 5). The design can be used for static debugging, runtime debugging, and to provide understandable runtime errors. Without this tool, or a similar one, a regular user would not receive color-coded differences of symbolic transformations. Instead the user would need to detect such differences manually in the text-based debugging metadata, which, while not as useful as a GUI, is still more readable than the translated source code.

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

This work focused mainly on operations performed on individual equations. By analyzing the Block Lower Triangular (BLT) matrix of the equation system, a user could additionally be presented with equations that need to be solved before or after the selected equation. The BLT matrix is implicitly available in the debugging metadata for different sections of the equation system (for example the initial and regular equation systems).

Operations performed on algorithmic code or optimizations that can be performed on the hybrid parts of the system, such as dead
code elimination of unreachable discrete events, have not yet been considered.

Information about the number of states, sizes of equation systems, and similar statistics are useful in gaining an overall perception of how efficient a model is to simulate. OpenModelica keeps track of this and is able to show the information to a user. But the debugger is currently unable to display this information, or similar information. If a user forgets to ask for this information, the compiler needs to run one additional time to produce this information. It would be useful if such debug information was always included in the model metadata, along with all such relevant information, to give the user a complete picture. The overhead memory cost of collecting and writing such information is minimal and is always performed. The cost of writing the information to disk is not costly either and could be accomplished in a general manner, providing the information as plain-text information in the metadata. However, it could also be provided as optional structured information so that the debugging tool could provide its own analysis of the system instead of the user.
Chapter 4

Profiling

By using performance profiling it is possible to obtain information about how long certain parts of a program or model take to execute or how many times they are executed. For Modelica models it can be important to detect which equations take a long time to execute, such as when considering real-time simulations. It may also be desirable to speed up regular simulation models for example when performing a large parameter sweep. Profiling is a useful tool for both users (modelers) of Equation-based Object-Oriented (EOO) tools and EOO tool developers alike.

The requirements of the implemented profiling tool:

- Recording the total execution time that is spent trying to solve any given equation, function, and the overall simulation executable.

- Recording how the amount of execution time changes over time for any given equation, function, and the overall simulation executable.

- Low overhead so that it is possible to run the profiler on code that is running in real-time.

- Recording worst-case execution times for the time step of any given equation, function, or the overall simulation executable.

- Ability to connect the generated data to the transformations browser (Chapter 5).
4.1 Standard Profiling Tools

The methodology used when profiling Modelica programs is quite different from profiling of general programs. Part of the reason for the difference is due to performance, but the connection with real hardware and real-time constraints makes a different approach necessary.

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 certain 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. Two CPU-time profilers for C-based programs are described below to provide a sense of common profiling approaches:

4.1.1 gprof

The gprof [64] 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 is currently executing. This is statistical sampling. After the program (or multiple runs of the same program) finishes, the output files can be analyzed.

Experience shows that (GNU) gprof is not suitable for profiling huge amounts of generated code since it is common to get a measurement of no time spent in short functions. While the (BSD) gprof manual references a higher resolution timer, it is only used for kernel profiling. Profiling with gprof also only works truly well if every single source-file is recompiled, including supporting libraries, with the -pg flag. While running the instrumented code is fast, analyzing the data afterwards is slow. An advantage is that gprof is almost universally available – GNU gprof is supported on any platform the GNU Compiler Collection (GCC) [62] runs on.

4.1.2 Valgrind

Valgrind [96] is known as a memory debugging tool, but it is actually a framework for creating dynamic analysis tools. One such use is pro-
4.1. STANDARD PROFILING TOOLS

Valgrind currently provides three (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 function names only) and might achieve more intuitive results if optimizations like function inlining are disabled. Because Valgrind is essentially a Virtual Machine (VM), the executed code for callgrind runs approximately 100 times slower than usual. It is however 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 fast since the callgrind tool performs more processing than gprof. Experience shows that 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 [93] (the C code used to solve the causal equation 564). 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 solver is 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 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{residualFunc\_564}) 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 total 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 optimization is required to make it easier for the nonlinear system
Figure 4.1: KCachegrind graph when examining eqFunction_564, showing clock cycles of the VM spent calling each function as well as its call count. The residual function is called 2,387,425 times for 3,917 calls to the function solving the equation block. This means the nonlinear solver is calling the residual function on average 609 times in order to solve this strongly connected component in the BLT matrix.
solver to find the roots. An alternative would be to modify the model to become linear instead.

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

4.2 Connection to Debugging

Because Modelica does not have named or numbered equations, it is necessary to associate a generated equation with the source code and transformations performed on this source code. OpenModelica keeps track of this information and has a debugger for displaying the generated equation system and all transformations performed on this system (see Chapter 3).

For a model M, the files M.prof.json, M.prof.intdata and M.prof-realdata are generated from the simulation executable when performing profiling (details on how are covered in Section 4.3). Listing 4.1 contains a sample JavaScript Object Notation (JSON) profiling file. Note that each equation has its equation index stored - this is used to connect the profiling data to the debugger. The JSON file format was chosen because it is easy to read and most programming libraries support accessing the data format in an easy manner.

Listing 4.1: Profiling metadata for DoublePendulum.

```json
{
    "name": "DoublePendulum",
    "prefix": "DoublePendulum",
    "date": "2014-08-15 13:23:06",
    "method": "dassl",
    "outputFormat": "mat",
    "outputFilename": "DoublePendulum_res.mat",
    "outputFilesize": 1165863,
    "overheadTime": 0.000639289,
    "preinitTime": 0.00154915,
    "initTime": 0.0120341,
```
"eventTime":0.000109358,
"outputTime":0.0073238,
"linearizeTime":0,
"totalTime":0.0393712,
"totalStepsTime":2.1047e-05,
"totalTimeProfileBlocks":0.0149883,
"numStep":499,
"maxTime":0.001274192,
"functions":[
],
"profileBlocks":[
  {"id":87,"ncall":3,"time":0.011467365,"maxTime":0.011468191},
  {"id":876,"ncall":6858,"time":0.003520976,"maxTime":0.000561836}
]}

The JSON file contains a list of the functions and blocks that are profiled, which correspond to rows or columns in the binary files (the files are transposed as a final step, to make reading of trajectories more efficient). The execution count and times in each individual time step is recorded in these binary files and is used to display graphs with all the details. The summary is useful in detecting which equations are slow, and the graphs are used to detect if an equation becomes slow at a specific point in time.

4.3 Implementation

The Modelica code that is being profiled consists of equation blocks and function calls. Equation blocks are defined as Strongly Connected
4.3 IMPLEMENTATION

Components (SCCs) in the Block Lower Triangular (BLT) matrix. It is necessary to combine the profiling information with information from the transformations browser in order to relate the profiling information to variable names or positions in the source.

The OpenModelica profiler uses compiler-assisted source code instrumentation. There is one call to the clock before executing the equation block or function call and one call to the clock after execution of the block. Associated with each call is a counter that keeps track on how many times this function was triggered for the given time step. Similarly, each call is associated with clock data – one variable for the total time spent in the block for all time steps, and one variable for the total time spent in the block for the current time step. These calls to the clock are in the code generation as a macro set that is generated if profiling is desired – this means zero overhead unless profiling is explicitly enabled.

Profiling can be enabled for all equations and function calls. With profiling enabled only for equation blocks (SCCs) and functions, the overhead cost is low compared to the cost of solving most nonlinear systems of equations, which is more suitable for real-time simulation. The instrumentation is performed by compiling a model with additional code generated to query real-time clocks at appropriate places. The time measurement uses the best real-time clock available on the platform used\(^1\). Windows, Linux, and OS X 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 were used. The output is written to binary files at each time step, as a count (number of calls, an integer in the intdata file) and time spent (in the realdata file). The time spent is stored as double precision floating point in seconds or CPU cycles instead of the internally measured time which is usually an integer of CPU cycles or a time in nanoseconds.

The advantage to using a statistical approach is that accurate average numbers are provided, but for real-time systems worst-case

\(^1\)OpenModelica uses \texttt{clock_gettime} (Linux), \texttt{QueryPerformanceCounter} (Windows), or \texttt{mach_absolute_time} (OSX).
execution time needs to also be considered to 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, the method can be used together with statistical methods. Simply put, the numerical solvers perform many time steps in order to attain accurate simulation results. The data collected from running many time steps is then analyzed to calculate an estimate of worst- and average-case execution time to compute one time step, for each profiled equation block as well as the time to compute the accepted step (including solver overhead).

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 help("profiling") to see the options supported by profiling. There is a drop-down box in the OpenModelica Connection Editor (OMEdit) where these profiling options can be set (see Figure 5.3 in the next chapter).

4.4 Using Profiling Data

Before OMEdit included support for profiling data, support was added to generate an HTML report including graphs and a user-friendly summary. Using this summary, it should be easy for example, to improve the quality of OpenModelica-generated source code. In OpenModelica, profiling has detected many scaling issues and detected poor code generation for many operations that work with arrays.

By observing the output of the profiler, it can be seen that in the SimpleNonLinear model example, \( \cos(x) \) will be called three times at each time step:

Listing 4.2: Simple nonlinear equation.

```plaintext
model SimpleNonLinear
    Real x = cos(x);
end SimpleNonLinear;
```
4.4. USING PROFILING DATA

While it is possible to determine the value of $x$ during compile-time or initialization, and then keep the value constant, OpenModelica does not yet perform these optimizations. It is thus important to model it as a parameter or 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, tenCos is called a total of 593 times for 499 time steps:

Listing 4.3: 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;
```

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 initialization. While this behavior is intended when using the default Ordinary Differential Equation (ODE) solver, DASSL, it causes the graph (Figure 4.2) to look a little odd. DASSL 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 a decision is made. This results in tenCos being called between two and four times for some output points, when DASSL performs some internal calculations, and one 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.
4.5 Summary and Discussion

By using profiling techniques, it is possible to detect which equations cause low performance in a simulation. With 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 optimized equation system which means that it is hard to know their origin [23]. By combining the profiling information and the debugging information it is possible to not only see which systems were performing slowly, but also why they were performing slowly.

Measured functions do not have an option to measure each function call separately. Currently only the top-most function is measured for efficiency but it has been determined that it is often functions called from within this function that take the longest time 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 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. The profiling information could be extended to include all such relevant statistics at one convenient location.

There would be no major impact on performance by including the above proposed additions to the profiler. However, all useful information cannot be collected if real-time performance is required. An additional mode that would allow a high overhead to collect all information could be of interest.

One alternative approach to consider is creating a replacement for KCachegrind that combines the information from callgrind and debug information to attain a more accurate result than in the version that has been presented, for example collecting execution times of all function calls (not just the top-level function calls). The obvious disadvantage is that the performance penalty would be so great that it could not be used for real-time profiling and that it would be tied to Linux.

Missing in OpenModelica is the option to generate code for real-time platforms, as well as cross-compiling code for different CPU architectures. This is an important feature that will be addressed in future projects. The current approach uses external libraries such as DeviceDrivers [138] to synchronize the clock to real-time. This approach only works well for fixed step numerical solvers such as Euler or Runge-Kutta since Modelica libraries cannot access any internals of the numerical solver.
Chapter 5

Graphical User Interface and Usage Examples

The topics of the previous two chapters were debugging and performance profiling. More specifically: how to generate the information needed in order to create debugging and profiling tools for Equation-based Object-Oriented (EOO) languages, with special emphasis on the Modelica language. The topic of this chapter is the current functionality of such a debugging tool with a graphical user interface and what it could provide in the future.

The main tool described in this chapter was designed and developed to be a part of the OpenModelica Connection Editor (OMEdit). The debugging tool is called the transformations browser. This chapter begins by describing the basic usage of the user interface, and continues with more realistic use cases.

There is also an algorithmic code debugger implemented both in Eclipse and as part of OMEdit (Section 5.6).

5.1 Introducing a Runtime Error

Listing 5.1 contains a model with chattering behavior. Chattering in the numerical integrator means that the integrator needs to reduce the step size to a tiny value due to a state variable changing rapidly.

\footnote{The model is based on Debugging.Chattering.ChatteringEvents1, found in Appendix E.}
Chattering detected around time 1.0000001..1.00000199 (100 state events in a row with a total time delta less than the step size 0.0022). This can be a performance bottleneck. Use -lv LOG EVENTS for more information. The zero-crossing was: \( x \geq 0.0 \) Debug more

Figure 5.1: Runtime error message from model with chattering behavior.

around one point. In this case, the derivative of the state variable is either -1 or 1, but never 0:

Listing 5.1: Model with chattering behavior.

```plaintext
model Chattering
    Real x(start = 1.0, fixed = true);
end Chattering;
```

When running this model, the progress bar will nearly halt and the error in Figure 5.1 is generated and presented. There are several useful things about this error-message: first of all, there is information that the simulation is experiencing slow performance. The zero-crossing expression that causes the slowness is also displayed. And the best part in the error-message is the link generated at the end. Clicking on Debug more opens up the transformations browser shown in Figure 5.2 with the equation containing the zero-crossing selected. If the model is larger, this is very important since it is possible to see where the zero-crossing is coming from and why it contains the variable that it says it contains.

5.2 Using the Profiler

Enabling use of the profiler is simple: choose the set of equations to be profiled from the drop-down box in the OMEdit simulation settings as shown in Figure 5.3. As soon as the simulation finishes, the transformations browser will open and display the measured times of each measured equation. Figure 5.4 shows that the equations have
5.2. USING THE PROFILER

Figure 5.2: Transformations browser opened for runtime error message from model with chattering behavior. When opened using the link from the simulation warning, the problematic equation is selected.

Figure 5.3: Setting up the simulation to use the equation profiler is as simple as selecting which class of equations should be profiled. Choosing “all” will profile all equations and functions. Choosing “blocks” will profile only strongly connected components and functions.
Figure 5.4: Profiling results of the Modelica standard library DoublePendulum example, sorted by execution time.

<table>
<thead>
<tr>
<th>Index</th>
<th>Type</th>
<th>Equation</th>
<th>Executions</th>
<th>Max Time</th>
<th>Execution Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Regular</td>
<td></td>
<td>0.0000</td>
<td>0.098</td>
<td>0.0000</td>
</tr>
<tr>
<td>2</td>
<td>Regular</td>
<td></td>
<td>0.0000</td>
<td>0.200</td>
<td>0.0000</td>
</tr>
<tr>
<td>3</td>
<td>Regular</td>
<td></td>
<td>0.0000</td>
<td>0.300</td>
<td>0.0000</td>
</tr>
<tr>
<td>4</td>
<td>Regular</td>
<td></td>
<td>0.0000</td>
<td>0.400</td>
<td>0.0000</td>
</tr>
<tr>
<td>5</td>
<td>Regular</td>
<td></td>
<td>0.0000</td>
<td>0.500</td>
<td>0.0000</td>
</tr>
<tr>
<td>6</td>
<td>Regular</td>
<td></td>
<td>0.0000</td>
<td>0.600</td>
<td>0.0000</td>
</tr>
<tr>
<td>7</td>
<td>Regular</td>
<td></td>
<td>0.0000</td>
<td>0.700</td>
<td>0.0000</td>
</tr>
<tr>
<td>8</td>
<td>Regular</td>
<td></td>
<td>0.0000</td>
<td>0.800</td>
<td>0.0000</td>
</tr>
<tr>
<td>9</td>
<td>Regular</td>
<td></td>
<td>0.0000</td>
<td>0.900</td>
<td>0.0000</td>
</tr>
<tr>
<td>10</td>
<td>Regular</td>
<td></td>
<td>0.0000</td>
<td>1.000</td>
<td>0.0000</td>
</tr>
<tr>
<td>11</td>
<td>Regular</td>
<td></td>
<td>0.0000</td>
<td>1.100</td>
<td>0.0000</td>
</tr>
<tr>
<td>12</td>
<td>Regular</td>
<td></td>
<td>0.0000</td>
<td>1.200</td>
<td>0.0000</td>
</tr>
<tr>
<td>13</td>
<td>Regular</td>
<td></td>
<td>0.0000</td>
<td>1.300</td>
<td>0.0000</td>
</tr>
<tr>
<td>14</td>
<td>Regular</td>
<td></td>
<td>0.0000</td>
<td>1.400</td>
<td>0.0000</td>
</tr>
<tr>
<td>15</td>
<td>Regular</td>
<td></td>
<td>0.0000</td>
<td>1.500</td>
<td>0.0000</td>
</tr>
<tr>
<td>16</td>
<td>Regular</td>
<td></td>
<td>0.0000</td>
<td>1.600</td>
<td>0.0000</td>
</tr>
<tr>
<td>17</td>
<td>Regular</td>
<td></td>
<td>0.0000</td>
<td>1.700</td>
<td>0.0000</td>
</tr>
<tr>
<td>18</td>
<td>Regular</td>
<td></td>
<td>0.0000</td>
<td>1.800</td>
<td>0.0000</td>
</tr>
<tr>
<td>19</td>
<td>Regular</td>
<td></td>
<td>0.0000</td>
<td>1.900</td>
<td>0.0000</td>
</tr>
<tr>
<td>20</td>
<td>Regular</td>
<td></td>
<td>0.0000</td>
<td>2.000</td>
<td>0.0000</td>
</tr>
</tbody>
</table>

**Variable:**
- damperReal
- damperTime
- damperFrame

**Definitions:**
- damperReal = damper
- damperTime = damper
- damperFrame = damper

**Notes:**
- Damper results for the Modelica standard library DoublePendulum example.
5.3. INSPECTING TRANSFORMATIONS

now been annotated with execution count, total execution time and maximum execution time for a single time step. The figure is sorted by total execution time, which shows that a single linear equation system solving for two variables dominates execution time. It also shows the names of those two variables and can show the equations associated with the rows in the A matrix of the linear system.

5.3 Inspecting Transformations

Inspecting transformations is one of the basic operations that can be performed by a user of the transformations browser. Figure 5.5 shows some of the transformations for that particular model. The transformations can be viewed for many purposes. For real-time systems, the browser can be used to understand why a nonlinear system is formed (and which term to remove in order to improve performance). If the simulation crashes, it can be used to see which equation the compiler generates (maybe it solves for a different variable than the user expected and triggers a division-by-zero).

5.4 Use Case: Common Subexpression Elimination

This section illustrates a use-case for one of the debugging tasks of the transformations browser that a modeler (user) would find useful. Other optimizations can be debugged in a similar manner.

The Modelica Media and Fluid libraries have been designed around the assumption that the Modelica compiler will perform many different kinds of optimizations. One of these optimizations is Common Subexpression Elimination (CSE). In OpenModelica ticket #3102 [28], an example from the ThermoPower library is discussed. Performance of this model is a major concern for the modeler who submitted the ticket and the performance profiling from Chapter 4 showed that the (expensive to compute) water properties functions were called a surprising number of times.

One interesting and perhaps counter-intuitive aspect of CSE is that it helps improve performance even if the function call only exists
CHAPTER 5. GRAPHICAL USER INTERFACE AND USAGE EXAMPLES

Figure 5.5: Inspecting the operations (transformations) performed on a single equation in the Modelica standard DoublePendulum example. The diff before-after is color-coded (red is replaced text, green is new text).

```plaintext
= Equation Operations
```
5.4. **USE CASE: COMMON SUBEXPRESSION ELIMINATION**

<table>
<thead>
<tr>
<th>Index</th>
<th>Type</th>
<th>Equation</th>
<th>Executions</th>
<th>Max time</th>
<th>Time</th>
<th>Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1417</td>
<td>regular</td>
<td>non-linear, size 9</td>
<td>817162</td>
<td>9.83</td>
<td>41.5</td>
<td>29.3%</td>
</tr>
<tr>
<td>1410</td>
<td>regular</td>
<td>Plant:Downcomer:wbar...Plant:Downcomer:N[(j)]</td>
<td>812086</td>
<td>0.0218</td>
<td>0.0382</td>
<td>0.0269%</td>
</tr>
<tr>
<td>1408</td>
<td>regular</td>
<td>Plant:Downcomer.dM6...Plant:Downcomer:p</td>
<td>812086</td>
<td>0.0259</td>
<td>0.0356</td>
<td>0.0251%</td>
</tr>
<tr>
<td>1414</td>
<td>regular</td>
<td>-15.923*Plant:Downcomer:Q:Q_single[t1] (residual)</td>
<td>812086</td>
<td>0.0208</td>
<td>0.0336</td>
<td>0.0237%</td>
</tr>
<tr>
<td>1413</td>
<td>regular</td>
<td>Plant:Header:Lower.dE...Downcomer:hold[t1]</td>
<td>812086</td>
<td>0.0209</td>
<td>0.0315</td>
<td>0.0222%</td>
</tr>
<tr>
<td>1411</td>
<td>regular</td>
<td>Plant:Downcomer:outf...Plant:Downcomer:NT</td>
<td>812086</td>
<td>0.0193</td>
<td>0.0283</td>
<td>0.0199%</td>
</tr>
<tr>
<td>1415</td>
<td>regular</td>
<td>Plant:Header:Lower:R*...:derLower.p (residual)</td>
<td>812086</td>
<td>0.0195</td>
<td>0.0264</td>
<td>0.0186%</td>
</tr>
<tr>
<td>1409</td>
<td>regular</td>
<td>der(Plant:Header:Lower...ER.Plant:Downcomer:p)</td>
<td>812086</td>
<td>0.0173</td>
<td>0.019</td>
<td>0.0134%</td>
</tr>
<tr>
<td>1416</td>
<td>regular</td>
<td>(Plant:Downcomer:out:...ressDrop.w (residual)</td>
<td>812086</td>
<td>0.0173</td>
<td>0.0182</td>
<td>0.0128%</td>
</tr>
<tr>
<td>1320</td>
<td>regular</td>
<td>non-linear, size 27</td>
<td>140118</td>
<td>0.102</td>
<td>14.7</td>
<td>10.4%</td>
</tr>
</tbody>
</table>

Figure 5.6: Debugger showing the most expensive equations to calculate when simulating CISESim120501. The most expensive nonlinear system has been expanded and the figure cropped to focus only on this part.

in a single assignment. The reason for this is that equations may be placed in Strongly Connected Components (SCCs) and thus become part of a nonlinear system of equations. Such systems typically execute each residual equation at least three times. With CSE, if the function call’s actual parameters do not use any iteration variable, the function call can be moved out from the equation and be executed only a single time. This gives the model simulation a significant performance increase. If the same function call is present in multiple equations, as is the case in the ThermoPower example, the performance gain is even greater.

The model has been translated twice, with and without the experimental CSE module, in order to produce the debugging session shown below\(^2\). Figure 5.6 is what the debugger displays with mostly default settings, and Figure 5.7 when applying CSE.

When applying the CSE optimization, the debugger can be used to view which CSE variables are defined and used in the equations of the nonlinear system. Table 5.1 shows the variables introduced when performing the CSE optimization and which of the equations in the nonlinear system references them. Note that only one of the

\(^2\)Exact flags used: +d=infoXmlOperations,nogen +profiling=all +replaceHomotopy=actual as well as +d=disablePartitioning +cseCall +cseEachCall for CSE.
CHAPTER 5. GRAPHICAL USER INTERFACE AND
USAGE EXAMPLES

Figure 5.7: Debugger showing the most expensive equations to calculate when simulating CISeSim120501 when applying CSE. The most expensive nonlinear system has been expanded and the figure cropped to focus only on this part. Applying the CSE optimization has reduced the time taken for this nonlinear system compared to Figure 5.6.

equations defines a CSE variable and six variables are used. Five of these variables are not defined in this nonlinear system (or any nonlinear system), which means they will be executed before the nonlinear system runs.

Figure 5.8 shows the information that one obtains when looking for one of the variables used in Table 5.1. When selecting the variable introduced by the CSE algorithm, a list of equations that define or use this variable is shown. It is also possible to sort the list of equations by which variable it assigns to in order to show more of the variables from Table 5.1 in the same figure to get an overview of their total simulation times. The number of calls to these equations (5,072) is significantly fewer than the number of calls to the equations inside the nonlinear system (748,104). And the equations in the nonlinear system are easier to solve than the same system without CSE (where the solver chooses to execute the equations a total of 812,806 times).

Without CSE, the nonlinear system takes 41.5 seconds which is 29.3% of the total 141.6 seconds including measuring overhead. With CSE, the nonlinear system takes 1.25 seconds which is 10.2% of the total 12.3 seconds including measuring overhead. The nonlinear system with CSE introduces a few additional variables and equations that take some additional time, but it is not significant as the total

<table>
<thead>
<tr>
<th>Index</th>
<th>Type</th>
<th>Equation</th>
<th>Execution time</th>
<th>Max time</th>
<th>Time</th>
<th>Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1551</td>
<td>regular</td>
<td>-15.923 * Plant.Downc..._single[1] (residual)</td>
<td>753176</td>
<td>0.202</td>
<td>1.22</td>
<td>10.2%</td>
</tr>
<tr>
<td>1548</td>
<td>regular</td>
<td>Plant.Downc..._single[1] (residual)</td>
<td>748104</td>
<td>0.0137</td>
<td>0.0221</td>
<td>0.186%</td>
</tr>
<tr>
<td>1544</td>
<td>regular</td>
<td>Plant.Downc..._single[1] (residual)</td>
<td>748104</td>
<td>0.0132</td>
<td>0.0203</td>
<td>0.171%</td>
</tr>
<tr>
<td>1549</td>
<td>regular</td>
<td>Plant.Downc..._single[1] (residual)</td>
<td>748104</td>
<td>0.0132</td>
<td>0.0198</td>
<td>0.166%</td>
</tr>
<tr>
<td>1547</td>
<td>regular</td>
<td>Plant.Downc..._single[1] (residual)</td>
<td>748104</td>
<td>0.0132</td>
<td>0.0192</td>
<td>0.161%</td>
</tr>
<tr>
<td>1542</td>
<td>regular</td>
<td>Plant.Downc..._single[1] (residual)</td>
<td>748104</td>
<td>0.0119</td>
<td>0.018</td>
<td>0.151%</td>
</tr>
<tr>
<td>1545</td>
<td>regular</td>
<td>Plant.Downc..._single[1] (residual)</td>
<td>748104</td>
<td>0.0127</td>
<td>0.0171</td>
<td>0.143%</td>
</tr>
<tr>
<td>1546</td>
<td>regular</td>
<td>Plant.Downc..._single[1] (residual)</td>
<td>748104</td>
<td>0.0132</td>
<td>0.0167</td>
<td>0.14%</td>
</tr>
<tr>
<td>1543</td>
<td>regular</td>
<td>Plant.Downc..._single[1] (residual)</td>
<td>748104</td>
<td>0.0117</td>
<td>0.0122</td>
<td>0.102%</td>
</tr>
<tr>
<td>1550</td>
<td>regular</td>
<td>Plant.Downc..._single[1] (residual)</td>
<td>748104</td>
<td>0.0117</td>
<td>0.0118</td>
<td>0.0991%</td>
</tr>
</tbody>
</table>
5.4. USE CASE: COMMON SUBEXPRESSION ELIMINATION

Figure 5.8: Clicking on the variable cse43 in CISESim120501 displays which equations the variable is defined in (assigned to) as well as which equations use the variable. Equation 1542 (from Figure 5.7) uses this equation and it is through that equation variable cse43 was clicked. The variable is defined in equation 1498, and clicking it shows the equation and symbolic operations performed on this equation. Note that the execution count of the equation is much lower than it is for the equations in the nonlinear system shown in Figure 5.7.
CHAPTER 5. GRAPHICAL USER INTERFACE AND USAGE EXAMPLES

<table>
<thead>
<tr>
<th>Equation index</th>
<th>Defines</th>
<th>Uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>1542</td>
<td>cse43, cse45</td>
<td></td>
</tr>
<tr>
<td>1544</td>
<td>cse20</td>
<td></td>
</tr>
<tr>
<td>1546</td>
<td></td>
<td>cse268</td>
</tr>
<tr>
<td>1547</td>
<td></td>
<td>cse264</td>
</tr>
<tr>
<td>1548</td>
<td></td>
<td>cse20, cse42</td>
</tr>
</tbody>
</table>

Table 5.1: Variables introduced by common subexpressions used in the analyzed nonlinear system of CISESim120501 shown in Figure 5.7. Each of these variables introduced an additional equation somewhere in the model that can be found by searching for the variable and listing the equations that assign to the variable. The variable cse20 is defined in an equation that is part of the nonlinear system; because the equation is part of the nonlinear system it will be executed multiple times in each evaluation of the system. The other uses of CSE variables come from outside the nonlinear system, which means they should be executed fewer times than without CSE (since the function call would still be part of the system).

Simulation time with CSE is shorter than only this single nonlinear system without CSE applied. When running the simulation without profiling, the overhead of measuring every single equation is lost, reducing the simulation time to 7.8 seconds for CSE and 68.9 seconds without CSE, corresponding to an 8.8 times reduction in simulation time from performing the expected CSE optimizations.

5.5 Use Case: Finding Bugs in the Compiler

Another use case for the debugger is finding out when the compiler does something wrong. This particular example is more useful for developers since they know how to solve the problem, but a modeler can find out which operation is causing the error and write a good bug report.

In order to test debugging of a compiler-introduced bug, a bug was purposefully introduced into the OpenModelica Compiler (OMC). The model tested is Modelica.Mechanics.MultiBody.Examples.Loops.EngineV6 and the results are shown in Figure 5.9.
The following assertion has been violated at time 0.000000


Debug more

simulation terminated by an assertion at initialization

Figure 5.9: Runtime domain error from EngineV6 model with a compiler generating bad code.

The transformations browser lists 9,899 equations after removing aliases and it starts with the focus on the equation and transformations shown below:

Listing 5.2: Symbolic transformations for the equation that the debugger stops at after a bug is introduced through the compiler, in diff format.

Simplify


Inline

The goal here is in trying to figure out if the compiler performed an unexpected operation or if there is a modeling error. The first operation, inlining the length function appears to be correct since the length of a 3-dimensional vector \( \vec{v} \) is \( \sqrt{\vec{v} \cdot \vec{v}} \). In the second operation the inlined expression is simplified wrongly to \( \sqrt{-x^2 - y^2 - z^2} \) instead of the expected \( \sqrt{x^2 + y^2 + z^2} \). It is a compiler error since the trace shows that the correct expression was given, but it is transformed into an expression that is not equivalent to the first one.

5.6 Using the Algorithmic Code Debugger

The debugger part for algorithmic Modelica code is implemented within the OpenModelica environment as a debugging plugin for Modelica Development Tooling (MDT) which is a Modelica programming perspective for Eclipse. The Eclipse-based user interface of the new efficient debugger is depicted in Figure 5.10. The same functionality also exists in OMEdit, so that users will be able to debug code without installing Eclipse (Figure 5.11).

The algorithmic code debugger provides the following general functionalities:

- Adding and removing breakpoints.
- Step over – moves to the next line, skipping the function calls.
- Step in – steps into the called function.
- Step return – completes the execution of the function and returns to the point from where the function is called.
- Suspend – interrupts the running program.
- Resume – continues the execution from the most recent breakpoint.
- Terminate – stops the debugging session.

It is much faster and provides more stepping options than the old algorithmic code debugger [113]. The old debugger was based on a high-level source code instrumentation which made the performance
5.6. Using the Algorithmic Code Debugger

Figure 5.10: The debug view of the new efficient algorithmic code debugger in the MDT Eclipse plugin.

Figure 5.11: A preview of the upcoming OMEdit version of the algorithmic code debugger.
CHAPTER 5. GRAPHICAL USER INTERFACE AND USAGE EXAMPLES

Figure 5.12: The stack frames view of the debugger.

slow down proportionally to the size and number of variables in the function. The debug view primarily consists of two main views:

• The stack frames view

• The variables view

The stack frames view, shown in Figure 5.12, shows a list of frames that indicates how the flow has moved from one function to another or from one file to another. This allows backtracing of the code. It is possible to select the previous frame in the stack and inspect the values of the variables in that frame. However, it is not allowed to select any of the previous frames and start debugging from there.

Each frame is shown as `<function_name at file_name:line_number>`. The variables view (Figure 5.13) shows the list of variables at a certain point in the program. It contains four columns:

• Name – the variable name.

• Declared Type – the Modelica type of the variable.

• Value – the variable value.

• Actual Type – the mapped C type.

By preserving the stack frames and the variables it is possible to keep track of the variables values. If the value of any variable is
### 5.6. Using the Algorithmic Code Debugger

#### Figure 5.13: The variables view of the new debugger.

<table>
<thead>
<tr>
<th>Name</th>
<th>Declared Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>b</code></td>
<td>Boolean</td>
<td>false</td>
</tr>
<tr>
<td><code>b1</code></td>
<td>Boolean</td>
<td>false</td>
</tr>
<tr>
<td><code>b2</code></td>
<td>Boolean</td>
<td>false</td>
</tr>
<tr>
<td><code>count</code></td>
<td>Integer</td>
<td>13</td>
</tr>
<tr>
<td><code>inStatements</code></td>
<td>record&lt;Interactive.Statements.JSTMTS&gt;</td>
<td>record&lt;Interactive.Stmt&gt;</td>
</tr>
<tr>
<td><code>interactiveStmtList</code></td>
<td>list&lt;record&lt;Interactive.Statement,ASP&gt;&gt;</td>
<td>&lt;1 item&gt;</td>
</tr>
<tr>
<td><code>exp</code></td>
<td>record&lt;Absyn.Exp.CALL&gt;</td>
<td>record&lt;Absyn.Exp.CALL&gt;</td>
</tr>
<tr>
<td><code>function_</code></td>
<td>record&lt;Absyn.ComponentRef,CREF_IDENT&gt;</td>
<td>record&lt;Absyn.Componen...</td>
</tr>
<tr>
<td><code>name</code></td>
<td>String</td>
<td>'loadModel'</td>
</tr>
<tr>
<td><code>subscripts</code></td>
<td>list&lt;Any&gt;</td>
<td>&lt;0 item&gt;</td>
</tr>
<tr>
<td><code>functionArgs</code></td>
<td>record&lt;Absyn.FunctionArgs,FUNCTIONARGS&gt;</td>
<td>record&lt;Absyn.FunctionArgs&gt;</td>
</tr>
<tr>
<td><code>semicolon</code></td>
<td>Integer</td>
<td>1</td>
</tr>
<tr>
<td><code>inSymbolTable</code></td>
<td>record&lt;Interactive.SymbolTable,SYMBOLOTA&gt;</td>
<td>record&lt;Interactive.SymbolTable&gt;</td>
</tr>
<tr>
<td><code>ast</code></td>
<td>record&lt;Absyn.Program,PROGRAM&gt;</td>
<td>record&lt;Absyn.Program&gt;</td>
</tr>
<tr>
<td><code>depends</code></td>
<td>record&lt;Absyn.Dep,DEPENDS&gt;</td>
<td>record&lt;Absyn.Dep&gt;</td>
</tr>
<tr>
<td><code>exploredOut</code></td>
<td>Option&lt;Any&gt;</td>
<td>NONE()</td>
</tr>
<tr>
<td><code>instClst</code></td>
<td>list&lt;Any&gt;</td>
<td>&lt;0 item&gt;</td>
</tr>
<tr>
<td><code>lstatVal</code></td>
<td>list&lt;Any&gt;</td>
<td>&lt;0 item&gt;</td>
</tr>
<tr>
<td><code>compiledFunctions</code></td>
<td>list&lt;Any&gt;</td>
<td>&lt;0 item&gt;</td>
</tr>
<tr>
<td><code>loadedFiles</code></td>
<td>list&lt;Any&gt;</td>
<td>&lt;0 item&gt;</td>
</tr>
<tr>
<td><code>stmts</code></td>
<td>record&lt;Interactive.Statements.JSTMTS&gt;</td>
<td>record&lt;Interactive.Stmt&gt;</td>
</tr>
<tr>
<td><code>n</code></td>
<td>Integer</td>
<td>0</td>
</tr>
<tr>
<td><code>p</code></td>
<td>record&lt;Absyn.Program,PROGRAM&gt;</td>
<td>record&lt;Absyn.Program&gt;</td>
</tr>
<tr>
<td><code>st</code></td>
<td>record&lt;Interactive.SymbolTable,SYMBOLOTA&gt;</td>
<td>record&lt;Interactive.SymbolTable&gt;</td>
</tr>
</tbody>
</table>

"loadModel"
changed while stepping then that variable will be highlighted yellow (the standard Eclipse way of showing the change).

5.7 Combining the Algorithmic Code Debugger and Transformations Browser

The algorithmic code debugger (dynamic information) can be integrated with the transformations browser (static information; dynamic information at runtime from the executable messages), although this is not yet performed. It is easier to combine the information if both debuggers run within the same tool, although it also works even if they as currently are provided as separate parts of the OMEdit tool.

If the algorithmic debugger halts on a line of code, there will be a call stack available at this point of the execution. In OpenModelica, each equation has its unique index and a unique function generated for each function. Most expressions evaluated at runtime will have been called from such a function, and for the remaining cases an equation could be generated in order to improve debugging.

By inspecting the call stack in Figure 5.11, it is possible to observe that the function was called from the equation with index=1. It is possible to include a button in the algorithmic debugger that switches to the transformations browser and gives focus to the equations that provide the inputs of the called function.

5.8 Summary and Discussion

Two graphical user interfaces have been created, the transformations browser and the algorithmic code debugger. The transformations browser is integrated with the performance profiler and the capabilities of the tool was demonstrated. The algorithmic code debugger is very similar to traditional instruction-level debuggers. The two tools should eventually be integrated into a single tool.

Currently, the transformations browser is unable to provide graph plots of results from partial and completed simulations. Because all the information needed to show these graphs is available, this can be provided in the future.
Once debugging information for domain-specific debuggers is available, the transformations browser also needs to support display of the information in a user-friendly way.

It would be useful to provide a custom diff algorithm which displays diffs that have some knowledge of how expressions are represented by the tool, making the diff look more natural than the current diff. One way would have been to include diff tags in the operations, but it is probably better to include a lexer and apply a standard diff algorithm to this instead of to the text fragments only.

The profiling parts of the graphical user interface currently only displays measurements for equations. The profiler also logs information for function calls, and the generated HTML reports can display information for these calls. There is a work-around for displaying the information in OMEdit. The work-around is used in Section 5.4 and it is simply to enable the experimental CSE optimization for each function call. This creates one equation for each called function, which means that all function calls in the equation system can be measured individually and displayed in the equation list.
Chapter 6

Compiler Bootstrapping

What connection does compiler bootstrapping (making the compiler compile itself) have to the main topic of this thesis, debugging and performance analysis? It is a somewhat separate but related matter. The debugger, when used only for the current standard Modelica, also works without using the bootstrapped OpenModelica Compiler (OMC). However, the vision is a general debugger and performance analyzer that can handle both the current standard as well as a future Modelica language standard including constructs for language modeling and symbolic transformations such as available in MetaModelica. For this the bootstrapped compiler is necessary. An important achievement, part of bootstrapping, is the development of a general code generation approach to portable C code, supporting efficient execution of large applications as well as efficient debugging and performance analysis of such applications.

This chapter is based on [130] and uses measurements from an older version of OpenModelica (early 2014) which supports compilation using both the bootstrapped OMC and the old MetaModelica Compiler (MMC) implemented in Standard ML. Support for MMC was discontinued after the release of OpenModelica 1.9.1 (fall 2014). The source code of OMC has since been updated in order to use performance-enhancing language features that MMC did not support. The performance increase gained from these changes was quite dramatic in some cases. However, since these language features are not supported by MMC it is impossible to compile and measure the im-
proved OMC using MMC. Comparing an old version of OMC to a more recent one also is unfair because many things were changed under the hood – new optimizations are performed, parts of the compiler were parallelized, and so on. Therefore, the measurements from the original publication have not been updated in this chapter. Section 6.5.1.2 compares the performance of the old and new OMCs for a select few models with comments on why performance improved as much as it did.

6.1 Introduction

The phenomenon of programming languages and compiler bootstrapping is not uncommon. Early examples include Lisp [136] and Pascal [152]. One of the advantages is assumed to be higher quality since the designers and developers of a language and its compiler will be major users, and will therefore be highly motivated to correct possible design flaws and errors. Another advantage is portability – the bootstrapped compiler is primarily dependent on itself, not on other languages, once it has been bootstrapped.

Bootstrapping means that the language along with its compiler is defined and implemented using itself. On a first impression this sounds impossible, especially for a new language, since the language must exist before it is used. The term bootstrapping comes from lifting yourself by your own boot straps, which of course is impossible. However, bootstrapping of compilers is possible. A common approach is to write a subset language compiler in another available language, then rewrite the subset compiler into its own language, extend the compiler to handle the remaining constructs, and finally update the compiler to use the full language.

This chapter describes the first bootstrapping of a full-scale Equation-based Object-Oriented (EOO) modeling language, in this case the Modelica language. Modelica is rather unusual since it supports mathematical modeling with equations, that is, differential, algebraic, and discrete equations. However, such equations are not very suitable for modeling language semantics and symbolic transformations. Modelica was therefore extended with pattern equations, pattern-
matching, as well as tree and list data structures. The extended
language is called MetaModelica (recall Section 2.3).

The bootstrapping of Modelica is a large-scale practical effort
since the compiler currently has about 330,000 lines of code and ap-
proximately ten full-time developers are using MetaModelica on a
daily basis. The standard Modelica language is used by several thou-
sand developers including people developing large industrial applica-
tions, many of which are described in Modelica conference proceedings
[10, 26, 32, 102, 146].

In the case of Modelica, there are at least two motivations for
bootstrapping in addition to those mentioned previously. One mo-
tivation is a scientific experiment in language design with the goal
of attaining a generalized equation-based language suitable for both
physical system modeling and language modeling [115].

A second motivation comes from the growth of the Modelica lan-
guage itself, and the corresponding increase of complexity and size of
its compilers. This is caused by steadily increasing user application
requirements on the language including widening of its application
domains.

One approach to manage such increasing language and compiler
complexity is the language core library approach used by several func-
tional languages (Section 6.7). A number of language features are
defined in core libraries rather than in the compiler itself.

After several years of discussion and a number of language pro-
totypes [20, 45, 56, 97, 113, 115, 155], this approach finally became
accepted as a design goal for future Modelica versions during the 67th
Modelica design meeting [92].

This chapter gives a rather complete account of the Modelica boot-
strapping effort in the OpenModelica compiler project of which an
early preliminary description is available [129].

6.1.1 Specification of Language Constructs

The Modelica specification and modeling language was originally de-
developed as an object-oriented declarative equation-based specification
formalism for mathematical modeling of complex systems, in par-
icular physical systems.
However, it turns out that with some minor extensions, the Modelica language is also well suited for another modeling task, namely modeling of the semantics, that is, the meaning, of programming language constructs. Since modeling of programming languages is often known as meta-modeling, the name MetaModelica [54, 56, 58, 115] is used for this slightly extended Modelica.

Well-known language specification formalisms such as Structured Operational Semantics and Natural Semantics [108, 110] also have the property of being declarative equation-based formalisms. These formalisms fit well into the style of the Modelica specification language, which explains why Modelica with some minor extensions is well suited as a language specification formalism. In fact, structured operational semantics were used for specifying early versions of Modelica, using the Relational Meta-Language (RML) tool to generate compilers, see Section 6.2.2. However, only an extended subset of Modelica, referred to here as MetaModelica, is needed for language specification. Many parts of the language designed for physical system modeling are not used at all for language specification.

Another great benefit of using and extending Modelica in this direction is that the language becomes suitable for meta-programming and meta-modeling. This means that Modelica can be used for specification and transformation of models and programs, including transforming and combining Modelica models into other (lower-level) Modelica models, that is, a kind of compilation.

6.2 Vision – Extensible Tools

Traditionally, a model compiler performs the task of translating a model into executable code, which subsequently is executed during simulation of the model. The symbolic translation step is thus followed by an execution step, a simulation, which often involves large-scale numeric computations.

However, as requirements on the usage of models grow, and the scope of modeling domains increases, the demands on the modeling language and corresponding tools increase. This causes the model compiler to become large and complex.
6.2. VISION – EXTENSIBLE TOOLS

Moreover, the modeling community needs not just tools for simulation but also languages and tools to create, query, manipulate, and compose equation-based models. Additional examples are optimization of models, parallelization of models, checking and configuration of models.

If all this functionality is added to the model compiler, it tends to become large and complex.

An alternative idea already mentioned in Section 6.1 is to add features to the modeling language defined in library packages that can contain model analysis and translation features that therefore are not required in the model compiler. An example is a Partial Differential Equation (PDE) discretization scheme that could be expressed in the modeling language itself as part of a PDE package instead of being added internally to the model compiler.

6.2.1 Motivation for Compiler Bootstrapping

As mentioned in the introduction, bootstrapping means that a compiler can compile itself. Why is this relevant in the context of language specification and language semantics modeling? The most important factor is probably that the language itself can be used as a large test case for language specification using the developed specification language. This has mostly advantages, but also a disadvantage:

- The implemented language becomes well tested, since the developers are using it on a large application (the compiler).
- The developers are motivated to conduct a high quality implementation, since they are using it themselves.
- The developers are motivated to create a good development environment, since they are using it themselves.
- There is also a negative factor: Because the tool must be able to build itself, more effort is needed to create it since the implementation must be good enough to be usable. Moreover, initially a new language and environment may lack some qualities and tools available in existing languages. For example, during the
CHAPTER 6. COMPILER BOOTSTRAPPING

first years of using MetaModelica there was no good debugger available.

6.2.2 Stages of Bootstrapping OMC

Bootstrapping of the OpenModelica Compiler (OMC) has been an 8-year effort, consisting of the following stages:


2. Implementation of the MetaModelica Compiler (MMC) by adding a new compiler frontend to the old RML compiler [57, 108], translating MetaModelica into RML intermediate form, spring-fall 2005.

3. Automatically translating the whole OpenModelica compiler, 60,000 lines, from RML to MetaModelica.

4. In parallel, developing a new Eclipse plugin, MDT, for Modelica and MetaModelica [116, 119], including both browsing, debugging, semantic context-sensitive information, etc., 2005-2006.

5. Initial use of this MetaModelica 1.0 (an extended subset of Modelica), the MMC compiler, and the new MDT Eclipse plugin for the OpenModelica compiler development, at that time 3-4 full-time developers. This version 1.0 of MetaModelica is described in [46, 55]. Fall 2006.


7. Continuation of the work on better support for pattern-matching compilation, support for lists, tuples, records, etc. in OpenModelica. This was part of the meta-modeling support in the OMC Java interface that was implemented in the spring and fall, 2009 [123].
8. Implementation of higher order functions as used in MetaModelica [21]. Higher order functions also became part of the Modelica language specification [94]. Fall 2009, spring 2010.

9. The current work on finalizing the bootstrapping reported in this chapter. The bootstrapped compiler supports MetaModelica 2.0, which includes both standard Modelica as well as further improved MetaModelica extensions aiming at becoming future Modelica extensions. Fall 2010, spring 2011.

10. Further addition, enhancement, and redesign of MetaModelica language features [58] were based on usage experience, the Modelica design effort, and inspiration from functional languages and languages such as Scala [101]. Refactoring parts of the compiler to use the enhanced features.


12. Improving the build system, parallel builds. Reaching full test suite coverage, good performance, and running the tests nightly. 2013.

13. Release of the original article that this chapter is based on [130], Spring 2014.

14. Dependency analysis was added to the build system which now supports separate compilation, that is, it only recompiles a module if the interface of a module it depends on has changed. Spring 2014.

15. MMC support was discontinued after the release of OpenModelica 1.9.1. New language extensions were used throughout the compiler. Fall 2014.

6.3 Implementation

This section provides some information about the implementation of the compiler and the runtime system for the MetaModelica extensions.
CHAPTER 6. COMPILER BOOTSTRAPPING

<table>
<thead>
<tr>
<th>Compiler phase</th>
<th>Files</th>
<th>Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constants</td>
<td>1</td>
<td>27</td>
</tr>
<tr>
<td>Entry-point</td>
<td>1</td>
<td>902</td>
</tr>
<tr>
<td>Scripting environment</td>
<td>6</td>
<td>26,788</td>
</tr>
<tr>
<td>Utility functions</td>
<td>48</td>
<td>24,202</td>
</tr>
<tr>
<td>FrontEnd (up to flat Modelica)</td>
<td>106</td>
<td>165,569</td>
</tr>
<tr>
<td>BackEnd (from flat Modelica to sorted eq.syst.)</td>
<td>46</td>
<td>96,372</td>
</tr>
<tr>
<td>Code generation (hand-written Susan runtime)</td>
<td>5</td>
<td>13,816</td>
</tr>
<tr>
<td>Code generation (Susan template source code)</td>
<td>21</td>
<td>36,624</td>
</tr>
<tr>
<td>Code generation (generated MetaModelica code)</td>
<td>21</td>
<td>211,088</td>
</tr>
<tr>
<td>Total size (excl. generated code)</td>
<td>213</td>
<td>327,676</td>
</tr>
</tbody>
</table>

Table 6.1: Sizes of OMC compiler phases, lines of code.

6.3.1 Compiler

OMC is implemented mostly in MetaModelica. The compiler frontend currently comprises almost half of the source code. Here the frontend is defined as the compiler phases that convert the source model into a flattened intermediate form consisting of equations, variables and functions. It elaborates Modelica and MetaModelica code into so-called flattened intermediate code, which is passed on to the backend equation optimizer (for simulation models) or directly to the code generator (for functions).

The number of lines of MetaModelica code used to implement OpenModelica are in Table 6.1. Empty lines and C-style comments have been removed, but Modelica-style string comments are included. The OpenModelica text generation template language Susan described in [60] is used in the compiler to describe code generation templates in the code generation module. These templates are translated by a compiler module into MetaModelica source code including automatically using efficient string builders since building strings incorrectly is very inefficient. It is also easier to understand and edit the text template source code than optimized MetaModelica source code.

OpenModelica also requires a runtime system. It is divided into five parts:

- The basic runtime system contains all Modelica built-in function definitions, such as `String()` or `div()` and handles array
6.3. IMPLEMENTATION

operations. This runtime needs to be linked to the compiler itself and any source code it produces.

- The simulation runtime system contains the numerical solvers and also performs event handling.

- The compiler runtime system handles runtime options, settings, CORBA [100] communication between OMC and clients, as well as system calls.

- The parser sources are currently generated by the ANTLR [106] parser generator. This tool produces code that can be called through the foreign function interface of MetaModelica (Section 6.3.4.2). When run, the parser produces a MetaModelica abstract syntax tree that the compiler can use directly. An alternative approach would be to use a parser generator that produces MetaModelica code instead, analogous to MLLex [4] and MLYacc [137] for the SML language. A recent prototype of such a parser generator is OMCCp [81].

- The built-in environment of predefined functions, types, and constants is specified by a Modelica file that contains a list of all built-in functions in the Modelica language as well as the scripting functions that are available in OpenModelica.

The MMC foreign function interface is completely different from the OMC interface, which means they cannot use the same code directly. Most of the runtime is written as an implementation conforming to the OMC interface and an MMC wrapper around this, as shown below:

```c
RML_BEGIN_LABEL(Lapack__dorgqr)
{
    void *A, *WORK;
    int INFO;
    LapackImpl__dorgqr(RML_UNTAGFIXNUM(rmlA0), ...);
    rmlA0 = A;
    rmlA1 = WORK;
    rmlA2 = (void*)RML_TAGFIXNUM((long)INFO);
    RML_TAILCALLK(rmlSC); /* Success continuation */
}
RML_END_LABEL;
```
CHAPTER 6. COMPILER BOOTSTRAPPING

<table>
<thead>
<tr>
<th>Supporting functionality</th>
<th>Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parser (ANTLR sources)</td>
<td>1,968</td>
</tr>
<tr>
<td>Parser (generated C sources)</td>
<td>49,256</td>
</tr>
<tr>
<td>Parser (wrapper code)</td>
<td>339</td>
</tr>
<tr>
<td>Compiler runtime</td>
<td>6,159</td>
</tr>
<tr>
<td>Simulation runtime</td>
<td>3,405</td>
</tr>
<tr>
<td>Basic runtime (excl. MetaModelica and External)</td>
<td>9,675</td>
</tr>
<tr>
<td>Basic runtime (third party code)</td>
<td>13,268</td>
</tr>
<tr>
<td>Basic runtime (MetaModelica)</td>
<td>1,930</td>
</tr>
<tr>
<td>Modelica built-in environment</td>
<td>744</td>
</tr>
<tr>
<td>MetaModelica built-in environment</td>
<td>825</td>
</tr>
<tr>
<td><strong>Total size (excl. generated and external code)</strong></td>
<td><strong>25,045</strong></td>
</tr>
</tbody>
</table>

Table 6.2: Sizes of OMC parser and runtime.

The lines of code listed in Table 6.2 exclude these MMC wrapper functions. Header files are included in these measurements, but comments and blank lines are not counted in either headers or source code.

6.3.2 Platform Availability

OMC runs on all the major platforms, including Windows, Mac OS X, and a number of Linux variants. With the bootstrapped compiler it is now also possible to compile on platforms without a Standard ML compiler, which used to be a limiting factor in porting OpenModelica to new operating systems.

6.3.3 Language Feature Implementations

This section details the challenges faced when the MetaModelica language extensions were implemented.

6.3.3.1 Pattern Matching Implementation

The matchcontinue construct was implemented using the continuation passing style in the old MMC compiler. This is the same as in the RML language on which it was based [109]. It provides excep-
tion handling without additional cost. The main drawback is the lack of a regular C stack. The lack of a regular C stack precludes some optimizations typically performed by C compilers, and also makes it impossible to use standard performance analyzers and debuggers.

There have been three main attempts to add support for this language feature in OMC. These attempts all used a regular C stack and exception handling to model matchcontinue.

The first approach [135] introduced C-like constructs (like goto) to the intermediate representation so that it could be mapped to the runtime at an early stage. It created a Deterministic Finite Automaton (DFA), a deterministic state machine, so that results of pattern-matching in earlier cases could be remembered when matching later cases. But the DFA was unable to handle all of the aspects of the matchcontinue semantics. The code that generated the DFA was too hard to maintain, and did not produce faster code, only slightly smaller code size in some cases. This part was therefore removed in the second approach.

Removing the DFA solved the issues of match and matchcontinue producing unexpected results. It also made it possible to nest match-expressions. But there were still flaws in the implementation. Each match-expression needed to be a statement, and the inputs of the match were required to be variable names. While this is the same restriction as in the MMC implementation, the language design requires expressions to be more general.

One of the problems with the implementation was that it elaborated the expression twice (from a typed expression back to untyped, and then typed again so that it could add temporary variables with the correct type name). Trying to add special code to allow for empty lists proved futile as its internal type (list of Any) cannot and should not be possible to express in the MetaModelica abstract syntax.

The start of the final attempt was the implementation of pattern-matching assignments, for example:

\[(a,1.0) := \text{fnCall}(\ldots)\]

Previously, pattern-matching assignments were handled by being converted to nested matchcontinue blocks. This did not always work as types of temporary variables could not always be expressed in Meta-
Modelica abstract syntax. The new implementation has the concept of *patterns* in addition to *expressions*. They are treated differently and elaboration is now performed at a single step without converting an elaborated expression back to abstract syntax. Generation of temporary variables is performed by the code generator. This allows a complete implementation of *matchcontinue* without any major issues, and also allows useful error messages when elaborating invalid patterns.

Since *match* expressions are now implemented as expressions instead of as a specific kind of assignment, they can now be used as ordinary expressions. Since they have result types, they may even serve as input for another *match* expression, for example:

```
match (match str
    case "Modelica" then true; else false;
    end match)
  case true then 3.0;
  else 2.0;
end match; /* if str=="Modelica" then 3.0 else 2.0 */
```

This approach is much simpler to maintain because it works at the correct level of abstraction, that is, elaborated expressions. The compiler has access to the full type information while performing the translation.

While this works correctly for all cases, it might be a bit slower than the first implementation in some cases. It no longer has a DFA to avoid pattern-matching the same thing several times. However, because the code for pattern-matching is essentially created during code-generation instead of during elaboration, it enabled switching from a C++ runtime to C (see Section 6.5). For example, if the target language includes *match*-expressions, it is possible that its compiler could optimize such constructs – this would be true if a functional language was the target language. Optimizations for pattern-matching have been implemented directly in OMC. For example, dead code elimination (unreachable patterns) and detection of expressions where a case can be directly selected instead of performing a linear search of all patterns (similar to *switch* in C, but for more data types). It
6.3. IMPLEMENTATION

is also possible to generate more low-level code (like a DFA) based on this structure. However, experience has shown that OMC currently generates sufficiently fast code directly from the intermediate code without allocating registers, creating temporary variables and so on. This is partly due to doing simple things such as resetting the temporary variable index between each statement in the function (which will enable some reuse of temporary variables).

6.3.3.2 Type Design and Implementations

Adding support for the Array, Option, List and Tuple types to a Modelica compiler is not an easy task due to subtle incompatibilities in the type systems of MetaModelica and Modelica.

Array expressions sometimes need to be type converted to lists, but there are more expressions than the array data constructor that elaborates to the same expression. For example, the \texttt{fill(3,2)} operator call produces an array expression \{3,3\}, which can then be cast to a list. The MetaModelica 2.0 design [58] solves these issues.

The Option type was easier to implement than the other types introduced in MetaModelica. The reason is that its syntax does not overlap with arrays (like lists), or multiple outputs of functions (like tuples).

Tuples are treated differently in the compiler and the runtime depending on whether they are multiple outputs of a function or a tuple. A function that returns \texttt{Tuple<Integer,Integer>} has a different type than one that returns two integers. This is different from most functional programming languages, but is consistent with the Modelica design as well as the design of the RML specification language.

The union type syntax looks like it would be accessed by \texttt{Package.Uniontype.RECORD}, but in MMC the record is implicitly added to the package scope so that it is accessed as \texttt{Package.RECORD}. In order to bootstrap the compiler, the same principle is used in OpenModelica. The type of a call to a record constructor is the union type, not the record type itself, which means that accessing fields from \texttt{uniontype} records is currently limited to pattern-matching.
6.3.3.3 Polymorphism

There could be several different ways to implement polymorphic functions in a Modelica compiler. Hindley-Milner-style type inference [65, 90] is used to infer the type of a function based on its interface.

That is, only inputs and outputs of functions are considered when conducting type inference. From the inputs of a call expression, a set of constraints is created and subsequently unified. If unification succeeds, the actual types of the type variables have been determined and are used to calculate the result type of the call. All other variables have a type and no type inference is performed for local variables.

Note that neither Modelica nor MetaModelica has the concept of a general Number type (some Modelica built-in operators take either Integer or Real as input, but this cannot be represented in Modelica code). Because there is no Number type, a call to valueEq(1,1.5) would not pass type checking because the types are different.

Even though implementing polymorphic functions in a Modelica compiler is rather straightforward, there are certain pitfalls. The names of type variables can be from the current scope, from the called function or from a function pointer used as an input argument. Keeping track of the different sources is the key to getting the correct semantics.

6.3.4 Runtime System

Because the OMC runtime initially only handled the static Modelica structures, the runtime system was extended to handle the new functional constructs.

6.3.4.1 Data Layout

The same layout of values (objects) in memory as for the MMC compiler runtime was used with some enhancements for faster debugging (see Section 6.6). Basically all values besides integers are boxes that contain a header followed by the actual data. The pointers are tagged (the small number 3 is added to them, and the allocated pointers are aligned at even 4-byte boundaries) to be able to differentiate between pointers and other values such as integers. This difference was needed
for the garbage collection implementation as used in MMC. It is now
instead needed in order to provide type information in the debugger.
The headers have 32 bits or 64 bits depending on the platform. Inside
the header the bits are split into:

- Slots – represent the size of data in words
- A constructor – represents the type of the data (that is, index
  in a uniontype or string, real, etc)

On 32-bit platforms the slots are 22 bits and the constructor (the tag)
is 10 bits. On 64-bit platforms the slots are 54 bits and 10 bits are
devoted to the constructor (the tag).

Integers are either 31 or 63 bits depending on the platforms and
are represented by even values; integer N is shifted left one bit (mul-
tiplied by two).

Bit zero is set if the box (node) contains pointers and unset other-
wise, or the other way around depending on which garbage collection
characteristics are desired.

A List value is represented as a box containing a CONS header, a
pointer to the element and a pointer to next. The end of the list is
represented by a box containing a NIL header with zero slots.

An Option value is represented as a box containing a SOME header
and a pointer to the element or a NONE header with zero slots.

There is a header that contains macros to access the desired fields
without requiring knowledge of which bits to read, taking care of cross
platform issues.

6.3.4.2 Foreign Function Interface

Modelica has support for calling external "C" functions with a map-
ing of Modelica types to C types. The types introduced in Meta-
Modelica (List, Tuple, Option, Array, and uniontype) all map to
void*. The macros used to access and create these types are different
for MMC and OMC, but a compatibility header exists so code can
be shared between the two implementations. One major difference
between MMC and OMC implementation is that within an external
function, variables may have been moved by the garbage collector after calling another function in the MMC implementation. In OMC, calling other functions is considered safe.

6.3.4.3 Built-in MetaModelica Functions

New built-in functions are needed to perform operations on the MetaModelica boxed values. The functions can be used directly in the MetaModelica code or they are generated in the C code from operators. These functions are classified based on the types they operate on. These are some of the available built-in functions:

- **Booleans**: boolEq, boolString

- **Integers**: intEq, intLt, intLte, intGt, intGe, intNe, intString, intAdd, intSub, intMul, intDiv, intMod, intMin, intMax

- **Reals**: realString, realEq, realLt, realLe, realGt, realGe, realNe, realAdd, realSub, realMul, realDiv, realMod, realMin, realMax

- **Strings**: stringEq, stringCompare, stringHash, stringGet, stringAppend

- **Lists**: listAppend, listMember, listGet, listReverse, cons (:: operator), stringAppendList

- **MetaArrays**: arrayCreate, arrayGet ([x] index operator), arrayCopy, arrayUpdate

6.3.4.4 Garbage Collection

Garbage Collection (GC), that is, automatic memory reclamation, of unused heap-allocated data is a must for a functional language to be able to collect unused values and reuse memory. The garbage collector used in MMC is a simple 2-generational copying compacting garbage collector [108, 151]. It has two main memory areas: a young generation where the objects were initially allocated and an older region to which the objects are promoted if they survive a minor
collection. This old MMC garbage collector could not be directly reused for the OMC runtime because it moves pointers and makes it hard to write external functions.

The interface for the garbage collector in OMC is currently simple in order to make it easier to test different strategies. The garbage collector in the remainder of the chapter is the Boehm-Demers-Weiser conservative mark-and-sweep garbage collector [15, 16]. It is simply plugged into OMC-generated code and collects garbage by analyzing the stack, searching for live objects to free.

6.3.5 Issues

There are several problems with the MMC and OMC as well as design issues in the Modelica language that needed to be worked around in order to bootstrap OMC.

6.3.5.1 MMC Problems

The old MMC compiler has several problems. While it provides good performance, maintaining the code is cumbersome. The OpenModelica compiler developers prefer having only one tool to maintain (OMC). Before OpenModelica could switch to using OMC as the default MetaModelica compiler, OMC needed to be able to compile the same code as MMC during a transition period. The problem is that the MetaModelica to RML translator is very relaxed regarding the syntax it accepts.

In order to detect certain common errors, the entire abstract syntax of a function is traversed to verify that all matchcontinue expressions have the same input and output as the function. This is a limitation of MMC, but it does not actually check that this assertion holds. Moreover, while MMC does perform some type checking, it does not type-check expressions of the following kind:

\[ x = \text{fnCall}(x); \]

For these expressions, the left side \( x \) and right side \( x \) may have different types and MMC may allow some code that is not valid MetaModelica code. Finding such code and rewriting it takes considerable
time, but produces code that is easier to read and maintain. Other issues include allowing code like:

```plaintext
import Env;
type Env = Env.Env;
```

In the standard Modelica language the import will essentially be ignored and the tool will probably get a stack overflow because the type `Env` is recursive on itself. Since the same restrictions apply for OMC, these language constructs were refactored away.

### 6.3.5.2 OpenModelica Issues

Some parts of the compiler have been rewritten so that it became easier to detect common errors that MMC does not handle. As a result, OMC now has vastly better error messages for algorithmic code. OMC has also been enhanced to propagate file, line and column information to more error messages because when there is a large application that needs to be corrected, it is essential that the correct line is found quickly. Improving the error messages probably saved more time than it cost to implement and as a result, the compiler is now producing better error messages both for Modelica and MetaModelica code.

OpenModelica has problems with shadowing of certain built-in operators. For example, creating a function `ndims` and calling it in the same package will result in the Modelica built-in `ndims` operator being used instead of this function. A few of these functions existed in the compiler and had to be renamed – rewriting lookup in OpenModelica is an alternative, but would take more time.

### 6.3.5.3 Modelica Problems

The Modelica specification [94] states that external functions map `Integer` to `int`, with no way to change their behavior. This is a problem if external functions in the compiler runtime need to access large integers. An example of such a function is `referenceEqual`, which checks whether two pointers are equal and the `stringHash` family of functions. Values were being truncated because of the limited precision of `int` on (some) 64-bit platforms, so functions were moved from
external C functions in the compiler into the MetaModelica language itself as external built-in functions.

6.4 Separate Compilation

Note that this section was added after publishing the original article [130]. Other sections refer to an older version of the build system.

Separate compilation is a useful feature in a build system. By treating the full program as consisting of separate modules and compiling the modules separately, much time can be saved by avoiding re-compiling unchanged parts when making small changes to the program.

6.4.1 Implementation

The OMC source code is stored as a set of files (each top-level Modelica package is stored in a separate file), for example Absyn.mo and Main.mo. The described build system provides separate compilation for the MetaModelica language, which is a superset of Modelica. The method will also work in the same way for Modelica code without any changes. Here only compilation of functions (and constants) in packages is considered. There is currently no separate compilation of equation-based simulation models.

Parsing the entire source code of the OpenModelica compiler in a single process takes several seconds on a laptop. This is too long to wait for simply parsing code that is going to be re-compiled. Parsing text should be a small part of the total compilation time, and with the provided implementation the compilation time is shorter than the time to parse all files in one process when making a small change to the compiler. This includes analysis, code generation, compilation, and linking.

The hardest part about separate compilation of Modelica code is that essentially any function can depend on any part of the total program. This means that to compile any package separately, the entire program needs to be parsed in order to create the dependencies.

By placing the restriction that all top-level packages are encapsulated and breaking the encapsulation is only allowed by importing a
package, all dependencies of a module are thus marked by an import statement. This improves performance in subsequent steps since the compiler knows which files are reachable from the given package. The restriction does not suffice since all code is reachable from the main function.

An additional useful restriction is that public parts of any public function, constant, or type may only refer to other public elements. By introducing this restriction, it is possible to create an interface file for each package that strips out protected elements and algorithm sections without breaking lookup rules. Everything that remains in the resulting file is part of the interface, and loading the complete interface file takes roughly $1/10$ as much time as loading the full source files.

When calculating the list of interface files that a package depends on, start with adding all the import elements in the package that will be compiled for the work list. Neither of the packages in the work list contain protected imports since they are interface files. Iterate the work list, loading each package and adding any imported packages to the work list (ignoring any item already added). The result is the list of all interfaces needed to calculate all types used in functions of the package that will be compiled.

### 6.4.2 Steps in detail

These are the basic steps performed by the compilation process. Note that there are no problems with running processes in parallel since files are read by multiple workers, but no process updates the files used in the given step. The only synchronization needed is between the steps. These steps are explicit in the make system and will be executed in the order shown below:

- Create Makefile targets needed to compile files (at first run only, and when adding new files to the compiler).

- Whenever a file is updated, also update its interface file (run in parallel, parses only the file that was updated).
• If the interface of a file changed, load all interface files and write out the dependencies of the program to a Makefile.

• Translate files from MetaModelica to C code. Loads one mo-file and a set of interface files. Only changed files or files that depend on an updated interface file are translated. This step can be run in parallel and parses a minimal set of small interface files.

• Compile C-files to object code (in parallel). Furthermore, if the C code did not change after translation, the old object code is used instead of running the C compiler again.

• Run the linker and add the main driver that initializes the executable and calls the MetaModelica function `Main.main`.

6.4.3 Performance

The tests were performed using a 4-core Intel Core i7 3820 @3.60 GHz system with 32 GB RAM and an SSD. There were 250 packages in the compiler sources. The performance for the different steps is shown below:

• Creating the interface file of a package takes between 0.1 and 0.6 seconds depending on file size. Typical duration is 0.2 seconds for a large 13,000 line source file.

• Updating all dependencies takes 0.35 seconds and this step is only sometimes run.

• Compiling from .mo to .c takes 2.78 seconds for a 13,000 line source file and 8.7 seconds for slowest file.

• Compiling .c to .o takes 4 seconds for a 13,000 line source file and 9.7 seconds for the slowest file.

Total time, including linking when updating a file without changing its public interface, is shown in Table 6.3.

Compilation time without separate compilation (using the build system from Section 6.5.1) takes 2m50s regardless if compiling from
Table 6.3: Total compile-times of files with different file sizes.

<table>
<thead>
<tr>
<th>File</th>
<th>Lines</th>
<th>Time</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absyn.mo</td>
<td>6,000</td>
<td>4.5s</td>
<td>Typical file size</td>
</tr>
<tr>
<td>Static.mo</td>
<td>13,000</td>
<td>8.6s</td>
<td>Large file</td>
</tr>
<tr>
<td>CodegenC.mo</td>
<td>66,000</td>
<td>21.1s</td>
<td>Generated code</td>
</tr>
</tbody>
</table>

scratch or with a change in only a single file. With separate compilation, total compilation time is 2m27s from scratch or as shown in Table 6.3, between 4.5 and 21.1s for changes in only a single file. The new build script using separate compilation is always faster. For developers making many small changes to the code every day, it presents considerable time savings.

Splitting files into multiple packages improves separate compilation performance and often makes it easier to find functions in code.

### 6.5 Performance

The compile-time performance\(^1\) of the compiler was not fully satisfactory at the time of writing this section of the text\(^2\), although it had about the same performance as the old compiler (MMC) despite the garbage collector using up a large part of the total execution time.

The compilation time performance of the working examples varied widely in the first versions of the bootstrapped compiler. Some tests could be substantially improved by making small changes.

In the first executable version of the bootstrapped compiler even simple test cases were nine times slower to compile than in the MMC version. Since the new version of OMC uses a real C stack, it is possible to use general-purpose debugging and profiling tools, such as the GNU project debugger (GDB), gprof or valgrind on it. It is also possible to use OpenModelica’s own MetaModelica debugger on

---

\(^1\)Everything OMC does from parsing a file until it generates C code for a model. Running a C compiler on the source code and running the executable takes the same amount of time regardless of whether MMC or OMC is used.

\(^2\)The measurements were performed in 2014 and compare the performance of the old OMC to MMC [130]. The bootstrapped compiler performance is now satisfactory; see Section 6.5.1.2.
To improve performance, a standard tool was used to profile a simple example: `valgrind --tool=callgrind`. The PEXPipe model was chosen because it took 1-2 seconds to run, making it of suitable size (profiling adds 10-100 times overhead). The model spent about 80% of its time on exception handling. The C++ try-catch feature had been used to implement exception handling [119] of `matchcontinue` expressions. C++ exceptions are slow because they are not supposed to be used all that often. However, in MetaModelica 1.0, `matchcontinue` (that is, match with possible backtracking and continuation of the next matching case) is used as the basic general control flow construct instead of special purpose constructs such as `if`-statements (which MMC does not support). The exception handling was rewritten using the cheaper `setjmp/longjmp` operations for inter-procedure jumps and `goto` operations for intra-procedure jumps. Since the OpenModelica runtime does not use reference counting\(^3\), C++ classes or C++-heap-allocated data, this level of exception handling is sufficient.

The new exception handling resulted in a tenfold speedup for certain examples (on the average threefold). Exception handling still uses approximately 25% of the compiler execution time. OMC now automatically optimizes `matchcontinue` expressions that are known to not use backtracking. This removed some of the `setjmp` overhead.

### 6.5.1 Benchmarks

All benchmarks were performed using a desktop computer running 64-bit Ubuntu 13.10. The machine was equipped with a 6-core Intel Core i7-3930K CPU @ 3.20 GHz system and 32 GB of memory. Both compilers (OMC and MMC) were taken from the standard OpenModelica package repository, compiled with default settings\(^4\).

The two implementations use virtually the same parser, the only difference is in the garbage collector. The Modelica Standard Library\(^3\) `longjmp` does not unwind the stack so using reference counting becomes hard to use.\(^4\) OMC revision r19074, using repository http://build.openmodelica.org/apt/saucy/nightly
(MSL) [93] is distributed in multiple files that are parsed and merged into a single tree by OMC. Version 3.2.1 of the MSL was used, which is 10.9 MB of Modelica text when stored as a single file.

Unparsing the tree is the process of going from the internal abstract syntax tree structure back to Modelica concrete syntax (source code). In Table 6.4, it is shown that the bootstrapped implementation is slightly faster at parsing code. This is due to the MMC implementation having slower memory management for external C code – the data is allocated in a special buffer which is then copied to the normal heap in an extra phase. The OMC implementation uses the Boehm GC [15, 16] as its garbage collector, which allows for using a parallel mark phase. The only disadvantage is parallel marking increases the total work of the thread due to thread synchronization, which means it should only be enabled if the CPU would be idle without it.

Having the option to use parallel marking is advantageous for most users of the compiler since they will typically work on only one model at a time. Compiler developers on the other hand will run enough tests at a time to fully utilize over 40 threads without parallel marking.

A few known large models were chosen to test the scalability of the compiler for large tests run as a single process. The tests that take the longest to execute are simulation tests, code is generated, compiled, run, and the results are verified. However, most of the time is spent running a C compiler on generated code that is the same in both compilers. Thus, the chosen tests are large models that are not simulated.

The HumMod model [73] has more than 28,083 equations and the V6 engine from MSL [93] contains 9,016 equations. What the results in Table 6.5 tells us is that large models have a higher overhead than

<table>
<thead>
<tr>
<th>Task</th>
<th>MMC</th>
<th>OMC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>serial mark</td>
<td>parallel mark</td>
</tr>
<tr>
<td>Single file</td>
<td>1.8s 0.96 GB</td>
<td>1.74s 1 GB</td>
</tr>
<tr>
<td>Multiple files</td>
<td>1.71s 0.22 GB</td>
<td>1.52s 0.23 GB</td>
</tr>
<tr>
<td>Parse, unpars</td>
<td>2.89s 0.37 GB</td>
<td>3.32s 0.53 GB</td>
</tr>
</tbody>
</table>

Table 6.4: Compiler performance, parsing of MSL 3.2.1.
6.5. PERFORMANCE

Table 6.5: Compiler performance, checkModel() on the large HumMod and analytic EngineV6 models, time spent and memory consumption.

<table>
<thead>
<tr>
<th>Task</th>
<th>MMC</th>
<th>OMC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>serial mark</td>
<td>parallel mark</td>
</tr>
<tr>
<td>HumMod</td>
<td>92.21s 0.76 GB</td>
<td>134.6s 1.1 GB</td>
</tr>
<tr>
<td>EngineV6</td>
<td>6.85s 0.43 GB</td>
<td>11.27s 0.76 GB</td>
</tr>
</tbody>
</table>

Table 6.6: Compiler performance, OpenModelica test suite, run in parallel. The fast version of the test suite is comprised of 1,622 simple tests, with the full version having 2,499 tests.

<table>
<thead>
<tr>
<th>Tests</th>
<th>MMC</th>
<th>OMC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>serial mark</td>
<td>parallel mark</td>
</tr>
<tr>
<td>Fast</td>
<td>126.64s 0.5 GB</td>
<td>142.66s 0.8 GB</td>
</tr>
<tr>
<td>All</td>
<td>1076.7s 2.5 GB</td>
<td>1291.5s 4.5 GB</td>
</tr>
</tbody>
</table>

small models compared to MMC. While OMC performs well overall, the parallel mark phase speeds up performance considerably which suggests that memory management can be further improved and that it will suffer on throughput of parallel tests where parallel mark is not a good option.

Running the entire OpenModelica test suite is another important application for developers. The results in Table 6.6 are rather good. For users who run only a single model at a time, parallel marking will improve performance roughly to the average speed of MMC. While the results show that MMC is faster on throughput, this is because a few functions use algorithms that scale poorly with the bootstrapped compiler implementation. These issues were resolved once the compiler sources started using the new language features that only the bootstrapped OMC is capable of handling.

There is a difference in how the two tool chains create the executable compiler, but it is currently quite similar. Both tool chains analyze the MetaModelica source code to build the dependencies. OMC is more conservative and always succeeds while MMC uses a more optimistic approach and does not always recompile required files. Both OMC and MMC compile files in parallel since it is impor-
Table 6.7: Running the new build script for compiling OpenModelica.

<table>
<thead>
<tr>
<th>Task</th>
<th>MMC serial mark</th>
<th>OMC serial mark</th>
<th>OMC parallel mark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generate code</td>
<td>44.58s 1.1 GB</td>
<td>44s 1.3 GB</td>
<td>43.05s 1.3 GB</td>
</tr>
</tbody>
</table>

Table 6.8: Compiling the OpenModelica source code with the old build system (MMC) and the new one (OMC).

Table 6.8: Compiling the OpenModelica source code with the old build system (MMC) and the new one (OMC).

<table>
<thead>
<tr>
<th>Task</th>
<th>MMC</th>
<th>OMC</th>
<th>Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Translating MetaModelica to C</td>
<td>135.2s</td>
<td>43.05s</td>
<td>0.32x</td>
</tr>
<tr>
<td>Compiling C-files using gcc 4.8 -O2</td>
<td>423.8s</td>
<td>44.55s</td>
<td>0.11x</td>
</tr>
<tr>
<td>Compiling C-files using clang 3.3 -O2</td>
<td>36.13s</td>
<td>36.04s</td>
<td>1x</td>
</tr>
</tbody>
</table>

Table 6.7 shows the performance of the OMC approach to compiling the compiler using an older build system than the one described above and in Section 6.4. The main difference is that dependency analysis is not performed and since OMC is compiled from scratch, the dependencies do not matter. Not surprisingly, the performance figures of MMC and OMC are very close since profilers were run on the bootstrapped compiler to maximize performance on this benchmark.

Comparing the MMC and OMC build systems in Table 6.8, OMC shows better performance. GCC versions since 4.5 have problems handling the huge volume of C code generated by MMC. GCC has no problems with code generated by OMC however, giving OMC an advantage for compiler developers preferring GNU Compiler Collection (GCC) over the LLVM-based clang.

6.5.1.1 Comparison with Old Benchmarks

Compared to preliminary work performed in [129], the numbers reported here reflect a number of updates made in OpenModelica since 2011. Most important is the addition of garbage collection. The old implementation simply allocated a huge chunk of memory and never deallocated it. As such, a number of larger test cases could not run and were removed from the comparison. Now all test cases run
and produce identical results. For some test cases, garbage collection takes close to 50% of total runtime, which means performance has also improved since 2011 despite the numbers at that time indicating MMC and OMC were closer in 2011.

Furthermore, the tests can now be run in parallel since they use much less memory. The 2011 build system for the bootstrapped compiler was always run in serial, utilizing a single CPU. In this version, parallel runs of the test suite are compared for both MMC and OMC.

There is also a comparison on parallel garbage collection using parallel marker threads.

Judging by these changes, the current version of the bootstrapped compiler is much more mature than the 2011 version.

### 6.5.1.2 Comparison with Latest OMC Version

To compare the latest OMC version to the most recent benchmarked comparison between MMC and OMC, the figures in Table 6.5 were compared to the new measurements in Table 6.9. The comparison was performed on the same machine as before and shows that using the new language features of the bootstrapped OMC considerably boosted performance.

Note that the latest OMC parses the MSL and HumMod sources in parallel. This is due to a prototype using external C functions that was tested on the highly parallel task of parsing compiler sources. The Boehm GC will stop all threads when collecting garbage, severely limiting performance since half of the serial execution time is spent on collecting garbage (parsing files and generating an abstract syntax tree is a memory-intensive task). Speedup when parsing Modelica...
source code is limited to around 2x due to Amdahl’s law [2] (see Table 6.4).

6.6 Algorithmic Code Debugger

The MMC compiler has a debugger developed for Modelica Development Tooling (MDT) [113] (similar to the algorithmic code debugger in Section 5.6), including a data inspector for the MetaModelica types. It uses code instrumentation and tracing to perform debugging, which produces huge executables and slow execution. Moreover, the debugger is required to fetch the complete data of every single variable at each breakpoint. It also parses the source code of the compiler to find out the names and fields of records in uniontypes. Needless to say, the debugger is slow. There are ways of making it faster, such as by limiting the maximum depth of data that it fetches from a variable. The debugger also lacks the capability to show the stack frames to view a call chain, since the MMC compiler does not use the regular C stack. The same user interface is used for debugging C code generated by the bootstrapped OMC, but the implementation of the debugger is different (see Section 3.3).

One aspect of the design of the bootstrapped compiler was to make sure that the debugged compiler code would have low overhead compared to the released version, to only query data that the user asks for and to make sure that the stack is preserved (except for tail-recursive calls). The changes made to the representation of data ensure that there is no need to parse the source code of the compiler to find out if a variable has the value NONE() or the empty list {}, and that the name of the record and its fields are stored as members in records (see Figure 6.1). Since the bootstrapped compiler uses the regular C stack and is compiled into C, it is possible to generate debug symbols for it and debug it using GDB. Compiling it takes no longer than compiling in the usual manner. Disabling compiler optimizations gives access to more variables at the cost of significantly more stack space being used.

The algorithmic code debugger uses the GDB machine interface to query runtime functions in order to get the types and values of
variables whenever a breakpoint is reached. When the user inspects a
certain variable, only the required data is fetched and displayed (the
MMC debugger fetched all information to a certain maximum depth;
the new debugger displays each level as it is expanded in the graphical
user interface (GUI)). When the debugger steps in or out, special care
is taken for setjmp/longjmp calls to make sure that the debugger
works as expected. Stepping over a function call will either end up
right after the call or at the last point caught by a matchcontinue
or failure statement.

6.7 Related Work

Functional programming language compilers often bootstrap them-
selves during the build process. Some also include integrated lexer and
parser generators specific to their own programming language. The
OpenModelica project has recently developed the OMCCp parser gen-
erator [81], integrated with MetaModelica using scanning and parsing
tables generated by flex and bison [80]. It uses features in OMC,
such as for-loops, that were never intended to be supported by MMC.
OMCCp could therefore not be used as the regular OpenModelica
parser generator until MMC was phased out completely at the end of
the bootstrapping process. It might now be included in the compiler
once performance improves.
In the Lisp \[136\] family of languages, bootstrapped compilers and interpreters have been available since the 1960s. These languages are dynamically strongly typed and defer checks to runtime instead of at compile-time. Type checking is thus only performed on code that is executed and type errors may still be present in unexecuted code.

Objective Caml \[31\], SML/NJ \[14\] and MLton \[84\] are examples of bootstrapped compilers in the ML family of programming languages. These languages are similar to MetaModelica in that they both use similar language constructs, statically strong typing and type inference. This is not surprising since RML/MMC is written in Standard ML and compiles using either SML/NJ or MLton. One major difference is that all variables in MetaModelica have a specific type, while in ML each expression has a most general type. MetaModelica can generate error messages that are easier to understand because type inference only has to be performed when calling a polymorphic function. However, this design choice also results in more local variable declarations since all temporary variables need to be declared. This is both positive (the expected type of the variable is documented) and negative (one ends up with many local variable declarations).

The concept of the `matchcontinue` expression, that is, matching with backtracking, is something that the ML family is lacking. Explicit exception handling or guard expressions can instead be used, which is often sufficient. The ML family of languages also has lambda functions, which is similar to Modelica 3.3 partial function application which OMC supports.

To summarize: the `matchcontinue` expression is more general than the `match` expression, which is common in functional programming.Clauses in logic programming are related to `matchcontinue` expressions since both provide backtracking on failure. However, in Prolog \[98\] there are usually many possible answers to a given logic program since it evaluates all combinations of clauses that satisfy the program. In MetaModelica only the first valid answer is returned and no subsequent case is evaluated. MetaModelica thus is more efficient and consistently, statically strongly typed, whereas logic programs sometimes can be expressed more concisely.
6.8 Summary and Discussion

It has been demonstrated that it is possible for a compiler of an EOO language such as Modelica to compile itself. All of the static and translational semantics of Modelica are thus expressed in a slightly extended version of Modelica. It has also been shown that the performance of the bootstrapped OMC was sufficient and seems to have further improved in the very latest versions of OMC. The effort to achieve these goals was higher than initially expected, and included not only developing the compiler but also a development environment including an Eclipse plugin and a debugger.

Many of the MetaModelica language extensions that allow language modeling are in line with the design goals for future versions of Modelica that allow modeling of language features in libraries. This has been used as input and proof-of-concept in design work.

The benefits for users of OpenModelica are many. It is possible to use OMC as a shared library and directly call functions asynchronously in threads. Calling functions directly is much faster than the current synchronous interface which sends text strings over the network and does not allow calls to be interrupted. For users of the graphical user interface, calling a function in a thread will enable OMEdit to include progress bars for long operations, in addition to the ability to cancel the operation.

The bootstrapped compiler is also able to use threads, utilizing multiple cores that are now found in nearly all consumer CPUs. Since the bootstrapped compiler and garbage collector handle threads, language constructs for threading should be introduced to allow compiler developers to write parallel code without resorting to foreign C code.

It is expected that new compiler features can be developed much faster since developers will now be able to use concepts like loops and if-expressions without using function pointers (a limitation of MMC).

Many requested features, mostly regarding interfacing with OpenModelica, had been put on hold since they are much simpler to implement in the bootstrapped compiler.

Garbage collection could still be improved by generating code that provides better hints to the garbage collector, such as allocation of data that does not contain pointers. This has been implemented for
strings, but not for vectors of numbers and similar structures used in
the backend.

An efficient build system was presented, but it does not allow
inlining of function calls due to stripping function bodies from the
interface code. Keeping the function body for inline functions could
provide a slight performance increase in the compiler.

Introduction of register allocation (optimized intermediate code)
in the compiler would help reduce the burden of the garbage collection
algorithms. The current version tries to limit the number of registers,
but it is not optimal. It is an optimization that greatly helps for using
less stack space even when generating code for debugging.

Furthermore, to realize important design goals, a number of lan-
guage features should be moved into libraries and an enhanced Ap-
lication Programming Interface (API) for accessing compiler func-
tionality from such libraries need to be developed.
Chapter 7

Parallel Simulation by TLM-Style Decoupling

This chapter is based on the work achieved in [125] (2010 paper). The initial paper did not cover parallelization, but gave the basic framework to perform parallelization. Here, a more complete solution to the problem is presented, which includes the parallelization of equation-based models as presented in [124] (2013 thesis). OpenModelica includes several different code generators for parallelization, one of which is presented here.

7.1 Background

An increasingly important way of creating efficient computations is to use parallel computing, that is, dividing the computational work among the multiple processors that are available in multi-core systems. Such systems may use either a Central Processing Unit (CPU) [66] or General-Purpose computing on Graphics Processing Units (GPGPU) [71, 145]. Since multi-core processors are becoming more common than single-core processors, it is becoming important to utilize such resources. Taking full advantage of the new parallel architectures requires support in compilers and development tools.

However, while parallelization of models expressed in Equation-based Object-Oriented (EOO) languages is not an easily resolved task, increased performance if successful is 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. Computational work can be divided by executing several processes or jobs at once, each utilizing 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 parallelization. The compiler itself analyzes the program or model, partitions the work, and automatically produces parallel code.

Automatic parallelization is usually the preferred way because users do not need to learn parallel programming, which is often error-prone and time-consuming. This is even more true in the world of equation-based languages because the modeler can be a systems designer or modeler with no real knowledge of parallel programming or algorithms.

It is not trivial to achieve automatic parallelization of models in equation-based languages. Not only is it necessary to decide which processor should perform a particular operation, it is also needed to determine the order in which to schedule computations necessary 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 [5].

There are methods that can make automatic parallelization easier by introducing parallelism over time, for example distributing solver work over time [120]. However, parallelism over time produces very limited speedup for typical Ordinary Differential Equation (ODE) systems of equations. Pipelining, inlining and distributing the solver as phases on different processors [82, 83] is a method that removes
the central solver and provides some additional speedup. However, the potential for splitting a solver into pieces is usually limited. In [83], splitting a Runge-Kutta solver into three phases produces an additional speedup of a factor of two.

A single centralized solver is the normal approach to simulation in most of today’s state-of-the-art open-source and commercial simulation tools. Although major 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.

In contrast, distributed modeling, 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 [8], also called Transmission Line Modeling (TLM), have been developed at Linköping University for many years.

This method has been successfully implemented in the HOPSAN [79] simulation package, which is one of very few simulation packages that regularly utilizes this technology, within applications in mechanical engineering and fluid power. Recent versions of HOPSAN include multi-core support [18]. TLM has also been demonstrated in [77] and subsequently in [25]. Although the method has its roots back 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 in different application areas and models is not well understood.

In this chapter the focus is on introducing distributed simulation based on TLM technology in Modelica and combining this with solver inlining, which further contributes to avoiding the centralized solver bottleneck. These techniques have demonstrated the capability to attain speedup when performing simulations.
CHAPTER 7. PARALLEL SIMULATION BY TLM-STYLE DECOUPLING

Summarizing the main contents of this chapter:

• Presentation of a structured method of modeling with model partitioning using transmission lines in Modelica that is compatible with existing Modelica tools (Section 7.3).

• Showing that such a method can use distributed solvers and may contain subsystems with different time steps, which may improve simulation performance dramatically (Section 7.5).

• Demonstrating 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.

• Showing that a simple partitioning algorithm can indeed be used to parallelize a Modelica model using transmission lines.

7.1.1 Related Work

Several people have performed work on parallelization of Modelica models [5, 27, 30, 82, 83, 85, 99, 134, 150, 157], but there are still many unsolved problems to address.

The work most similar to ours is [99], where Nyström uses transmission lines to perform model partitioning for parallelization 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 with communication through TCP/IP is especially difficult due to the unpredictability of the protocol. There is an increasing need to parallelize 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 modeled. In this work, regular Modelica models are used
without function calls for communication between model elements. Nyström used an external function interface to perform server-client communication. The method used is a more explicit way of parallelization, since it looks for the sub-models that the user created to achieve a co-simulation of sorts. However, Nyström did not develop a fully general solution for the TLM-based parallelization for Modelica; the approach was only demonstrated for a small electrical model.

This work is also similar to the HOPSAN simulation package since the used model has been adapted from HOPSAN.

The SKF TLM co-simulation framework is capable of combining models from many different simulation tools. Co-simulation works by creating a TLM interface for each simulation tool and connecting the tools together through this asynchronous interface \[95, 122\]. The co-simulation framework is the result of many years of work at SKF as well as research at PELAB.

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 \[12\], which imports and co-simulates Functional Mock-up Units (FMUs) \[13\]. Parallelism is introduced into the model by manually partitioning the system, where the states in each system should have similar time constants. It is similar to TLM in that the multiple subsystems can be co-simulated using different step sizes of the numerical solvers used for the different partitions. Discontinuities are isolated to each subsystem to avoid restarting the numerical integration in the other subsystems. For performance reasons, xMOD can relax data dependencies which means using the previous value of a signal in order to parallelize the computation at the cost of numerical precision. 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 \[82, 83\].

Casella \[27\] proposed an algorithm that causes the compiler to utilize a certain equation structure in order to parallelize the equations. This approach was shown in \[150\] to work best for the fluid domain as expected, although some performance was also gained in
Quantized State System (QSS) methods [30] belong to a different family of solvers than the solvers used in this work. QSS algorithms solve equations asynchronously with communication only at event times and only with neighboring integrators. This should make it suitable to parallelize equations.

Parallelization of Modelica-based simulations using GPGPU programming has been explored by Stavåker [134] and Östlund [157].

7.1.2 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 approach that is most common by far is to use a centralized solver algorithm that solves all equations together in a system of DAEs or ODEs. 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 parallelize 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. A special case where this is especially suitable is the transmission line element method. This allows each component to have its own step size in its solver provided that the maximum step size is guaranteed to be lower than the TLM delay time. Such a simulator has numerically highly robust properties and a high potential for taking advantage of multi-core platforms [75]. Despite these advantages, distributed solvers have never been widely adopted and centralized solvers have remained the de facto strategy on the simulation software market for Differential-Algebraic Equation (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 lan-
guages such as Modelica use implicit, centralized solvers since these solvers are easy to use. Typically the simulation is robust without a single parameter being tweaked.

Modeling for multi-core-based simulation also requires applications of significant size in order for the 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 distributed solvers not yet having been widely adopted is limited tool support and the fact that the modeler has been required to manually perform partitioning and decoupling of model parts.

The fundamental idea behind the TLM method is to model a system in such a way that components can be somewhat decoupled 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 [75]. The transmission line element method originates from the method of characteristics used in HYTRAN [1] and from TLM [70], both developed back in the sixties [8]. Today it is used in the HOPSAN simulation package for fluid power and mechanical systems (Section 7.1.3), and in the SKF TLM-based co-simulation framework [95, 122]. TLM has also been used to model other media-related Modelica models [147].

Mathematically, a transmission line can be described in the frequency domain by the four-pole equation [148]. Assuming that friction can be neglected and transforming these equations to the time domain, they can be described according to equations:

\[
p_1(t) = p_2(t - T) + Z_c q_1(t) + Z_c q_2(t - T), \quad (7.1)
\]

\[
p_2(t) = p_1(t - T) + Z_c q_2(t) + Z_c q_1(t - T). \quad (7.2)
\]
Figure 7.1: Transmission line components calculate wave propagation through a line using a physically correct separation in time.

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 7.1. In order to solve these equations explicitly, two auxiliary variables are introduced:

$$c_1(t) = p_2(t - T) + Z_c q_2(t - T), \quad (7.3)$$

$$c_2(t) = p_1(t - T) + Z_c q_1(t - T). \quad (7.4)$$

These variables are called wave variables or wave characteristics and they represent the delayed communication between the end nodes. Putting equations 7.1 through 7.4 together will yield the final relationships between flow and pressure:

$$p_1(t) = c_1 + Z_c q_1(t), \quad (7.5)$$

$$p_2(t) = c_2 + Z_c q_2(t). \quad (7.6)$$

These equations can now be solved using boundary conditions. These are provided by adjacent (resistive) components. In the same way, the resistive components receive 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.
Note that this delay time is a parameter in the component and can 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 so as to avoid extrapolation of delayed values. This means that the solver needs to be constrained to always take steps lower than or equal to the minimum TLM delay time.

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 1,000 m/s, which means that a time delay of 1 ms will represent a length of 1 m. [77]

7.1.3 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 1970s [41]. The simulation engine is based on the transmission line element method described in Section 7.1.2, with transmission lines (called C-type components) and restrictive components (called Q-type) [79]. 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 centralized libraries.

In the new version of HOPSAN [76], all equation solvers are completely distributed as a result of an object-oriented programming approach [9]. Numerical algorithms in HOPSAN are explicitly modeled as causal discrete-time models in the underlying C++ source code. 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 [18].
7.1.4 Hopsan-Modelica Differences

Modelica supports event handling as well as delayed expressions in equations. These properties will be used later in the 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 perform 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 on the choice of solver. This is where sampling and delaying expressions comes into play. The Modelica 3.3 specification [94] states the following about the delay operator:

The `delay()` operator allows a numerical sound implementation by [...] as well as a more simple realization by interpolating linearly in a buffer containing past values of expression expr [...]

In principle, a `delay` operator could break algebraic loops. For simplicity, this is not supported because the minimum delay time has to be given as additional argument to be fixed at compile time [sic]. Furthermore, the maximum step size of the integrator is limited by this minimum delay time in order to avoid extrapolation in the delay buffer.

Using the Modelica `delay` operator will not break algebraic loops since the delayed value may be interpolated by using the previous value and the value of the expression in the current time step. The fact that the current value of the expression is needed to calculate the delayed value means that it will not be possible to parallelize components connected using TLM connectors.

The delay line needed for TLM is the version of `delay` that the Modelica specification does not support. Provided that a fixed-step solver with the correct step size is used, the restriction is fulfilled. OpenModelica was enhanced to allow breaking algebraic loops using the `delay` operator if the user requests this mode with a compiler flag.
7.2 Example Model: 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 7.2. 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 7.3. 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.

Figure 7.2: The example system consists of a volume and a pressure relief valve. Boundary conditions are represented by a constant flow source and a constant pressure source.

Figure 7.3: A pressure relief valve is designed to protect a hydraulic system by opening at a specified maximum pressure.
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 suitable example model when comparing simulation tools. The reason is that it is based on dynamic equations and also includes several nonlinearities, making it an appropriate 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 analyzed by frequency analysis techniques, such as using bode plots or Fourier transforms. It also includes several 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 nonlinear phenomena such as cavitation and hysteresis, although these are not included here.

The volume is modeled 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 neighboring 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.

$$mZc = \frac{m\text{Bulkmodulus}}{m\text{Volume}} \times m\text{Timestep};$$
$$c10 = p2 + mZc \times q2;$$
$$c20 = p1 + mZc \times q1;$$
$$c1 = m\text{Alpha}*c1 + (1.0-m\text{Alpha})*c10;$$
$$c2 = m\text{Alpha}*c2 + (1.0-m\text{Alpha})*c20;$$

The pressure relief valve is a restrictive component, known as Q-type, which means that it receives characteristic variables and impedance from its neighboring components and returns flow and pressure. Advanced models of pressure relief valves are normally \textit{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 complicated however, and not intuitive. This is why 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 modeling.

\[
\begin{align*}
    mF_s &= mPilotArea \times mPref; \\
    p1 &= c1 + q1 \times Z_{c1}; \\
    F_{tot} &= p1 \times mPilotArea - mF_s; \\
    x_0 &= mFilter.value(F_{tot}); \\
    mTurb.setFlowCoefficient(mCq \times mW \times x_0); \\
    q2 &= mTurb.getFlow(c1, c2, Z_{c1}, Z_{c2}); \\
    q1 &= -q2; \\
    p1 &= c1 + Z_{c1} \times q1; \\
    p2 &= c2 + Z_{c2} \times q2;
\end{align*}
\]

### 7.3 TLM in an Equation-based Language

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

TLM has 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 that 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 sample() built-in function is shown below. Variables defined within when-equations in Modelica as below will have discrete-time variability.

```modelica
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;
```

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 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.

```modelica
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 since it does not introduce additional approximation of the delay line.

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

Figure 7.4 shows the results of simulating the example system, that is, the pressure relief valve from Section 7.2. Figures 7.5 and 7.6 are magnified versions that show the difference between the 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 G, with the problemSize parameter
Figure 7.4: Pressure increases until the reference pressure of 10 MPa is reached, at which the relief valve opens.

set to 1.

Note the similarity between the curves of the delay implementation and the ideal der operator. It looks like $\times$ and $+$ are combined into $\div$. 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 or the concept of equations at all. Instead it uses custom C++ code which uses other integrators and approximations directly in the components. The Modelica model corresponds to something that is slightly different from the HOPSAN model.

To verify that different Modelica tools produce the same results, a comparison was made between Dymola [34] and OpenModelica (Figure 7.7). It shows that both OpenModelica and Dymola produce almost exactly the same results. Comparisons of the delay and sample operators are not provided because OpenModelica performs optimizations essential for models written using TLM. Dymola cannot 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 produces unexpected
Figure 7.5: Comparison of spool position using different TLM implementations. The curves for delay, der, and sample overlap almost completely. The Hopsan model is slightly different from the Modelica model, which causes the curves to also be slightly different.

Figure 7.6: Comparison of system pressure using different TLM implementations. The curves for delay, der, and sample overlap almost completely. The Hopsan model is slightly different from the Modelica model, which causes the curves to also be slightly different.
7.3. TLM IN AN EQUATION-BASED LANGUAGE

Figure 7.7: 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 results file.

<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>

Table 7.1: Performance comparison of methods for performing transmission line modeling in Modelica.

results.

The OpenModelica DASSL solver could not be used since OpenModelica does not provide an option to limit the maximum step size for the TLM delay, resulting in extrapolation. For this reason, the RK4\(^1\) solver was used throughout since it provides better numerical stability than explicit Euler.

Table 7.1 shows average performance numbers for the models based on 20 runs. Only one segment was used for the pipe and the simulation did not write a results file, which means the simulations

\(^1\)Explicit Runge-Kutta solver of order 4, with root finding of zero-crossings.
run extremely fast. The only real surprise here is that sampling in
OpenModelica improved in performance by a few orders of magni-
tude since the original paper was published [125]. This is due to time
events being implemented during the past few years in OpenModelica,
which means exact times for events can be found without searching
for them in the zero-crossing function. While sampling is now rather
fast, at least for small systems, the OpenModelica prototype cannot
parallelize a sampled system since the parallelization scheme does not
yet consider discrete events.

So far only single-core simulations have been run and the problem
size is so small that it cannot really be parallelized.

The current OpenModelica simulation runtime system implementa-
tion does not have special efficient implementations for delayed ex-
pressions but is still fast compared to the alternatives.

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

7.4 Partitioning

The main reason for using TLM is that a coarse-grained paralleliza-
tion of the system is implicitly gained. A general approach to parti-
tioning a system of equations that utilizes the time delay introduced
by TLM is presented below.

All of the partitions will run in the normal simulation loop in
OpenModelica, which means that all of the partitions need to be
synchronized at each global time step. Within the current time step,
each partition of the equation system will be independent from any
other and the evaluation of the partitions can be conducted in parallel.

To illustrate what the used algorithm does, consider the standard
approach to causalize an equation system (Section 2.5.2). The equa-
tion system is transformed into Block Lower Triangular (BLT) form,
where each BLT block corresponds to either a single equation from
7.4. PARTITIONING

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 7.8b, the entry using $X$ has been removed from Figure 7.8a by using a TLM delay. The new matrix can be divided into two separate ones and solved sequentially or in parallel since the two blocks are now totally independent from each other (Figure 7.8c).

7.4.1 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 completely connected in the graph. The benefit of only looking at the adjacency matrix is that the equation system can be partitioned before optimizations are performed, some of which are costly to perform on large systems since they do not have linear time complexity.

Because not all nodes are connected to each other, the graph is not a tree, but a forest. The goal is to find all the trees in the forest since these are possible to run in parallel. There are many ways to rapidly perform this operation. The method to choose depends mostly on the used data representation. Cormen [33] contains an algorithm of
interest using disjoint sets. The chosen approach instead uses a depth-first search, marking all reachable nodes and then choosing the next unmarked vertex and repeating the algorithm until all the trees have been found. The algorithm has a time complexity of $O(|V| + |E|)$, as any connected components algorithm should.

7.5 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 small time step only for certain components. The advantage of such a distributed system becomes apparent in [77]. It is also possible to use asynchronous communication with distributed solvers, which enables numerical solvers to choose the step size instead of being forced to stop at synchronization points – this is the approach used by the SKF TLM co-simulation framework [95, 122].

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 [94] introduced synchronous language features and annotations that can be used to select fixed-step solvers and step sizes for the newly introduced clocked partitions. This is not yet supported in OpenModelica.

7.6 Parallel Performance

Generating parallel code for an equation system with TLM connections is simple. It has already been shown that the partitioning algorithm works in linear time and produces the sets of equations to be solved in parallel. The OpenModelica Compiler (OMC) provides many different parallel code generators. The version used here is not maintained as the newer code generators are more advanced. However, this version is the only one that parallelizes the algorithmic parts \texttt{(functionAlgebraics)} of the equation system (the others only parallelize calculation of state variables, which do not exist when using the delay operator instead of der in Modelica). The code generator
is very simple and utilizes the partitioned systems of equations. The generated code has the general structure shown below:

Listing 7.1: OpenMP code.

```c
static void functionAlg_system1 (DATA *data)
{
  eqFunction_268(data);
  ...
}
static void (*functionAlg_systems[2])(DATA *)
  = {
    functionAlg_system0,
    functionAlg_system1
  };
void functionAlgebraics (DATA *data)
{
  #pragma omp parallel for private(id) schedule
    (static)
    for(id=0; id<2; id++)
      functionAlg_systems[id](data);
}
```

The serial code is provided as a reference point:

Listing 7.2: Serial code.

```c
static void functionAlg_system0 (DATA *data)
{
  eqFunction_268(data);
  ...
}
void functionAlgebraics (DATA *data)
{
  functionAlg_system0(data);
}
```

Code can be generated 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
CHAPTER 7. PARALLEL SIMULATION BY TLM-STYLE DECOUPLING

<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>

Table 7.2: Performance of parallelization on a four-core machine.

algorithm. The execution time estimates are too simplistic to make static scheduling a real choice using the current compiler prototype.

To test 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 parallelization.

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 7.2 and the speedup values in Figure 7.9. This is equivalent to one volume with a step size of $10^{-4}$, which was used earlier. Figures 7.10 and 7.11 show a comparison of the volume modeled 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 is 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 results file using a serial implementation and comparing the files using the `compareSimulationResults()` API function in OpenModelica. These tests produced the exact same results.

A four-core i7-3820 with hyper-threading was used for performance measurements using GCC 4.4 on Linux. Some speedup was obtained with static scheduling when more than four threads were used. The reason is that the rough estimates of execution time significantly differ from the real-world performance. Dynamic scheduling
7.6. PARALLEL PERFORMANCE

Figure 7.9: Speedup graph for parallelization using static scheduling.

Figure 7.10: Comparison of spool position using a volume split into more segments.
works as well as static scheduling, but neither was much faster than the serial implementation. Part of the reason is that the model was not split into many different parts. Scaling this particular model into more parts involves cutting the volumes into more parts, which shortens the delay line and forces the simulation 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 within a reasonable time frame. Twenty segments was a small enough size to show that the model scales when more cores were used.

While good performance is the goal, work so far has been to show that the generated simulation code behaves as expected and produces correct results. Note that while in the best case, minimum time, more threads improve performance. 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.

Figure 7.11: Comparison of system pressure using a volume split into more segments. There is a numerical error that is amplified by splitting the volume into several segments.
The chosen example is simple for good reason: at the time, the OpenModelica runtime code was not thread-safe in the general case. The OMC backend has optimized the model to the point that no memory allocation, like creating a temporary array, needs to be performed during a time step.

### 7.6.1 Thread Safety

Since the code worked as expected, the OpenModelica runtime system was changed to be thread-safe in order to support parallelization. A global data structure as well as a thread-local data structure is passed along in every function. Thread-local variables are used instead of static or global buffers. Memory allocation is performed by a thread-safe Garbage Collection (GC) algorithm.

In the bootstrapped OpenModelica compiler itself (Chapter 6), the thread safety of the generated code is used to speed up for example parsing of source code. Parallel code generators for OMC reuse the same code generation and runtime for some of its targets, taking advantage of the runtime thread safety.

### 7.7 Summary and Discussion

A model of a pressure relief valve using the TLM technique has been translated from the Hopsan simulation environment to Modelica. The model has a configurable implementation of the transmission lines using different Modelica language constructs. Each implementation produces the correct results.

The test model has been scaled up to demonstrate that a simple parallelization algorithm works fine and provides a slight speedup for the model. The reason that speedup is not better is that the focus of the overall thesis has shifted from parallelization to debugging over the course of time. There is ongoing work by other researchers that is partly based on the methods presented here. These methods provide

---

2The OpenMP versions use a different GC mechanism since the default one requires registering the stack start address for each thread, which is only exposed when using the pthreads API.
better speedup due to using a better estimation for execution time, more advanced dynamic scheduling algorithms, or both.

The main benefits of this work that still remain in wide use is that the simulation runtime environment is thread-safe. It is now easy to create new parallel code generators assuming that scheduling of the equation system has been performed.

The partitioning algorithm also remains in use, where it provides minor speedup for models using automatic parallelization or simply improving translation speed. The algorithm can also be used if work on modeling TLM models in Modelica continues, as opposed to using the proposed stand-alone TLM simulator below.

There are two different ways to proceed with this work: continue with a pure Modelica approach for transmission line modeling, or an approach based on Functional Mock-up Interface (FMI) for co-simulation. They are very similar and both are currently being evaluated.

### 7.7.1 Modelica-based Approach

To make performance even faster OMC could improve the code generation of the \texttt{delay} operator, which currently evaluates the delay expression even when only historical data is necessary. Modelica could also be extended\footnote{Such an extension might become part of Modelica in the future due to the desire to run FMUs inside Modelica models: \url{https://trac.modelica.org/Modelica/ticket/1626}} to allow generation of an event whenever the numerical solver accepts a step. This would allow implementing a custom TLM \texttt{delay} operator in Modelica code.

Since the current prototype partitions the Modelica model it would be able to simulate subsystems using distributed solvers with different step sizes \cite{69, 77}, also referred to as multi-rate integration \cite{29, 78}. 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 synchronization overhead.

Parallelizing 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 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 parallelized – dependent variables are calculated after the states and would then still be calculated in serial unless methods were combined.

Using clocked synchronous variables from Modelica 3.3 [94] instead of delays to model TLM would be of interest. These language elements were introduced with real-time systems in mind and support implementation of fixed step solvers within Modelica itself. Support for the synchronous language features introduced in Modelica 3.3 would then first need to be implemented in OpenModelica. Time delays using a Backward Differentiation Formula (BDF) solver however, should provide better performance for systems that do not have real-time constraints.

There is a need to write more Modelica models using TLM. The current example is a single model from the fluid domain. While this model was chosen because it contains all the necessary elements for modeling using TLM, more advanced models are needed to test parallelization.

### 7.7.2 FMI-based Approach

The Modelica-based approach to TLM has disadvantages in that Modelica was designed around not providing access to the numerical solver from the modeling language. As such, it is impossible to generate an event when the solver accepts a step. It is also impossible to access the previous value for a variable and so on. Modelica language extensions would make it possible to implement an efficient TLM package in Modelica, but the extensions adversely affects maintainability and tool compatibility.

When this work was started [125], FMI was not yet supported by many tools (standardized during 2010, with tool support emerging mostly in 2011). It is now available in many tools and offers support for controlling the numerical solvers (using FMI for model exchange), or step size control (FMI for co-simulation). There is one limitation
removing in FMI for co-simulation with respect to TLM simulation perform – it is only possible to ask the solver to take a step with a given step size, rather than asking the solver to take a single step and return the length of this step.

Basic FMUs are simulated by giving all inputs, simulating one step and receiving all the outputs. This is sufficient for TLM simulation, simply connecting FMUs together. Each FMU will advance one time step (corresponding to the delay modeled in Modelica) and data is exchanged. The system is partitioned with one partition for each FMU, making parallelization trivial (the input/output dependency is broken by the simulator introducing the time delay).

However, an FMI simulation tool is needed in order to connect different FMUs together. This is the tool that needs to perform the parallelization, which in relation to this work moves the effort of parallelization from OMC to the FMI simulator.

### 7.7.3 Stand-alone TLM Simulator

Due to the limitations of Modelica, work has begun on developing an open-source stand-alone tool for TLM co-simulation between different tools (after a code donation from the existing SKF TLM co-simulation framework [95, 122]). One of the new targets of this tool could be FMUs [13]. Another target could be Modelica models with extensions to communicate efficiently with the numerical solver. The tool will be integrated in the OpenModelica Connection Editor (OMEdit).

Recent development of HOPSAN [79] has taken a similar approach, where FMUs can be imported into the HOPSAN modeling environment, giving the user the ability to combine for example HOPSAN and OpenModelica.
Chapter 8

Conclusion

The advantages of high-level of abstraction of Equation-based Object-Oriented (EOO) modeling formalisms such as Modelica when used in today’s EOO simulation tools currently have drawbacks regarding debugging and finding performance problems in models. If there is an error when executing a simulation it is very difficult for the user to understand, based on the generated code, which part or parts of the model give rise to the problem. The main reason for this difficulty is the large gap in abstraction between the model and the executable simulation code.

To remedy these and related problems new and improved methods and tools that scale to large and complex models have been developed, including the following:

- The first truly efficient runtime and compile-time debugger for equation-based models. It includes a method and efficient implementation for tracing the symbolic transformation operations performed in the backend of an EOO modeling language (Modelica) compiler to be used in debugging tool support and in generating explanations for the user. The debugger graphical user interface (GUI) can map error positions in the executing low-level generated code back to the equations in the high-level model that give rise to the problem. A schema for standardizing the storage of the debugging metadata has been developed as part of this.
• The first runtime simulation performance profiler for an EOO modeling tool that is integrated with a high-level debugger. The tool can detect which (parts of) equation systems are slow to simulate. Combined with debug information, the tool can also point out why they are slow to simulate. Certain configurations of the profiler have very low overhead and are thus suitable for real-time applications.

Moreover, compilation and simulation methods and tools have been developed as part of this work:

• A general and efficient compilation scheme including a code generator that generates efficient C code that is easy to debug and profile using standard, usually C-based, development tools. This compilation scheme has been created to efficiently support both standard Modelica and MetaModelica, and has been evaluated on a very large application: the bootstrapping of the OpenModelica compiler itself, a program of approximately 330,000 lines of code. This bootstrapped OpenModelica Compiler (OMC) is the first industrial scale EOO language compiler that can compile itself.

• An algorithmic code debugger has been developed for debugging the C code generated when compiling both Modelica and MetaModelica code, with support for displaying values for data types in the languages.

• A method for parallelizing simulation models based on decoupling models parts using Transmission Line Modeling (TLM) elements, with automatic dependency analysis and partitioning of systems of equations. Such an approach has been developed for the Modelica language implemented in OpenModelica and has been evaluated on a scalable model that uses delay lines, producing some speedup. Additionally, the OpenModelica runtime has been made thread-safe, enabling experimentation with many parallelization techniques.
8.1 Future Work

The work so far has only broached the subject. There is a lot more to be done especially for debugging equation-based models. One important area of debugger improvement is specific support for debugging nonlinear systems of equations. Other areas are debugging and performance analysis of parallelized models simulating on multi-core architectures, as well as models using the recent Modelica 3.3 clocked synchronous language constructs.

By visualizing the Block Lower Triangular (BLT) matrix of the debugged equation system, a user could be presented with an overview of how the equation systems are solved, which equations block parallel execution of the model, and if there are parts of the simulation model that are not needed in order to calculate the variables that are of interest at the moment.

The two debuggers in the thesis have been developed as different tools that should now be integrated with each other, allowing the user to seamlessly switch views between algorithmic code debugging and the transformations browser.

Performance profiling on parallel simulations is a topic that is of especial interest for comparing different parallelization code generators.

Standardization of extensions to the Modelica language specification is of major interest to OpenModelica developers and users. This includes adding language features that control numerical algorithms, making it easier to import Functional Mock-up Units (FMUs) and improve performance of TLM models. Other features are data types introduced as part of bootstrapping OMC, making OpenModelica source code more similar to standard Modelica. Introducing parallel execution models in the Modelica language would allow OpenModelica source code to use the same execution model, making the source code easier to read and maintain than if MetaModelica used a different parallel execution model.
Appendix A

Flat Modelica

Flat Modelica is type-checked Modelica code where the object-oriented parts have been flattened and connections have been expanded. The following is the flattened RC circuit from Chapter 2:

Listing A.1: Flattened RC circuit.

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") "Current flowing into 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) = 1e-06 "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";

173
Real r.p.i(quantity = "ElectricCurrent", unit = "A") "Current flowing into the pin";
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.offset "Number of periods (< 0 means infinite number of periods)";
parameter 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_start(quantity = "Time", unit = "s") "Start time of current period";
protected Integer pv.signalSource.count "Period count";
protected Real pv.signalSource.T_width (quantity = "Time", unit = "s") = 0.01 * 
    pv.signalSource.period * 
    pv.signalSource.width;
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) 
        >= 1e-15, "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 + pv.n.i + g.p.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

Translated Code

The flat Modelica from Appendix A is translated by a compiler into an intermediate representation that can easily be expanded into sequential C code that numerical solvers are able to utilize. Listing B.1 shows how the flattened code looks after passing through the OpenModelica backend. The information is taken from the debugging metadata (see Appendix D for the schema). Note that the file has been stripped of some information (such as the initial equation system, the parameter equation system, file references, and so on).

Listing B.1: Translated RC circuit.

```json
{
   "info": {
      "name": "RC",
      "description": ""
   },
   "format": "Transformational debugger info",
   "variables": {
      "g.p.i": {
         "source": {
            "operations": [
               { "data": [ "g.p.i = 0.0" ] },
               { "display": "solved", "op": "info" }
            ]
         },
         "data": [ ]
      }
   }
}
```
"(-r.i) + r.i + g.p.i = 0.0",
"g.p.i = 0.0"
],
"display": "simplify",
"op": "before-after"
},
{
"data": [
"c.n.i + pv.n.i + g.p.i",
"(-r.i) + r.i + g.p.i"
],
"display": "substitution",
"op": "chain"
},
{
"data": [
"c.n.i + pv.n.i + g.p.i = 0.0",
"c.n.i + pv.n.i + g.p.i = 0.0"
],
"display": "simplify",
"op": "before-after"
}
],
"r.R_actual": {
"source": {
"operations": [
{
"data": [
"r.R_actual = r.R * (1.0 + r.alpha * (r.T - r.T_ref))"
]
"display": "solved",
"op": "info"
}]
"data": [
"r.R * (1.0 + r.alpha * (r.T_heatPort - r.T_ref))"
],
"display": "substitution",
"op": "chain"
}];
"data": [  "R_actual = R * (1 + alpha * (T_heatPort - T_ref));" ,  "r.R_actual = r.R * (1.0 + r.alpha * (r.T_heatPort - r.T_ref));"  ],  "display": "flattening",  "op": "before-after"
]}

"pv.signalSource.T_width": {  "source": {  "operations": [  {  "data": [  "pv.signalSource.T_width = 0.01 * pv.signalSource.period * pv.signalSource.width"  ],  "display": "solved",  "op": "info"
  ]
]}

"$whenCondition1": {  "source": {  "operations": [  {  "data": [  "count = pre(count) + 1;",  "pv.signalSource.count = 1 + pre(pv.signalSource.count);"
  ],  "display": "flattening",  "op": "before-after"
  ]
]}

"version": 1,
"equations": [
APPENDIX B. TRANSLATED CODE

```json
{
  "section": "regular",
  "eqIndex": 21,
  "source": {
    "operations": [
      {
        "data": [
          "$whenCondition1 = integer((time - pv.signalSource.startTime) / pv.signalSource.period, 0) > pre(pv.signalSource.count)"
        ],
        "display": "solved",
        "op": "info"
      },
      {
        "data": [
          "count = pre(count) + 1;",
          "pv.signalSource.count = 1 + pre(pv.signalSource.count);"
        ],
        "display": "flattening",
        "op": "before-after"
      }
    ],
    "tag": "assign",
    "uses": [
      "pv.signalSource.count",
      "pv.signalSource.period",
      "pv.signalSource.startTime",
      "time"
    ],
    "equation": [
      "integer(DIVISION(time - pv.signalSource.startTime, pv.signalSource.period), 0) > pre(pv.signalSource.count)"
    ],
    "defines": [
      "$whenCondition1"
    ]
  }
}
```
"uses": [ "$whenCondition1", "time" ],
"equation": [ "time" ],
"defines": [ "pv.signalSource.T_start" ]
},
{
"section": "regular",
"eqIndex": 23,
"tag": "when",
"uses": [ "$whenCondition1", "pv.signalSource.count" ],
"equation": [ "1 + pre(pv.signalSource.count)" ],
"defines": [ "pv.signalSource.count" ]
},
{
"section": "regular",
"eqIndex": 24,
"source": {
"operations": [
{
"data": "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)"
],
"display": "solved",
"op": "info"
}
"data": [
  "pv.signalSource.y",
  "pv.v"
],
"display": "substitution",
"op": "chain"
},
"data": [
  "y = offset + (if time < startTime or nperiod == 0
  or nperiod > 0 and count >= nperiod then 0
  else if time < T_start + T_width then amplitude
  else 0));",
  "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);"
],
"display": "flattening",
"op": "before-after"
],
"tag": "assign",
"uses": [
  "pv.signalSource.amplitude",
  "pv.signalSource.T_width",
  "pv.signalSource.T_start",
  "pv.signalSource.count",
  "pv.signalSource.nperiod",
  "pv.signalSource.startTime",
  "time",
  "pv.signalSource.offset"
],
"equation": [
  "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)"
]
"defines": [
  "pv.v"
],
"section": "regular",
"eqIndex": 25,
"source": {
  "operations": [
    {
      "data": [
        "r.v = pv.v - c.v"
      ],
      "display": "solved",
      "op": "info"
    },
    {
      "data": [
        "r.p.v - r.n.v",
        "pv.v - c.v",
        "pv.p.v - c.p.v"
      ],
      "display": "substitution",
      "op": "chain"
    },
    {
      "data": [
        "v = p.v - n.v;",
        "r.v = r.p.v - r.n.v;"
      ],
      "display": "flattening",
      "op": "before-after"
    }
  ],
  "tag": "assign",
  "uses": [
    "c.v",
    "pv.v"
  ],
  "equation": [
    "pv.v - c.v"
  ],
  "defines": [}
"r.v"
],
{
"section": "regular",
"eqIndex": 26,
"source": {
"operations": [
{
"data": [
"r.v = r.R_actual * r.i",
"r.i = r.v / r.R_actual"
],
"display": "solved",
"op": "before-after"
}
},
"data": [
"v = R_actual * i;",
"r.v = r.R_actual * r.i;"
],
"display": "flattening",
"op": "before-after"
}
],
"tag": "assign",
"uses": [
"r.R_actual",
"r.v"
],
"equation": [
"DIVISION(r.v, r.R_actual)"
],
"defines": [
"r.i"
]
},
{
"section": "regular",
"eqIndex": 27,
"source": {
"operations": [
{
"data": [
"r.v = r.R_actual * r.i",
"r.i = r.v / r.R_actual"
],
"display": "solved",
"op": "before-after"
}
},
"data": [r.v]
"r.LossPower = r.v * r.i"
],
"display": "solved",
"op": "info"
},
{
"data": [
"LossPower = v * i;",
"r.LossPower = r.v * r.i;"
],
"display": "flattening",
"op": "before-after"
}
],
"tag": "assign",
"uses": [
"r.i",
"r.v"
],
"equation": [
"r.v * r.i"
],
"defines": [
"r.LossPower"
]
},
{
"section": "regular",
"eqIndex": 28,
"source": {
"operations": [
{
"data": [
"r.i = c.C * der (c.v)",
"der (c.v) = r.i / c.C"
]
},
"display": "solved",
"op": "before-after"
}
],
"data": [
"c.i",
"r.i"
]
"display": "substitution",
"op": "chain"
},
{
"data": [
"C * der(v)",
"C * der(v)"
],
"display": "differentiate d/dtime",
"op": "before-after"
},
{
"data": [
"i = C * der(v);",
"c.i = c.C * der(c.v);"
],
"display": "flattening",
"op": "before-after"
}]
},
"tag": "assign",
"uses": [
"C",
"r.i"
],
"equation": [
"DIVISION(r.i, C)"
],
"defines": [
"der(v)"
],
"source": {
"operations": [
{
"data": [
"1.0 + r.alpha * (r.T_heatPort - r.T_ref) >= 1e-15",
"1.0 + r.alpha * (r.T - r.T_ref) >= 1e-15"
],
"display": "substitution",
"op": "chain"
}]}
{  
  "data": [  
    "assert(1 + alpha * (T_heatPort - T_ref) >=
   Modelica.Consts.\eps, \"Temperature outside
   scope of model!\") ;"
  ],  
  "display": "scode",
  "op": "info"
}  
]  
},  
"section": "regular",
"tag": "algorithm",
"eqIndex": 29,
"equation": [  
    "assert(1.0 + r.alpha * (r.T - r.T_ref) >= 1e-15,
   \"Temperature outside scope of model!\") ;"
  ]  
]  
]  
]  
}
Appendix C

Generated Code Fragments

The following files are code fragments from the C code and JSON files generated when compiling the model in Section 3.2.4:

Listing C.1: OpenModelica trace (JSON fragment).

```json
{
   "eqIndex":1,
   ...
 },
{
   "eqIndex":2,
   "section":"regular",
   "tag":"residual",
   "uses":["x","y"],
   "equation":["x + (-y) * time"],
   "source":{
      "info":{
         "file":"debug/M.mo",
         "lineStart":5,
         "lineEnd":5,
         "colStart":3,
         "colEnd":15
      },
      "operations":[]
   }
}
```

191
"display": "simplify",
"data": [
"x = y * time * 1.0",
"x = y * time"
],

"op": "before-after",
"display": "substitution",
"data": [
"x = y * time * z",
"x = y * time * 1.0"
]

"eqIndex": 3,
"section": "regular",
"defines": ["x", "y"],
"tag": "container",
"display": "nonlinear",
"equation": [1, 2],
"source": {
...
Listing C.2: OpenModelica ODE code fragment.

```modelica
/* 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 C.3: OpenModelica residualFunc3.

```modelica
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);
```
Appendix D

Transformations Browser

JSON Schema

The following is the JavaScript Object Notation (JSON) schema used by the transformations browser. Note that the file generated by Open-Modelica contains additional information, such as the component name and types that an equation originates from. This additional information can be ignored by the debugger GUI or used to present this additional information in some way.


```json
{
    "title": "Schema for transformational debugging",
    "type": "object",
    "properties": {
        "format": {
            "type": "string",
            "pattern": "^ Transformational debugger info$",
        },
        "version": {
            "type": "integer"
        },
        "info": {
            "type": "object",
            "patternProperties": {
                "": {
                    "type": "string"
                }
            }
        }
    }
}
```
"variables": {
  "type": "object",
  "patternProperties": {
    "": {
      "type": "object",
      "properties": {
        "comment": {
          "type": "string"
        },
        "kind": {
          "type": "string"
        },
        "type": {
          "type": "string"
        },
        "unit": {
          "type": "string"
        },
        "displayUnit": {
          "type": "string"
        },
        "source": {
          "$ref": "#/definitions/source"
        }
      }
    }
  }
},
"equations": {
  "type": "array",
  "items": {
    "type": "object",
    "properties": {
      "eqIndex": {
        "type": "integer",
        "description": "A unique index for each equation"
      },
      "section": {
        "type": "string",
        "description": "Group equations by sections like initial or regular"
      },
      "tag": {
        "type": "string",
        "description": "The type of equation, e.g. assign. Influences how tools describe the equation, especially for strong components."
      },
      "display": {
        "type": "string",
        "description": "What to display the tag as."
      },
      "defines": {

"type": "array",
"items": [{"type": "string"},
        "uniqueItems": true,
        "description": "A list of variables that will be assigned when executing this equation. For strong components or array equations, the number can be greater than one."
},
"uses": {
    "type": "array",
    "items": [{"type": "string"},
              "uniqueItems": true,
              "description": "A list of variables that this equation depends on. Typically, equations that define these variables need to be executed before this equation. The defines and uses of equations in a section may be used to create a BLT matrix for those equations."
},
"equation": {
    "type": "array",
    "items": [{"type": ["string", "integer", "array", "object"]},
              "description": "A list of most often strings used to display the equation. Some tags, like assign, use the defines property for the lhs of the equation. Other tags use multiple entries here."
},
"source": {"$ref": "/definitions/source"}
},
"required": ["eqIndex","tag"]
}
"info": {
  "type": "object",
  "properties": {
    "file": {"type": "string"},
    "lineStart": {"type": "integer"},
    "lineEnd": {"type": "integer"},
    "colStart": {"type": "integer"},
    "colEnd": {"type": "integer"}
  },
  "required": ["file","lineStart"]
},
"operations": {
  "type": "array",
  "items": {
    "type": "object",
    "properties": {
      "op": {
        "type": "string"},
      "display": {
        "type": "string",
        "description": "Used by the GUI to display similar operations in the same way, plus this label to describe what kind of operation was performed"
      },
      "data": {
        "description": "A list of strings to describe the operation performed; for before-after this is the before and after expressions."
      }
    },
    "required": ["op","data"]
  }
},
"required": ["info"]}
Appendix E

Debugging Package

Debugging.mo is a package created by Francesco Casella to demonstrate common problems that occur when modeling fluid systems. The package is part of the OpenModelica source repository and is used to test certain aspects of the debugger. Note that the package was written without looking at OpenModelica; as such, some of the documented expected behavior is different from how the problems appear in OpenModelica.

Listing E.1: Debugging.mo.

```modelica
within ;
package Debugging "Test cases for debugging of declarative models"

package Chattering "Models with chattering behaviour"

model ChatteringEvents1 "Exhibits chattering after t = 0.5, with generated events"
  Real x(start=1, fixed=true);
  Real y;
  Real z;
  equation
    z = if x > 0 then -1 else 1;
    y = 2*z;
    der(x) = y;
    annotation (Documentation(info="<html>
      <p>After t = 0.5, chattering takes place, due to the discontinuity in the right hand side of the first equation.</p>
    
```
Chattering can be detected because lots of tightly spaced events are generated. The feedback to the user should allow to identify the equation from which the zero crossing function that generates the events originates.

After $t = 0.5$, chattering takes place, due to the discontinuity in the right hand side of the second equation.

Chattering can be detected because lots of tightly spaced events are generated. The feedback to the user should allow to identify the equation from which the zero crossing function that generates the events originates.

Chattering can be detected because lots of tightly spaced events are generated. The feedback to the user should allow to identify the equation from which the zero crossing function that generates the events originates.
After $t = 0.5$, chattering takes place, due to the discontinuity in the right hand side of the first equation. The discontinuity does not generate state events, thanks to the noEvent operator.

If a variable-step-size integration algorithm with error control is used, the time step will be reduced to very small values once the discontinuity is hit, and this can be detected by monitoring the value of time at each time step.

Variable step size solvers usually allow to identify which state variable(s) give the biggest contribution to the error estimate, thus causing the step size reduction. The corresponding derivative shows very high frequency oscillations between two values. The end user can then use the BLT navigation functionality of the debugger to investigate which variable/equation is introducing the discontinuity.

```plaintext
model ChatteringNoEvents1
  "Exhibits chattering after t = 0.422, without generated events"
  Real x(start=1, fixed=true);
  Real w(start=0, fixed=true);
  Real y;
  Real z;
  equation
  der(w) = -w + 1;
  z = noEvent(if x > 0 then -1 else 1);
  y = 2*(z - w);
  der(x) = y;
  annotation (Documentation(info="<html>
  <p>After t = 0.422, chattering takes place, due to the discontinuity in the right hand side of the second equation. The discontinuity does not generate state events, thanks to the noEvent operator.</p>
  </html>

If a variable-step-size integration algorithm with error control is used, the time step will be reduced to very small values once the
discontinuity is hit, and this can be detected by monitoring the value of time at each time step. </p>

<p>Variable step size solvers usually allow to identify which state variable(s) give the biggest contribution to the error estimate (x, in this case), thus causing the step size reduction. The corresponding derivative shows very high frequency oscillations between two values. The end user can then use the BLT navigation functionality of the debugger to investigate which variable/equation is introducing the discontinuity. </p>

<html>
<p>, experiment(StopTime=1));
end ChatteringNoEvents2;</p>

model ChatteringFunction1
"Exhibits chattering after t = 0.4, without generated events"
  Real x(start=1, fixed=true);
  Real y;
  Real z;
  equation
   z = Functions.f_sign(x);
   y = 2*z;
   der(x) = y;
  annotation (Documentation(info=""

<p>After t = 0.5, chattering takes place, due to the discontinuity in the right hand side of the first equation. The discontinuity is caused by a discontinuous function, which does not generate events. </p>

<p>If a variable-step-size integration algorithm with error control is used, the time step will be reduced to very small values once the discontinuity is hit, and this can be detected by monitoring the value of time at each time step. </p>

<p>Variable step size solvers usually allow to identify which state variable(s) give the biggest contribution to the error estimate, thus causing the step size reduction. The corresponding derivative shows very high frequency oscillations between two values. The end user can then use the BLT navigation
functionality of the debugger to investigate which variable/equation is introducing the discontinuity. </p>
</html>
), experiment(StopTime=1));
end ChatteringFunction1;

model ChatteringFunction2
  "Exhibits chattering after t = 0.422, without generated events"
  Real x(start=1, fixed=true);
  Real w(start=0, fixed=true);
  Real y;
  Real z;
equation
  der(w) = -w + 1;
  z = Functions.f_sign(x);
  y = 2*(z - w);
  der(x) = y;
annotation (Documentation(info="<html>
<p>After t = 0.422, chattering takes place, due to the discontinuity in the right hand side of the second equation. The discontinuity is caused by a discontinuous function, which does not generate events.</p>
<p>If a variable-step-size integration algorithm with error control is used, the time step will be reduced to very small values once the discontinuity is hit, and this can be detected by monitoring the value of time at each time step.</p>
<p>Variable step size solvers usually allow to identify which state variable(s) give the biggest contribution to the error estimate (x, in this case), thus causing the step size reduction. The corresponding derivative shows very high frequency oscillations between two values. The end user can then use the BLT navigation functionality of the debugger to investigate which variable/equation is introducing the discontinuity.</p>
</html>
"), experiment(StopTime=1));
end ChatteringFunction2;
end Chattering;
package SolverFailure "Models showing different numerical failure modes"
import SI = Modelica.SIunits;

model NonlinearSolverFailureInitial
"Nonlinear system of equations has no solutions"
parameter SI.Pressure patm=101325 "Atmospheric pressure";
parameter Real Kv=1e-2 "Valve coefficient";
parameter Real dp_small=1 "Small dp for valve equation";
parameter Real dp0=3e5 "Pump dp @ zero flow";
parameter Real a1=1e6 "Pump coefficient";
parameter Real a2=3e2 "Pump coefficient";
parameter Real a3=3e2 "Pump coefficient";
parameter SI.Temperature T0=20 + 273.15 "Temperature of incoming fluid";
parameter SI.Density rho=995 "Density of fluid";
parameter SI.Area A=0.01 "Storage tank cross section";
parameter SI.MassFlowRate w_extra=0 "Extra mass flow rate into reservoir";
constant SI.Acceleration g= Modelica.Constants.g_n "Acceleration of gravity";
parameter SI.Temperature Tref=273.16 "Reference temperature for specific enthalpy computation";
parameter SI.SpecificHeatCapacity cp=4186 "Cp of the fluid";
SI.MassFlowRate w_pump "Mass flow rate from the pump";
SI.Pressure p1 "Pump discharge pressure";
SI.Pressure p2 "Storage tank inlet pressure";
SI.Pressure dp_pump "Pump dp";
SI.Pressure dp_valve "Valve dp";
Real sqrt_dp "Regularized sqrt(dp)";
SI.SpecificEnthalpy h0 "Pump inlet specific enthalpy";
SI.SpecificEnthalpy h1 "Pump discharge specific enthalpy";
SI.Power $W$;
SI.Length $y$ (start=40, fixed=true) "Reservoir level";
Real $\eta$ (final unit="1") = ($p_1 - \text{patm}$)*$w_pump$/\rho/W "Pump efficiency";
SI.Temperature $T_1$ "Pump discharge temperature";
SI.Time $\tau$ = 1 "Time constant of temperature sensor";
equation
dp_pump = $p_1 - \text{patm}$ "Pump dp";
dp_valve = $p_1 - p_2$ "Valve dp";
dp_pump = dp0 - a1*$w_pump^2"First
characteristic curve of the pump";
w_pump = $K_v*\sqrt{dp}$ "Regularized sqrt(dp)";
sqrt_dp = dp_valve/(dp_valve^2 + dp_small^2) ^0.25 "Valve equation";
$W = a2 + a3*w_pump"Second characteristic
curve of the pump";
w_pump*(h1 - h0) = $W"Energy balance of the pump";
rho*A*der($y$) = $w_pump + w_extra"Mass
balance of the reservoir";
p2 = rho*g*y + $\text{patm}$ "Static head of the storage tank";
h0 = $c_p*(T0 - Tref)$ "h(T) relationship";
h1 = $c_p*(T1 - Tref)$ "h(T) relationship";
annotation (Documentation (info="<html>
<p>This model describes a simple hydraulic system
with a pump, followed by a valve, which fills a
reservoir.</p>
<p>The initial value of the level of the reservoir
is too high for the pump sizing, so the
pressure $p_2$ is too high and consequently the
nonlinear algebraic system of equations that
determines $p_1$ and $w_pump$ has no solution.</p>
<p>It is possible to find a solution to the system
either by lowering the initial value of $y$, and
thus the pressure $p_2$, or by increasing the
value of the parameter $dp0$, increasing the head
the pump can provide.</p>
<p>The debugger should show the dependency of the
nonlinear system of equations on the parameters
$dp0$, a1, a2, a3, and $K_v$ (also showing their
values), as well as the dependency on $p2$ (which

\[\text{\textbf{\textit{...}}}\]"
has a too high value). Once one understands that p2 is too high, it should be possible to continue the analysis, looking at the equation that determines p2, which in turn depends on the value of the state y, which is the root cause of the problem. </p>

The nonlinear system that cannot be solved has five unknowns: \( w_{\text{pump}}, dp_{\text{pump}}, dp_{\text{valve}}, \sqrt{dp}, \) and \( p_1 \), which can be easily reduced to one by using \( dp_{\text{pump}} \) as a tearing variable. It should be possible to track the values of all five variables during the iterations of the Newton algorithm.</p>

This model describes a simple hydraulic system with a pump, followed by a valve, which fills a reservoir. </p>

The reservoir is filled both by the pump and by an extra source. The mass flow rate of the pump \( w_{\text{pump}} \) is determined by a nonlinear system with five unknowns: \( w_{\text{pump}}, dp_{\text{pump}}, dp_{\text{valve}}, \sqrt{dp}, \) and \( p_1 \), which basically computes the operating point of the pump as the intersection between the pump head curve and the load (valve + reservoir head) curve. Note that these curves have two intersections (see NonlinearSolverFailure3). As the level increases, \( w_{\text{pump}} \) is reduced, and the two intersections get closer to each other, until at time \( t = 269 \) they collide, making the system singular. As the level increases further due to the extra source, this system ceases to have any solution. This is a typical bifurcation pattern in nonlinear systems.
The debugger can show that the condition number of the Jacobian of the nonlinear system gets bigger and bigger as the critical time when the two operating curves become tangent to each other, suggesting that this system becomes singular for some reason. Understanding the reason why this happens requires physical insight into the model.

The model can be fixed by adding some mass storage depending on the pressure \( p_1 \), in order to avoid the singularity in determining \( p_1 \), and also by using a more realistic cubic curve for the pump model, so that when the limit level is reached, the solution will jump to a big negative pump flow. Again, this requires physical insight into the validity range of the implemented model.

This model describes a simple hydraulic system with a pump, followed by a valve, which fills a reservoir. As the reservoir level increases, the flow rate \( w_{\text{pump}} \) goes to zero. When it does, the energy balance equation causes the specific enthalpy \( h_1 \), and thus the temperature \( T_1 \), to go to infinity. The temperature \( T_1 \) is the input of a first-order linear system, representing the temperature sensor dynamics. If a variable step-size solver with error control is used, it
will try to compute the state trajectory, which also goes to infinity, so the solver eventually gets stuck at time \( t = 664 \).

The debugger should identify the state variable \( (T_s, \text{in this case}) \) whose error estimate is causing the step size to be reduced, then suggest the user to look at how its derivative \( \text{der}(T_s) \) is computed. It will be shown that it depends on \( T_1 \), which goes to infinity. \( T_1 \) in turn depends on \( h_1 \), which also goes to infinity. Finally, \( h_1 \) depends on the energy balance equation, which depends on \( w_{\text{pump}} \) - at that point it will become apparent that as the flow rate \( w_{\text{pump}} \) goes to zero, the model will become ill-posed. The solution in this case is to change the pump model, by adding to the energy balance some dynamic energy storage and/or some heat transfer to the ambient.

This model describes a simple hydraulic system with a pump, followed by a valve, which fills a reservoir. The operating point of the pump is determined by a nonlinear system with five unknowns: \( w_{\text{pump}}, dp_{\text{pump}}, dp_{\text{valve}}, \sqrt{dp}, \text{and } p_1 \). Dymola selects \( dp_{\text{pump}} \) as a tearing variable. At time \( t=0 \), this system has two solutions, one with positive \( w_{\text{pump}} \), and the other one with negative \( w_{\text{pump}} \). If the start value of the tearing variable \( dp_{\text{pump}} \) is chosen incorrectly, the solver will converge to the negative solution, then lock onto it for the rest of the simulation.
When the user sees the negative \( w_{\text{pump}} \), he/she should be able to analyze how this value was found at time \( t = 0 \). The debugger should show that \( w_{\text{pump}} \) is solved by that nonlinear system, and show the values of the tearing variables and of the torn variables at each iteration. It will then become apparent that the start value of the tearing variable \( dp_{\text{pump}} \) leads to a negative value of the torn variable \( w_{\text{pump}} \), leading to the solution of the problem, i.e., changing the start value of \( dp_{\text{pump}} \) to a value that allows to converge on the desired solution.

\[
\text{WrongInitialSolutionSelected;}
\]

\[
\text{SolverFailure;}
\]

\[
\text{package Functions}
\]

\[
\text{function f_sign "Computes the signum function"}
\]

\[
\text{input Real x;}
\]

\[
\text{output Real y;}
\]

\[
\text{algorithm}
\]

\[
\text{if x > 0 then}
\]

\[
y := 1;
\]

\[
\text{elseif x < 0 then}
\]

\[
y := -1;
\]

\[
\text{else}
\]

\[
y := 0;
\]

\[
\text{end if;}
\]

\[
\text{annotation (Inline=false);}
\]

\[
\text{end f_sign;}
\]

\[
\text{end Functions;}
\]

\[
\text{annotation (uses(Modelica(version="3.2")))};
\]

\[
\text{end Debugging;}
\]
Appendix F

MetaModelica Example: Expression Interpreter

The following files make up the complete exp1 language interpreter, including external routines for parsing the expressions to be evaluated.

Listing F.1: Expression interpreter: Exp1.mo.

```modelica
package Exp1

uniontype Exp
  record INTconst
    Integer integer;
  end INTconst;

  record ADDop
    Exp exp1;
    Exp exp2;
  end ADDop;

  record SUBop
    Exp exp1;
    Exp exp2;
  end SUBop;

  record MULop
    Exp exp1;
    Exp exp2;
  end MULop;

end Exp1;
```

211
record DIVop
  Exp exp1;
  Exp exp2;
end DIVop;

record NEGop
  Exp exp;
end NEGop;
end Exp;

function eval "Abstract syntax of the language
Exp1: Evaluation semantics of Exp1"
input Exp inExp;
output Integer outInteger;
algorithm
  outInteger := match (inExp)
  local
    Integer ival,v1,v2;
    Exp e1,e2,e;
    /* evaluation of an integer node is the
    integer itself */
    case (INTconst(integer = ival)) then ival;
    /*
    Evaluation of an addition node PLUSop is v1 +
    v2, 
    where v1 and v2 are the evaluated results of
    its children e1 and e2
    Subtraction, multiplication, division 
    operators have similar specs.
    */
    case (ADDop(exp1 = e1,exp2 = e2))
      algorithm
        v1 := eval(e1);
        v2 := eval(e2);
        then v1 + v2;
    case (SUBop(exp1 = e1,exp2 = e2))
      algorithm
        v1 := eval(e1);
        v2 := eval(e2);
        then v1 - v2;
    case (MULop(exp1 = e1,exp2 = e2))
      algorithm
        v1 := eval(e1);
        v2 := eval(e2);
        then v1*v2;
case (DIVop(exp1 = e1, exp2 = e2))
algorithm
  v1 := eval(e1);
  v2 := eval(e2);
  then intDiv(v1, v2);
case (NEGop(exp = e))
algorithm
  v1 := eval(e);
  then -v1;
end match;
end eval;
end Exp1;

Listing F.2: Expression interpreter: Parse.mo.

package Parse
import Exp1;

function yyparse
  output Integer i;
  external "C" annotation(Library = {"lexer.o", "parser.o"});
end yyparse;

function getAST
  output Exp1.Exp exp;
  external "C" annotation(Library = {"lexer.o", "parser.o"});
end getAST;

function parse
  output Exp1.Exp exp;
  algorithm
    0 := yyparse();
    exp := getAST();
end parse;
end Parse;

Listing F.3: Expression interpreter: lexer.l.

%{
#define YYSTYPE void*
#include "parser.h"
### APPENDIX F. METAMODELICA EXAMPLE: EXPRESSION INTERPRETER

```c
#include "meta/meta_modelica.h"
extern struct record_description
    Exp1__Exp_INTconst__desc;
#define Exp1__INTconst (X1) ( mmc_mk_box2 (3,& Exp1__Exp_INTconst__desc, (X1)) )

%%
digit           ("0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9")
digits          {digit}+
junk            \n
letter           [%_A-Za-z. ]
letters          {letter}+

%%
{digits}         { yylval = absyn_integer(yytext);
                   return T_INTCONST; }
"+"             return T_ADD;
"-"             return T_SUB;
"*"             return T_MUL;
"/"             return T_DIV;
"("             return T_LPAREN;
")"             return T_RPAREN;
{junk}+          ;
{letters}        return T_ERR;

%%

void* absyn_integer(char *s)
{
    return Exp1__INTconst(mmc_mk_icon(atoi(s)));
}

Listing F.4: Expression interpreter: parser.y.

%
#include <stdio.h>
#include <stdlib.h>

#define YYSTYPE void *

#%
void* absyntree;

int yyerror(char *s)
{
    extern int yylineno;
    fprintf(stderr,"Syntax error at or near line %d.
    \n",yylineno);
    exit(1);
}

int yywrap()
{
    return 1;
}

#include "meta/meta_modelica.h"
void* getAST()
{
    return absyntree;
}

extern struct record_description
    Exp1_Exp_ADDop__desc;
extern struct record_description
    Exp1_Exp_SUBop__desc;
extern struct record_description
    Exp1_Exp_MULop__desc;
extern struct record_description
    Exp1_Exp_DIVop__desc;
extern struct record_description
    Exp1_Exp_NEGop__desc;

#define Exp1__ADDop (X1,X2) ( mmc_mk_box3 (4,&
    Exp1_Exp_ADDop__desc, (X1),(X2)))
#define Exp1__SUBop (X1,X2) ( mmc_mk_box3 (5,&
    Exp1_Exp_SUBop__desc, (X1),(X2)))
#define Exp1__MULop (X1,X2) ( mmc_mk_box3 (6,&
    Exp1_Exp_MULop__desc, (X1),(X2)))
#define Exp1__DIVop (X1,X2) ( mmc_mk_box3 (7,&
    Exp1_Exp_DIVop__desc, (X1),(X2)))
#define Exp1__NEGop (X1) ( mmc_mk_box2 (8,&
    Exp1_Exp_NEGop__desc,(X1)))

%}
APPENDIX F. METAMODELICA EXAMPLE:

**EXPRESSION INTERPRETER**

```plaintext
/* Yacc BNF Syntax of the expression language Exp1 */

program : expression
    { absyntree = $1; }

expression : term
    | expression T_ADD term
    { $$ = Exp1__ADDop($1,$3);}
    | expression T_SUB term
    { $$ = Exp1__SUBop($1,$3);}

term : u_element
    | term T_MUL u_element
    { $$ = Exp1__MULop($1,$3);}
    | term T_DIV u_element
    { $$ = Exp1__DIVop($1,$3);}

u_element : element
    | T_SUB element
    { $$ = Exp1__NEGop($2);}

element : T_INTCONST
    | T_LPAREN expression T_RPAREN
    { $$ = $2;}
```

%token T_INTCONST
%token T_LPAREN T_RPAREN
%token T_ADD
%token T_SUB
%token T_MUL
%token T_DIV
%token T_GARBAGE
%token T_ERR

%%
Appendix G

TLM Example

The following is the example model used in Chapter 7\(^1\).

Listing G.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 will leave it as an exercise to the reader to find it.
Connector.C pressure;
    parameter Real P;
  equation
    pressure.c = P;
    pressure.Zc = 0;
  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 = der(left.p);
derleftp = der(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


REFERENCES


REFERENCES


REFERENCES


[91] Robin Milner, Mads Tofte, Robert Harper, and David Mac-
Queen. *The Definition of Standard ML*. Cambridge, MA, 

[92] Modelica Association. *Minutes of the Modelica Design Meeting*
67. Atlanta, Georgia, Sept. 2010. URL: http://modelica.org
(visited on 2011-10-07).

(visited on 2014-01-31).

Language for Physical Systems Modeling, Language Specifica-
URL: http://www.modelica.org/ (visited on 2015-03-06).

[95] Iakov Nakhimovski. “Contributions to the Modeling and
Simulation of Mechanical Systems with Detailed Contact
Analyses”. Doctoral thesis No 1009. Sweden: Department of
Computer and Information Science, Linköping University,
Apr. 2006. URL: http://urn.kb.se/resolve?urn=urn:nbn:

[96] Nicholas Nethercote and Julian Seward. “Valgrind: a frame-
work for heavyweight dynamic binary instrumentation”. In: *Proceedings of the 2007 ACM SIGPLAN conference on Pro-
gramming language design and implementation*. PLDI ’07.
San Diego, California, USA, 2007, pp. 89–100. DOI:
10.1145/1250734.1250746.

Hybrid Modeling from an Object-Oriented Perspective”. In: *Proceedings of the 1st International Workshop on Equation-
Based Object-Oriented Modeling Languages and Tools*.
Ed. by Peter Fritzson, François Cellier, and Christoph Nytsch-Geusen. Berlin, Germany: Linköping University


REFERENCES


| No 503 | Johan Ringström: Compiler Generation for Data-Parallel Programming Languages from Two-Level Semantics Specifications, 1997, ISBN 91-7219-045-0 |


Linköping Studies in Arts and Science


Linköping Studies in Statistics


Linköping Studies in Information Science


